Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____інфокомунікацій_____
(повна назва)
Кафедра _____інформаційно-вимірювальних технологій_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____

_____Тестування взаємодії та узгодженості даних у мікросервісах_____
_____
(тема)

Виконав:
здобувач _2_ року навчання,
групи ____ЗЯ$_{м}$-23-1___
_____Якимович М.В._____
(прізвище, ініціали)

Спеціальність __175_____
Інформаційно-вимірювальні технології_____
(код і повна назва спеціальності)
Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)
Освітня програма ___Забезпечення якості___
_____
(повна назва освітньої програми)

Керівник _____доц. Запорожець О.В._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ ____Захаров І.П.____
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет_____інфокомунікацій_____

Кафедра_____інформаційно-вимірювальних технологій_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність____175 Інформаційно-вимірювальні технології_____
(код і повна назва)

Тип програми _____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____Забезпечення якості_____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві _____Якимовичу Микиті Вадимовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи ____Тестування взаємодії та узгодженості даних у мікросервісах_____
_____
_____

затверджена наказом університету від _12_ __листопада__ 2024 р. № __1202 Ст___

2. Термін подання здобувачем роботи до екзаменаційної комісії _10_ __січня__ 2025 р.

3. Вихідні дані до роботи _____
__1. Об'єкт дослідження: програмні системи на основі мікросервісної архітектури._____
__2. Мета дослідження: тестування взаємодії між мікросервісами, перевірка узгодженості даних._____
_____
_____
_____
_____
_____
_____

4. Перелік питань, що потрібно опрацювати в роботі _____
__1. Сучасні технології мікросервісної архітектури._____
__2. Тестування як основний інструмент забезпечення якості при створенні програмного продукту._____
__3. Опис функціоналу та архітектури мікросервісів._____
__4. Опис реалізації мікросервісів._____
__5. Тестування взаємодії мікросервісів._____
__6. Результати тестування узгодженості даних між мікросервісами._____
_____
_____

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) _____
Слайди презентації кваліфікаційної роботи.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Визначення бізнес-логіки мікросервісів	25.11.2024 – 27.11.2024	
2	Вибір технологій мікросервісів, а саме мови	28.11.2024 – 05.12.2024	
	програмування, сховища даних, комунікації		
3	Створення мікросервісів	06.12.2024 – 15.12.2024	
4	Контейнеризація мікросервісів	15.12.2024 – 22.12.2024	
5	Написання автоматичних тестів	23.12.2024 – 31.12.2024	
6	Оформлення пояснювальної записки	01.01.2025 –09.01.2025	
7	Представлення закінченої кваліфікаційної	10.01.2025	
	роботи на кафедру		

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Запорожець О.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до магістерської кваліфікаційної роботи містить 52 сторінки, 4 рисунки, перелік посилань з 14 назв.

ТЕСТУВАННЯ, ФУНКЦІОНАЛ, МІКРОСЕРВІС, МІКРОСЕРВІСНА АРХІТЕКТУРА, УЗГОДЖЕНІСТЬ ДАНИХ

Об’єкт дослідження – програмні системи на основі мікросервісної архітектури.

Мета роботи – тестування взаємодії між мікросервісами, перевірка узгодженості даних

У кваліфікаційній роботі розглянуто сучасні технології мікросервісної архітектури, описано основні етапи створення та реалізації мікросервісів. Окрема увага приділена тестуванню як ключовому інструменту забезпечення якості при розробці програмного забезпечення. Проведено тестування взаємодії між мікросервісами та аналіз узгодженості даних, що включає оцінку результатів їх роботи в системі.

ABSTRACT

The explanatory note to the master's qualification work contains 52 pages, 4 figures, a list of references with 14 titles.

TESTING, FUNCTIONAL, MICROSERVICE, MICROSERVICE ARCHITECTURE, DATA CONSISTENCY

The object of research is software systems based on microservice architecture.

The purpose of the work is to test the interaction between microservices, to check the consistency of data

In the qualification work, modern technologies of microservice architecture are considered, the main stages of creation and implementation of microservices are described. Special attention is paid to testing as a key quality assurance tool in software development. Testing of interaction between microservices and analysis of data consistency, including evaluation of the results of their work in the system, were carried out.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – інтерфейс програмування застосунків

HTTP/HTTPS – протоколи передачі даних

RPC – виклик віддалених процедур

SQL / noSQL – мова програмування для взаємодії користувача з базами даних

ВСТУП

У сучасному світі інформаційних технологій мікросервісна архітектура стала одним з найбільш популярних підходів до розробки програмного забезпечення. Вона дозволяє розділяти складні системи на набір невеликих, незалежних один від одного сервісів, кожен з яких виконує конкретну функцію і може бути розроблений, розгорнутий та масштабований окремо. Це забезпечує високу гнучкість, адаптивність і продуктивність системи в умовах швидкозмінного бізнес-середовища.

Проте з впровадженням мікросервісної архітектури виникає низка викликів, пов'язаних із забезпеченням якості програмного продукту, зокрема тестуванням взаємодії та узгодженості даних між окремими мікросервісами. Розробка програмних систем на основі мікросервісів вимагає не тільки розуміння процесів розробки, але й ефективних механізмів контролю якості, адже втрата узгодженості або некоректна взаємодія між сервісами може призвести до значних збоїв у роботі системи.

Тестування займає ключове місце у життєвому циклі розробки програмного забезпечення, особливо в умовах мікросервісної архітектури, де кожен компонент є незалежним, але взаємопов'язаним з іншими. Від якості тестування залежить надійність усієї системи. Правильне тестування дозволяє не лише знайти дефекти, а й забезпечити стабільну роботу програми в реальних умовах експлуатації.

Мета цієї кваліфікаційної роботи полягає у дослідженні сучасних підходів до тестування взаємодії та узгодженості даних у мікросервісах. У роботі розглядаються основні принципи мікросервісної архітектури, описується процес розробки мікросервісів, а також досліджуються техніки тестування, що дозволяють забезпечити їх коректну взаємодію. Особлива

увага приділяється методам, які дозволяють оцінити узгодженість даних між мікросервісами, що є критичним аспектом у розподілених системах.

Таким чином, у роботі будуть проаналізовані як теоретичні аспекти мікросервісної архітектури та тестування, так і практичні результати, отримані в ході реалізації та тестування мікросервісної системи.

1 СУЧАСНІ ТЕХНОЛОГІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

Мікросервісна архітектура — це сучасний підхід до розробки програмних систем, який став одним із ключових трендів у галузі інформаційних технологій. Ця архітектурна парадигма є результатом еволюції від монолітних додатків до більш гнучких, модульних систем, що дозволяють легко адаптуватися до змін у бізнес-вимогах та швидше масштабуватися. Мікросервіси змінюють спосіб, у який створюється програмне забезпечення, розбиваючи великі, складні додатки на набір невеликих сервісів, які взаємодіють один з одним через чітко визначені API. У цьому розділі ми детально розглянемо ключові аспекти мікросервісної архітектури, її переваги, недоліки, а також сучасні технології, що використовуються для її реалізації.

1.1 Визначення мікросервісної архітектури

Мікросервіси — це архітектурний стиль, у якому програмна система складається з ряду невеликих автономних служб, кожна з яких відповідає за конкретну частину бізнес-логіки. Кожен мікросервіс працює незалежно, має свою базу даних та не залежить від інших компонентів системи для свого функціонування. Такий підхід суттєво відрізняється від традиційної монолітної архітектури, де весь код і логіка розміщені в одному великому додатку, що призводить до ускладнення підтримки і масштабування системи.

Основні характеристики мікросервісної архітектури включають:

– незалежність розробки і розгортання. Кожен мікросервіс можна розробляти, тестувати, розгортати і масштабувати незалежно від інших;

– автономність. Мікросервіси не залежать від спільної бази даних або загальних ресурсів; кожен сервіс управляє своїми даними;

– гнучкість. Оскільки кожен мікросервіс є незалежним, його можна масштабувати окремо, виходячи з конкретних вимог;

– чітке визначення меж. Взаємодія між мікросервісами відбувається через API (найчастіше HTTP/REST або gRPC), що дозволяє чітко визначити, які дані обмінюються між сервісами.

1.2 Переваги мікросервісної архітектури

Масштабованість. Однією з ключових переваг мікросервісів є можливість масштабування окремих компонентів системи. Якщо певний мікросервіс піддається більшому навантаженню, ніж інші, його можна легко масштабувати без необхідності змінювати всю систему.

Гнучкість у виборі технологій. Оскільки кожен мікросервіс розробляється незалежно, розробники можуть вибирати різні мови програмування, фреймворки та технології для кожного з них. Це дозволяє вибирати найкращі інструменти для вирішення конкретних задач, не обмежуючи весь проект однією технологією.

Простота розгортання та підтримки. Мікросервіси можна розгортати окремо, що робить процес оновлення та виправлення помилок значно простішим. Якщо один мікросервіс потребує оновлення, це не впливає на інші частини системи, що знижує ризик виникнення загальних збоїв.

Простота управління командами. Мікросервісний підхід дозволяє легко розподілити роботу між командами розробників. Кожна команда може бути відповідальною за певний мікросервіс, що дозволяє їм фокусуватися на конкретному аспекті системи, не втручаючись у роботу інших команд.

Відмовостійкість. Оскільки мікросервіси працюють незалежно, відмова одного з них не призводить до краху всієї системи. Це робить систему більш стійкою до збоїв і дозволяє швидко виявляти та ізолювати проблеми.

1.3 Недоліки мікросервісної архітектури

Незважаючи на численні переваги, мікросервісна архітектура має і свої виклики:

– складність управління. Збільшення кількості незалежних сервісів призводить до складності управління системою в цілому. Потрібні інструменти для моніторингу, логування та управління мікросервісами, що може вимагати значних ресурсів;

– мережеві затримки. Мікросервіси взаємодіють через мережу, що може призводити до затримок і підвищувати навантаження на інфраструктуру. Для оптимізації таких взаємодій потрібні спеціальні підходи, такі як кешування або балансування навантаження;

– узгодженість даних. Через те, що кожен мікросервіс має свою базу даних, забезпечення узгодженості даних між ними може бути складним завданням. Необхідно реалізовувати спеціальні механізми для синхронізації даних і уникнення конфліктів;

– ускладнене тестування. Оскільки мікросервіси взаємодіють один з одним через мережу, тестування всієї системи вимагає врахування різних аспектів взаємодії, таких як мережеві збої, неправильні відповіді або некоректна поведінка окремих сервісів.

1.4 Сучасні технології для реалізації мікросервісів

У процесі розробки мікросервісних архітектур використовується широкий спектр технологій, які дозволяють ефективно будувати, розгортати та управляти системами.

1.4.1 Контейнеризація та оркестрація

Одним із ключових компонентів мікросервісної архітектури є використання контейнерів для ізоляції кожного сервісу. Docker став стандартом для контейнеризації, дозволяючи пакувати мікросервіси разом з усіма залежностями у відокремлені середовища. Це полегшує їх розгортання та масштабування.

Для управління великою кількістю контейнерів використовується система оркестрації, такої як Kubernetes або Docker Compose. Кожен з них надає можливість автоматично керувати розгортанням, масштабуванням і балансуванням навантаження між контейнерами мікросервісів. Це забезпечує високу доступність і стійкість системи, а також спрощує управління складними розподіленими середовищами.

Kubernetes став стандартом у світі оркестрації мікросервісів завдяки своїм можливостям автоматизації процесів розгортання, управління життєвим циклом сервісів і обробки відмов. Основні функції Kubernetes включають:

– автоматичне масштабування: система здатна автоматично розподіляти ресурси залежно від навантаження на мікросервіси;

– моніторинг та відновлення: Kubernetes відстежує стан контейнерів і автоматично перезапускає їх у випадку збоїв;

– управління конфігурацією та секретами: система забезпечує безпечний розподіл конфігураційних параметрів і секретних даних між сервісами.

1.4.2 API Gateway

У мікросервісній архітектурі часто використовується API Gateway — посередник, що об'єднує запити від клієнтів і розподіляє їх між мікросервісами. API Gateway виконує такі функції, як маршрутизація запитів, балансування навантаження, управління безпекою, кешування та збирання

аналітичних даних. Популярні рішення для API Gateway включають Kong, Traefik і Nginx.

1.4.3 Сервісна сітка (Service Mesh)

Ускладнена взаємодія між мікросервісами вимагає спеціальних інструментів для управління мережею і зв'язками між сервісами. Service Mesh — це технологія, яка дозволяє спрощувати управління мережевою взаємодією між мікросервісами, вирішуючи такі завдання, як балансування навантаження, моніторинг, маршрутизація запитів, контроль доступу та безпеки.

Одним із найпопулярніших рішень для сервісної сітки є Istio, яке забезпечує комплексне управління мережею та дозволяє контролювати всі аспекти взаємодії між сервісами без змін у коді додатків.

1.4.4 Управління базами даних у мікросервісах

У мікросервісній архітектурі кожен сервіс має власну базу даних, що дає можливість ізолювати дані і уникати конфліктів. Проте це також створює виклики з узгодженістю даних. Для вирішення цих питань використовуються різні підходи, такі як Event Sourcing (зберігання кожної зміни у вигляді події) і CQRS (Command Query Responsibility Segregation), які дозволяють окремо обробляти запити на зміну і читання даних.

Технології баз даних, такі як NoSQL (MongoDB, Cassandra) і SQL (PostgreSQL, MySQL), застосовуються для різних потреб мікросервісів. NoSQL бази популярні для систем з великою кількістю неструктурованих даних, тоді як SQL бази використовуються для забезпечення строгих транзакційних вимог.

1.5 Сучасні фреймворки для розробки мікросервісів

Розробка мікросервісів вимагає використання спеціалізованих фреймворків, які забезпечують необхідну інфраструктуру для швидкої та ефективної реалізації окремих сервісів. До таких фреймворків належать:

– Spring Boot (для Java) — один із найпопулярніших фреймворків для створення мікросервісів на основі Java. Він пропонує прості механізми для розробки REST API, роботи з базами даних, конфігурації та інтеграції з іншими сервісами;

– Micronaut — полегшений фреймворк для мікросервісів, розроблений для оптимальної роботи з хмарними системами, підтримує швидкий час запуску і малий розмір контейнерів;

– Node.js — платформа для розробки мікросервісів на JavaScript або TypeScript, що дозволяє швидко створювати високопродуктивні, масштабовані сервіси;

– .NET Core — кросплатформний фреймворк для розробки мікросервісів на основі технологій Microsoft, який підтримує інтеграцію з іншими хмарними сервісами.

1.6 Майбутнє мікросервісної архітектури

Мікросервісна архітектура продовжує активно розвиватися, і її майбутнє пов'язане з появою нових технологій і підходів до організації розподілених систем. До найбільш перспективних напрямів належать:

– функції як сервіс (FaaS) — модель, при якій розробники зосереджуються лише на бізнес-логіці, залишаючи управління інфраструктурою провайдеру хмарних послуг. Цей підхід також відомий як серверлес-архітектура (Serverless), де функції виконуються лише за необхідності;

– розширення можливостей сервісних сіток — сервісні сітки (Service Mesh) стають важливою частиною мікросервісної архітектури, і з розвитком технологій вони продовжать вдосконалюватися, забезпечуючи ще більший контроль і безпеку взаємодії між сервісами;

– автоматизація управління системами — розвиток технологій штучного інтелекту та машинного навчання дозволить автоматизувати процеси моніторингу, оптимізації та масштабування мікросервісів, що значно зменшить втручання людини в управління системами.

Таким чином, мікросервісна архітектура залишається одним із найсучасніших і найгнучкіших підходів до розробки програмних систем, і її впровадження забезпечує значні переваги у швидкості розробки, масштабованості та надійності. Технології, що використовуються для її реалізації, постійно вдосконалюються, і їх подальший розвиток дозволить вирішувати нові виклики та створювати ще ефективніші системи.

2 ТЕСТУВАННЯ ЯК ОСНОВНИЙ ІНСТРУМЕНТ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРИ СТВОРЕННІ ПРОГРАМНОГО ПРОДУКТУ

Тестування програмного забезпечення є ключовим етапом у процесі його розробки, оскільки саме воно дозволяє виявити помилки, недоліки та невідповідності в роботі програми до того, як вона буде передана кінцевим користувачам. У цьому розділі детально розглядаються всі аспекти тестування, його місце в життєвому циклі розробки програмного забезпечення, основні принципи та методи, які забезпечують високу якість продукту.

2.1 Що таке тестування?

Тестування — це процес перевірки програмного забезпечення з метою виявлення помилок та перевірки відповідності програми очікуванням і вимогам. Тестування проводиться для того, щоб переконатися, що продукт працює коректно, відповідає вимогам користувачів і не має дефектів, які можуть призвести до некоректної роботи або відмови системи.

Тестування програмного забезпечення можна розділити на дві великі категорії:

– функціональне тестування: перевіряє, чи відповідає програма функціональним вимогам. Цей вид тестування орієнтований на те, як програма виконує свої завдання, і чи відповідає її поведінка заданим специфікаціям;

– нефункціональне тестування: перевіряє аспекти роботи програми, які не пов’язані з функціональністю, такі як продуктивність, безпека, зручність використання і стабільність.

2.2 Для чого потрібно тестувати програми?

Метою тестування є виявлення дефектів на ранніх етапах розробки, щоб уникнути серйозних проблем під час експлуатації продукту. Тестування дозволяє:

– забезпечити якість продукту: виявлення помилок та дефектів дозволяє їх усунути до того, як продукт буде розгорнуто для кінцевих користувачів, що значно знижує ризик появи проблем після релізу;

– перевірити відповідність вимогам: тестування допомагає переконатися, що програмне забезпечення відповідає вимогам, визначеним на етапі проектування, та забезпечує необхідну функціональність;

– забезпечити стабільність роботи: за допомогою тестування можна перевірити, як система поводить себе під навантаженням, у різних середовищах та за різних сценаріїв використання, що дозволяє гарантувати її стабільність і надійність;

– забезпечити безпеку: тестування безпеки допомагає виявити вразливості та запобігти їх експлуатації зловмисниками, що особливо важливо в сучасних умовах, коли кіберзагрози постійно зростають;

– знизити витрати на виправлення помилок: помилки, знайдені на ранніх етапах розробки, коштують набагато менше в усуненні, ніж ті, які були виявлені на пізніших етапах або після релізу продукту.

2.3 Роль і місце тестування в життєвому циклі розробки ПЗ

Тестування є невід'ємною частиною життєвого циклу розробки програмного забезпечення (Software Development Life Cycle — SDLC). Незважаючи на те, що в минулому тестування часто сприймалося як заключний етап перед випуском продукції, сучасні методології розробки, такі як Agile, передбачають інтеграцію тестування на всіх етапах розробки.

Життєвий цикл розробки ПЗ складається з кількох основних етапів.

Збір вимог: на цьому етапі визначаються функціональні та нефункціональні вимоги до програмного забезпечення. Вже на цьому етапі можна готувати плани тестування, щоб переконатися, що всі вимоги можна буде перевірити.

Проектування: на етапі проектування створюється архітектура програмного продукту. Важливо залучити тестувальників до обговорення, щоб вони могли визначити, як і що потрібно тестувати.

Розробка: на етапі розробки тестувальники розробляють тестові сценарії та проводять тестування окремих модулів системи. Це етап, коли активно використовуються юніт-тести (Unit Testing), інтеграційне тестування та інші види тестування.

Тестування: на цьому етапі тестується вже готова система або її частини. Виконуються різні види тестування: функціональне, нефункціональне, навантажувальне, безпеки тощо.

Випуск та підтримка: після випуску продукту тестувальники можуть продовжувати виконувати регресійне тестування, щоб переконатися, що нові оновлення або зміни не викликають нових помилок.

Роль тестування полягає у виявленні недоліків на кожному з етапів SDLC, забезпечуючи високий рівень якості кінцевого продукту.

2.4 Принципи тестування

Існує кілька основних принципів, якими керуються тестувальники для ефективного проведення тестування. Ці принципи були сформульовані на основі багаторічного досвіду тестування і є основою для будь-якої стратегії тестування.

Тестування показує присутність помилок, а не їх відсутність. Тестування може виявити дефекти, але ніколи не може довести, що дефекти відсутні

повністю. Тому важливо розуміти, що навіть якщо тестування не виявило помилок, це не гарантує, що їх немає.

Вичерпне тестування неможливе. Неможливо протестувати всі можливі комбінації вхідних даних, сценаріїв і середовищ виконання. Тому тестувальники повинні зосередитися на найбільш важливих та критичних областях системи, використовуючи ефективні техніки тест-дизайну.

Раннє тестування. Чим раніше розпочнеться тестування, тим більше помилок можна виявити на ранніх етапах розробки, що дозволить знизити витрати на їх виправлення. Це також допомагає запобігти накопиченню дефектів на пізніших стадіях проєкту.

Скупчення дефектів. Дефекти мають властивість концентруватися в одних і тих самих модулях або частинах системи. Це означає, що, виявивши помилки в певній області, слід ретельно перевіряти її на наявність інших проблем.

Парадокс пестицидів. Якщо одні й ті самі тести виконувати повторно, вони можуть перестати знаходити нові помилки. Тому тести потрібно періодично оновлювати, адаптувати до нових умов і змін у системі, щоб залишатися ефективними.

Тестування залежить від контексту. Тестування різних типів програмного забезпечення вимагає різних підходів. Наприклад, для веб-додатків критично важливо тестувати продуктивність і безпеку, тоді як для програмного забезпечення, що керує апаратними пристроями, важливо перевіряти реальну взаємодію з обладнанням.

Принцип відсутності помилок — ілюзія. Факт відсутності виявлених помилок у програмі не гарантує, що вона корисна і задовольняє потреби користувачів. Навіть ідеально працююча система може не мати цінності, якщо вона не виконує ті завдання, які від неї очікують.

2.5 Техніки тест-дизайну

Для ефективного тестування використовуються різні техніки тест-дизайну, які допомагають організувати процес тестування та вибрати найбільш важливі випадки для перевірки. Основні техніки тест-дизайну включають:

– еквівалентне розбиття: тестувальник ділить всі можливі вхідні дані на групи або класи еквівалентності, де кожен клас очікується як такий, що призведе до однакової поведінки програми. Це дозволяє зменшити кількість тестів, покриваючи різноманітні сценарії в рамках однієї категорії;

– аналіз граничних значень: використовується для тестування граничних умов системи, оскільки саме на кордонах часто виникають помилки. Наприклад, тестування мінімальних і максимальних допустимих значень для полів вводу;

– тестування на основі сценаріїв: це підхід, при якому тестування здійснюється відповідно до реальних сценаріїв використання системи. Такий підхід допомагає переконатися, що програма працює коректно з точки зору кінцевого користувача;

– тестування на основі ризиків: тут тестувальники концентруються на найбільш ризикованих частинах системи, де помилки можуть мати найбільш критичні наслідки;

– регресійне тестування: виконується після внесення змін до програмного забезпечення, щоб переконатися, що нові зміни не вплинули на вже працюючі функції.

2.6 Види тестування

Тестування програмного забезпечення можна поділити на кілька типів залежно від того, що саме перевіряється і на якому етапі розробки:

– юніт-тестування: перевіряє окремі модулі або компоненти системи на рівні коду. Це тестування виконується розробниками для переконання, що кожен компонент функціонує належним чином;

– інтеграційне тестування: перевіряє взаємодію між окремими компонентами або модулями системи, щоб переконатися, що вони працюють разом коректно;

– системне тестування: перевіряє всю систему в цілому на відповідність вимогам. Тут тестуються як функціональні, так і нефункціональні аспекти;

– приймальне тестування (Acceptance Testing): виконується кінцевими користувачами або замовниками для перевірки відповідності системи їх вимогам і очікуванням;

– навантажувальне тестування: перевіряє, як система працює під різними навантаженнями, щоб переконатися, що вона витримує очікувані обсяги трафіку та запитів;

– тестування безпеки: перевіряє систему на вразливості до кіберзагроз і гарантує, що дані користувачів захищені.

3 ОПИС ФУНКЦІОНАЛУ ТА АРХІТЕКТУРИ МІКРОСЕРВІСІВ

Мікросервісна архітектура є сучасним підходом до розробки програмних систем, який дозволяє створювати масштабовані, гнучкі та ефективні рішення. Основна ідея полягає в тому, що додаток розбивається на набір незалежних компонентів (мікросервісів), кожен з яких відповідає за певну функціональність. У цьому розділі детально розглядаються ключові аспекти функціоналу мікросервісів, принципи їх архітектури, а також переваги і виклики, з якими стикаються розробники при їх впровадженні.

3.1 Основні принципи мікросервісної архітектури

Мікросервіси є самостійними одиницями, які взаємодіють одна з одною через добре визначені API. Кожен мікросервіс виконує конкретну бізнес-функцію і може розроблятися, розгортатися та масштабуватися незалежно від інших. Основні принципи мікросервісної архітектури:

– індивідуальність і незалежність. Кожен мікросервіс є незалежним модулем, який виконує окрему задачу. Це дозволяє різним командам працювати над різними мікросервісами паралельно, мінімізуючи залежності між ними;

– децентралізація управління даними. На відміну від монолітної архітектури, де є єдина база даних для всієї системи, у мікросервісній архітектурі кожен мікросервіс може мати свою власну базу даних або сховище, що сприяє незалежності та ефективності;

– поліморфність технологій. Кожен мікросервіс може бути реалізований з використанням різних мов програмування або технологій, що дозволяє використовувати найбільш підходящі інструменти для конкретних задач;

– масштабованість. Мікросервіси легко масштабуються незалежно один від одного, що дозволяє збільшувати потужності лише для тих компонентів, які цього потребують, без необхідності масштабувати всю систему;

– автономне розгортання та оновлення. Кожен мікросервіс може бути розгорнутий і оновлений окремо від інших, що значно спрощує процеси релізу і знижує ризики, пов'язані з внесенням змін.

3.2 Архітектурні патерни мікросервісів

Архітектура мікросервісів будується на основі певних архітектурних патернів, що дозволяють забезпечити високу ефективність і надійність системи. Основні з них включають:

– шлюз API (API Gateway). Шлюз API є єдиною точкою входу для всіх клієнтських запитів до мікросервісів. Він виконує роль проксі, перенаправляючи запити до відповідних мікросервісів. Крім того, API Gateway може виконувати функції авторизації, балансування навантаження, кешування та моніторингу;

– сервіс-відкривач (Service Discovery). Це механізм, який дозволяє мікросервісам знаходити один одного в розподіленій системі. Сервіси реєструються у спеціальному реєстрі, а потім можуть бути знайдені іншими мікросервісами або API Gateway;

– балансувальник навантаження (Load Balancer). У системах мікросервісної архітектури використовуються балансувальники навантаження для рівномірного розподілу запитів між кількома екземплярами мікросервісів, що дозволяє забезпечити стабільну роботу навіть під високими навантаженнями;

– запобіжник (Circuit Breaker). Патерн запобіжника використовується для захисту системи від збою одного або декількох мікросервісів. Коли

мікросервіс перестає відповідати на запити, запобіжник блокує подальші запити до цього сервісу, щоб уникнути подальших проблем і перевантаження;

– подієво-орієнтована архітектура (Event-Driven). Мікросервіси часто взаємодіють між собою за допомогою подій. Використання брокерів повідомлень (Kafka, RabbitMQ) дозволяє мікросервісам відправляти та отримувати події без прямої прив’язки один до одного.

3.3 Опис функціоналу мікросервісів

У мікросервісній архітектурі кожен сервіс виконує одну або декілька окремих функцій, які пов'язані з конкретною бізнес-логікою. Наприклад, у типовій системі електронної комерції можна виділити такі мікросервіси:

– сервіс користувачів (User Service). Відповідає за реєстрацію, аутентифікацію та управління профілями користувачів. Він взаємодіє з базою даних, що зберігає інформацію про користувачів, а також з іншими мікросервісами, які потребують доступу до цієї інформації;

– сервіс товарів (Product Service). Зберігає та керує даними про товари, включаючи їхні характеристики, категорії та доступність. Він забезпечує функціонал пошуку, фільтрації та сортування товарів;

– сервіс кошика (Cart Service). Відповідає за управління кошиками покупців. Він зберігає інформацію про товари, додані в кошик, та взаємодіє з сервісом товарів для отримання актуальних даних про ціни та наявність;

– сервіс замовлень (Order Service). Керує створенням і обробкою замовлень, включаючи процес оформлення покупки, оплати та доставки. Він також може взаємодіяти з сервісами інвентаризації та платежів для перевірки наявності товарів і підтвердження платежів;

– сервіс оплати (Payment Service). Відповідає за інтеграцію з платіжними системами і обробку транзакцій. Він забезпечує безпечне проведення платежів та перевірку статусу оплат;

– сервіс повідомлень (Notification Service). Відправляє повідомлення користувачам про різні події, такі як підтвердження замовлень, статус доставки або інші важливі сповіщення.

3.4 Механізм взаємодії мікросервісів

Мікросервіси взаємодіють один з одним через стандартизовані API, використовуючи різні протоколи. Основними протоколами для комунікації є:

– HTTP/REST. Найбільш поширений протокол взаємодії між мікросервісами. Він забезпечує стандартизовану передачу даних через HTTP-запити (GET, POST, PUT, DELETE) з використанням REST-підходу. REST API дозволяє сервісам взаємодіяти через чітко визначені кінцеві точки;

– gRPC. Це високопродуктивний протокол від Google, який використовує HTTP/2 для передачі даних і забезпечує низьку затримку та ефективність при великій кількості викликів між мікросервісами;

– повідомлення через черги (Message Queues). Використання брокерів повідомлень (наприклад, RabbitMQ, Apache Kafka) дозволяє мікросервісам взаємодіяти асинхронно. Це особливо корисно для подієво-орієнтованих систем, де один мікросервіс публікує події, а інші їх підписуються та обробляють.

3.5 Переваги мікросервісної архітектури

Мікросервісна архітектура має багато переваг у порівнянні з монолітним підходом:

– масштабованість. Мікросервіси можна масштабувати незалежно один від одного, що дозволяє ефективно використовувати ресурси і підтримувати високу продуктивність системи;

– гнучкість розробки. Кожен мікросервіс може бути розроблений окремими командами, використовуючи різні технології, мови програмування та фреймворки. Це дозволяє вибирати найбільш відповідні інструменти для конкретних задач і швидко реагувати на зміни;

– прискорення розгортання. Оскільки мікросервіси можна розгортати незалежно один від одного, це дозволяє скоротити час на розгортання нових версій системи. Нові функції або виправлення помилок можна додавати без необхідності оновлювати всю систему;

– відмовостійкість. У мікросервісній архітектурі відмова одного сервісу не призводить до збоїв у всій системі. Це забезпечується ізоляцією мікросервісів один від одного, що дозволяє системі залишатися функціональною навіть при проблемах з окремими компонентами;

– простота обслуговування. Оскільки мікросервіси мають чітко визначені межі відповідальності, їх простіше підтримувати та оновлювати. Зміни в одному мікросервісі рідко впливають на інші частини системи;

– масштабування команд. У великих організаціях команди розробників можуть бути поділені на менші групи, кожна з яких відповідає за свій мікросервіс. Це дозволяє ефективно керувати великою кількістю розробників і спрощує координацію між ними.

3.6 Виклики мікросервісної архітектури

Незважаючи на численні переваги, мікросервісна архітектура також має свої виклики, які необхідно враховувати під час розробки:

– складність розподілених систем. Мікросервіси є розподіленими системами, що ускладнює управління їх взаємодією, синхронізацією даних та налагодженням. Відстеження проблем у такій системі може бути складним через велику кількість компонентів;

– проблеми з узгодженістю даних. Оскільки кожен мікросервіс має власне сховище даних, підтримання узгодженості між ними може бути складним. Зазвичай використовуються методи eventual consistency, що допускає певні затримки в синхронізації даних між сервісами;

– затримки у комунікації. Виклики, пов’язані з мережею, можуть створювати затримки або втрату запитів між мікросервісами, що може вплинути на продуктивність системи;

– моніторинг та логування. З мікросервісною архітектурою кількість точок, які потрібно моніторити, значно зростає. Це вимагає додаткових інструментів для централізованого логування, трасування запитів і моніторингу стану мікросервісів (наприклад, ELK stack, Prometheus, Grafana);

– управління версіями API. Оскільки мікросервіси взаємодіють через API, важливо ретельно керувати версіями цих API, щоб уникнути збоїв через несумісність версій між сервісами;

– тестування. Тестування мікросервісної архітектури складніше, ніж монолітної, оскільки необхідно тестувати не лише кожен окремий мікросервіс, але й взаємодію між ними. Це вимагає впровадження автоматизованих тестів, таких як інтерфейсні тести, тести взаємодії та системні тести;

– безпека. Захист кожного мікросервісу вимагає окремих підходів до забезпечення безпеки. Це може включати аутентифікацію, авторизацію, шифрування даних і захист мережевих взаємодій між сервісами.

3.7 Приклади використання мікросервісної архітектури

Мікросервіси отримали широке застосування у великих компаніях, які потребують масштабованих та надійних рішень. Нижче наведено кілька прикладів використання мікросервісів у відомих компаніях:

– Netflix. Компанія Netflix використовує мікросервісну архітектуру для забезпечення масштабованості та надійності своєї платформи потокового

відео. Кожна частина сервісу, така як рекомендації, пошук контенту або обробка платежів, реалізована як окремий мікросервіс, що дозволяє легко додавати нові функції та масштабувати окремі компоненти;

– Amazon. Amazon також використовує мікросервіси для управління своїм величезним інтернет-магазином. Завдяки мікросервісній архітектурі компанія може обробляти мільйони транзакцій одночасно, забезпечуючи високу продуктивність і доступність системи;

– Uber. Платформа Uber побудована на мікросервісах, що дозволяє забезпечити безперебійну роботу сервісу в режимі реального часу, обробляючи запити мільйонів користувачів і водіїв по всьому світу. Мікросервіси відповідають за різні аспекти роботи платформи, включаючи маршрутизацію, розрахунок вартості поїздок і підтримку картографічних сервісів.

4 ОПИС РЕАЛІЗАЦІЇ МІКРОСЕРВІСІВ

Один мікросервіс обробляє POST і GET запити для створення, редагування, видалення та отримання зустрічей, а другий — отримує дані від першого мікросервісу на будь-який POST запит, відображає зміни в WebSocket і відправляє відповідні дані на зазначені електронні пошти.

4.1 Переваги використання Node.js для мікросервісів

Node.js — це середовище виконання JavaScript, яке дозволяє розробникам використовувати JavaScript не лише для фронтенду, але й для бекенду. Основні переваги Node.js для мікросервісної архітектури:

– асинхронна обробка запитів. Завдяки асинхронній моделі обробки подій, Node.js може ефективно обробляти велику кількість запитів, що є важливим для масштабованих систем;

– висока продуктивність. Node.js базується на движку V8 від Google, який дуже швидко виконує JavaScript-код, забезпечуючи високу продуктивність;

– масштабованість. Node.js ідеально підходить для мікросервісної архітектури завдяки своїй легкій і модульній природі. Це дозволяє легко розділяти систему на окремі сервіси, які можна масштабувати незалежно;

– єдина мова для фронтенду та бекенду. Оскільки і фронтенд, і бекенд використовують JavaScript, це спрощує роботу команди, знижує кількість помилок і спрощує інтеграцію різних частин системи;

– широка екосистема. Node.js має велику кількість бібліотек і модулів у репозиторії npm, що спрощує реалізацію складних функцій без необхідності розробляти їх з нуля.

4.2 Переваги використання NestJS для мікросервісів

NestJS — це прогресивний фреймворк для Node.js, який використовує TypeScript і надає розробникам можливість створювати масштабовані та підтримувані додатки. Основні переваги NestJS:

– модульність. NestJS використовує модульну архітектуру, що дозволяє легко розбивати додаток на окремі частини (модулі), які можуть бути незалежними та повторно використовуваними;

– інтеграція з іншими технологіями. NestJS забезпечує підтримку таких технологій, як GraphQL, WebSockets, REST API, Microservices та інших, що дозволяє легко інтегруватися з іншими системами та використовувати різні методи комунікації;

– зручність роботи з TypeScript. NestJS спочатку розроблений з використанням TypeScript, що забезпечує статичну типізацію, покращену підтримку інструментів розробки та знижує ймовірність помилок під час розробки;

– інверсія контролю (Inversion of Control, IoC). NestJS використовує інверсію контролю, що дозволяє легко керувати залежностями в додатку через вбудований контейнер залежностей;

– структурований підхід. Завдяки архітектурі, заснованій на контролерах, сервісах і модулях, NestJS забезпечує чітку структуру проекту, що робить код легко підтримуваним і розширюваним;

– підтримка мікросервісів. NestJS має вбудовану підтримку для мікросервісної архітектури та надає інструменти для створення ефективної системи обміну повідомленнями між сервісами.

4.3 Архітектура мікросервісів

4.3.1 Перший мікросервіс (управління зустрічами)

Цей мікросервіс призначений для обробки CRUD операцій (створення, читання, оновлення та видалення) для об’єктів "зустрічі". Він надає два основні методи взаємодії через REST API:

а) POST-запити:

1) створення зустрічі: отримує дані для створення нової зустрічі та зберігає їх у базі даних;
2) редагування зустрічі: отримує дані для редагування існуючої зустрічі за її унікальним ідентифікатором (ID);
3) видалення зустрічі: отримує запит на видалення зустрічі за її унікальним ідентифікатором.

б) GET-запити:

1) отримання однієї зустрічі: повертає дані про конкретну зустріч за її ID;
2) отримання всіх зустрічей: повертає список усіх зустрічей.

Основні елементи реалізації:

– контролери (Controllers). Контролери відповідають за обробку HTTP-запитів і визначають, які дії мають бути виконані для кожного типу запиту. Наприклад, контролер для створення нової зустрічі буде приймати дані з POST-запиту та викликати відповідний сервіс для їх обробки;

– сервіси (Services). Сервіси містять бізнес-логіку для роботи із зустрічами. Вони відповідають за збереження даних у базу даних, їх отримання, оновлення та видалення;

– модулі (Modules). NestJS використовує модульну архітектуру, де кожен функціональний блок додатку може бути ізольованим у свій модуль.

Наприклад, модуль для роботи із зустрічами може містити всі необхідні контролери, сервіси та інші компоненти для цієї функціональності.

4.3.2 Другий мікросервіс (оповіщення через WebSocket та email)

Цей мікросервіс відповідає за отримання даних від першого мікросервісу на будь-який POST запит і виконує дві основні функції:

– логування через WebSocket. Після отримання даних від першого мікросервісу система надсилає повідомлення всім підключеним адміністраторам через WebSocket про зміну стану зустрічей;

– відправка повідомлень на електронну пошту. Окрім WebSocket-оповіщення, система також відправляє електронні листи на вказані адреси з інформацією про зміни в зустрічах.

Основні елементи реалізації:

– WebSocket шлюз. Використовується для взаємодії з адміністраторами в реальному часі. Підключені адміністратори отримують повідомлення про зміни в зустрічах після кожного POST запиту;

– Email-сервіс. Відповідає за формування та відправку електронних листів з інформацією про зміни.

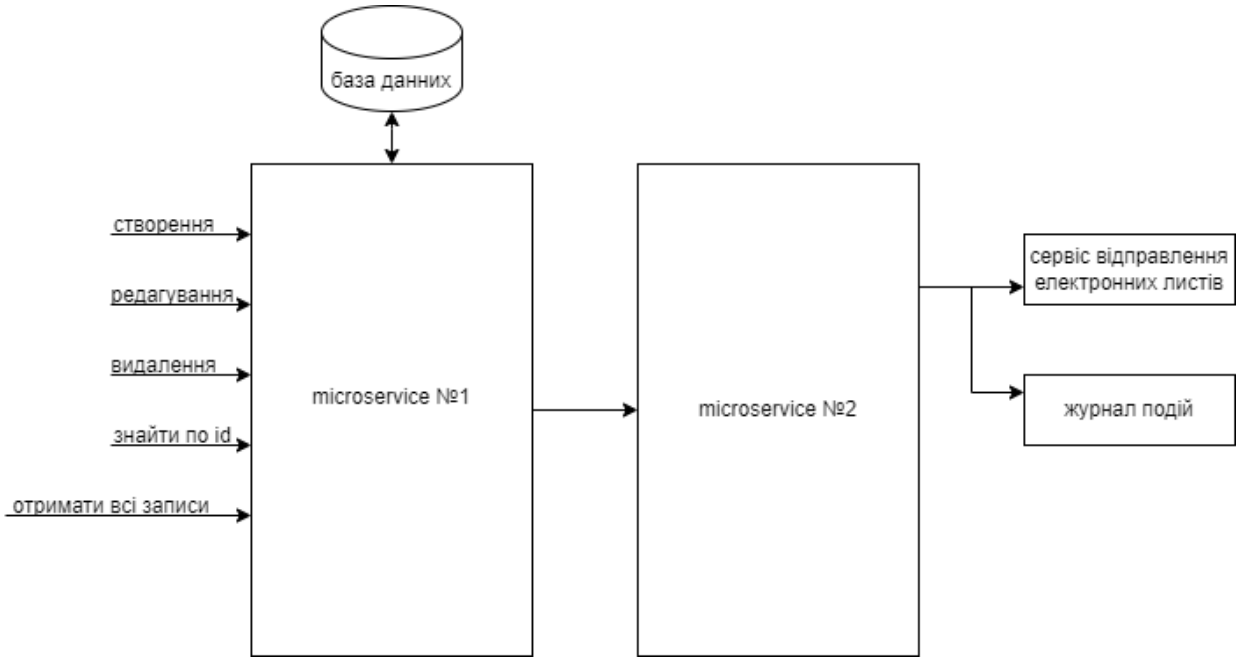

Рисунок 4.1 – Графічна схема

```ts
import { Body, Controller, Get, Param, Post } from '@nestjs/common';
import { MettingsService } from './mettings.service';

@Controller('mettings')
export class MettingsController {
  constructor(private readonly mettingsService: MettingsService) {}

  @Get('all')
  getAll() {
    return this.mettingsService.getAll();
  }

  @Get(':id')
  getOne(@Param('id') id) {
    return this.mettingsService.getOne(id);
  }

  @Post('update')
  update(@Body() { id, metting }) {
    return this.mettingsService.update(id, metting);
  }

  @Post('delete')
  delete(@Body() { id }) {
    return this.mettingsService.delete(id);
  }

  @Post('create')
  create(@Body() metting) {
    return this.mettingsService.create(metting);
  }
}
```

Рисунок 4.2 – Один з контролерів

```ts
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument } from 'mongoose';

@Schema({
  toObject: {
    versionKey: false,
  },
  toJSON: {
    versionKey: false,
  },
})
export class Mettings {
  @Prop({ unique: true })
  id: number;

  @Prop({ default: new Date() })
  datetime: Date;

  @Prop()
  title: string;

  @Prop()
  place: string;

  @Prop()
  members: string[];
}
```

Рисунок 4.3 – Схема об'єкта, що зберігається в базі даних

```
import { Test } from '@nestjs/testing';
import { NotifiesService } from '../notifies.service';
import { INotifies, MessageType } from '../interfaces/notifies.interfaces';

describe(NotifiesService.name, () => {
  //... - тут деякі налаштування для роботи тестів
  // заховані для зменшення обсягу скриншоту
  describe(NotifiesService.prototype.notify.name, () => {
    it('should send message', async () => {
      expect.hasAssertions();

      const sendMessage = jest.spyOn(
        NotifiesService.prototype as any,
        'sendMessage',
      );
      await notifiesService.notify(parameters());

      expect(sendMessage).toHaveBeenCalledTimes(1);
    });
    it('should send 4 messages, but 1 by 1', async () => {
      expect.hasAssertions();

      const sendMessage = jest.spyOn(
        NotifiesService.prototype as any,
        'sendMessage',
      );
      const members = Array(4).fill('4');

      await notifiesService.notify(parameters(members));

      expect(sendMessage).toHaveBeenCalledTimes(4);
    });
  });

  describe(NotifiesService.prototype.notifyV2.name, () => {
    it('should send message', async () => {
      expect.hasAssertions();

      const sendMessage = jest.spyOn(
        NotifiesService.prototype as any,
        'sendMessage',
      );
      await notifiesService.notifyV2(parameters());

      expect(sendMessage).toHaveBeenCalledTimes(1);
    });
    it('should send 10 messages', async () => {
      expect.hasAssertions();

      const sendMessage = jest.spyOn(
        NotifiesService.prototype as any,
        'sendMessage',
      );

      const members = Array(10).fill('10');

      await notifiesService.notifyV2(parameters(members));

      expect(sendMessage).toHaveBeenCalledTimes(10);
    });
  });
});
```

Рисунок 4.4 – Приклади unit-тестів

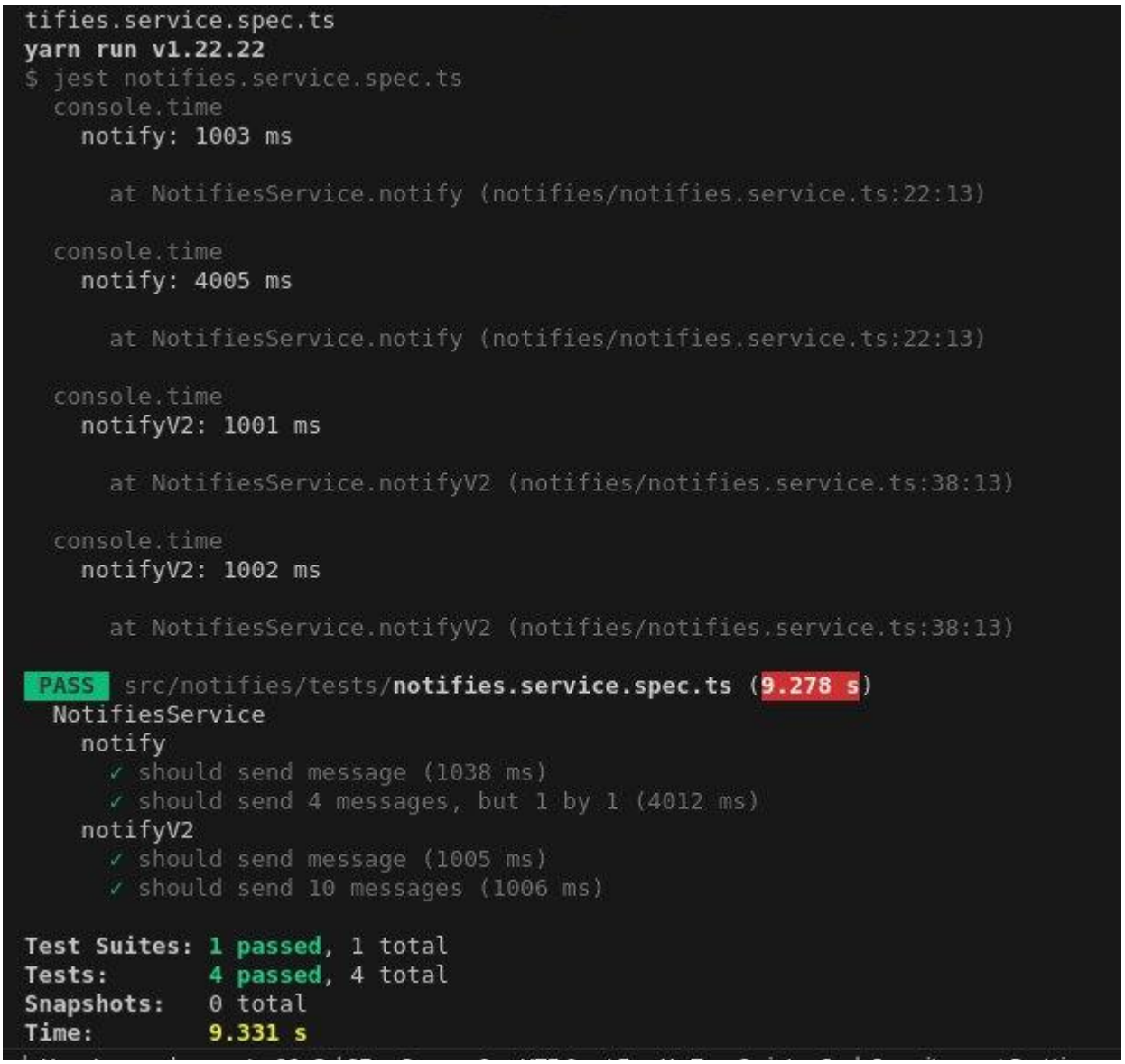

Рисунок 4.5 – Приклади виконаних unit-тестів с часом виконання

5 ТЕСТУВАННЯ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ

Тестування взаємодії мікросервісів є ключовим аспектом забезпечення надійності та узгодженості всієї системи. Мікросервісна архітектура, яка базується на незалежних мікросервісах, що взаємодіють між собою через різні протоколи (наприклад, HTTP, WebSocket, повідомлення через черги), вимагає особливої уваги до тестування взаємодій. Тестування взаємодії між сервісами дозволяє виявити проблеми, які можуть виникати на рівні комунікацій і спільного функціонування, а також забезпечує коректну передачу даних між компонентами системи.

5.1 Особливості тестування взаємодії мікросервісів

Тестування взаємодії мікросервісів відрізняється від класичного тестування монолітних систем. Основні виклики полягають у:

– розподіленій природі системи. Мікросервіси можуть бути розміщені на різних серверах або навіть у різних середовищах;

– залежності від мережі. Мікросервіси зазвичай комунікують через мережу, що додає фактори надійності, швидкості та затримок, які потрібно враховувати при тестуванні;

– незалежність сервісів. Кожен сервіс повинен бути максимально незалежним, і його тестування має враховувати це, хоча воно й має тестуватися у зв'язці з іншими сервісами;

– різні типи взаємодії. Взаємодія може відбуватися через HTTP-запити, WebSocket, повідомлення у чергах або інших протоколах, що потребує специфічних підходів до тестування кожного з цих видів комунікації.

5.2 Стратегії тестування взаємодії мікросервісів

Щоб забезпечити повноцінне тестування мікросервісної системи, зазвичай використовуються кілька типів тестів: контрактне тестування, тестування на рівні API, інтеграційне тестування, тестування через симуляцію, тестування продуктивності, тестування відновлення після збоїв.

5.2.1 Контрактне тестування (Contract Testing)

Це тип тестування, який забезпечує, що мікросервіси виконують свої зобов'язання перед іншими сервісами згідно з визначеними контрактами API.

Контрактне тестування забезпечує, що при будь-яких змінах у структурі даних або API одного з мікросервісів ці зміни не порушують роботу інших сервісів, що залежні від нього.

Один з найпоширеніших інструментів для контрактного тестування – це Pact, який дозволяє тестувати інтеграції між мікросервісами з обох сторін: як споживача API, так і постачальника.

5.2.2 Тестування на рівні API

Це найбільш поширений тип тестування взаємодії між сервісами. Тестування проводиться шляхом виконання HTTP-запитів до API одного мікросервісу та перевірки, чи правильно інший сервіс обробляє отримані дані.

Для цього використовуються інструменти, такі як Postman, JUnit (з бібліотеками для HTTP-запитів), або Supertest для Node.js.

5.2.3 Інтеграційне тестування (Integration Testing)

Інтеграційне тестування перевіряє взаємодію декількох мікросервісів у реальному середовищі.

У такому тестуванні вся система, або велика її частина, розгортається у тестовому середовищі, і тести виконуються на реальних даних та з реальними запитами.

Наприклад, тест може перевіряти, як мікросервіс для керування зустрічами взаємодіє з мікросервісом сповіщень через WebSocket та електронну пошту.

5.2.4 Тестування через симуляцію (Mock Testing)

У цьому підході взаємодії між мікросервісами тестуються шляхом створення макетів (mock) для зовнішніх сервісів.

Це дозволяє ізолювати тестування конкретного сервісу, при цьому перевіряючи, чи правильно він взаємодіє з іншими сервісами.

Наприклад, мікросервіс сповіщень може бути замінений на mock-версію, щоб перевірити, як мікросервіс для керування зустрічами обробляє відправку даних про нові зустрічі.

5.2.5 Тестування продуктивності (Performance Testing)

Це тестування допомагає визначити, як система реагує на навантаження і затримки між мікросервісами.

Для мікросервісної архітектури це особливо важливо, оскільки розподілені системи можуть мати вузькі місця у взаємодії між сервісами.

Інструменти, такі як JMeter або Artillery, можуть бути використані для навантажувального тестування, яке перевіряє стійкість мікросервісів під великим трафіком.

5.2.6 Тестування відновлення після збоїв (Fault Tolerance Testing)

Важливим аспектом тестування взаємодії мікросервісів є перевірка того, як система поводиться при збої одного або кількох сервісів.

Це тестування допомагає перевірити стратегії відновлення після збоїв, повторної спроби запитів, а також як система забезпечує узгодженість даних після збою.

5.3 Основні аспекти тестування взаємодії між конкретними мікросервісами

5.3.1 Тестування взаємодії мікросервісу керування зустрічами та мікросервісу сповіщень

Тестування POST-запитів на створення, редагування, видалення зустрічі.

Тестування повинно перевіряти, чи коректно мікросервіс керування зустрічами обробляє запити на створення або редагування зустрічі і чи відправляє відповідні дані до мікросервісу сповіщень.

Тестування відправки електронних листів.

Тестування повинно перевіряти, чи правильно мікросервіс сповіщень генерує та надсилає електронні листи після створення або зміни зустрічі в мікросервісі керування зустрічами.

5.3.2 Забезпечення узгодженості даних між мікросервісами

Мікросервіси повинні забезпечувати узгодженість даних навіть у випадках збоїв або нестабільних мережевих з'єднань. Тестування узгодженості передбачає перевірку того, що дані, які передаються між мікросервісами, залишаються коректними і актуальними на всіх рівнях взаємодії.

6. РЕЗУЛЬТАТИ ТЕСТУВАННЯ УЗГОДЖЕНОСТІ ДАНИХ МІЖ МІКРОСЕРВІСАМИ

Тестування узгодженості даних між мікросервісами є ключовим етапом для підтвердження того, що дані, які передаються між окремими сервісами, залишаються актуальними та коректними незалежно від типу взаємодії або сценарію використання. Під час тестування системи, що складається з двох мікросервісів — сервісу для керування зустрічами та сервісу сповіщень — було проведено кілька важливих перевірок, спрямованих на забезпечення надійності та узгодженості даних.

6.1 Мета тестування

Основною метою тестування було переконатися, що мікросервіси:

– коректно передають дані один одному незалежно від типу запиту (створення, редагування, видалення зустрічей);

– забезпечують узгодженість даних між усіма користувачами системи, як через HTTP-запити, так і через WebSocket;

– надсилають повідомлення про зміни зустрічей в реальному часі через WebSocket та електронну пошту;

– не допускають втрати або дублювання даних при комунікації між мікросервісами, навіть у випадку збоїв або затримок у передачі.

6.2 Використані підходи для тестування

Під час тестування були використані різні підходи та інструменти для перевірки взаємодії та узгодженості даних між мікросервісами:

– тестування API. Тестування взаємодії мікросервісів через HTTP-запити з використанням Jest для перевірки правильності обробки даних між

сервісами. Тести охоплювали створення, редагування та видалення зустрічей. Було перевірено, чи отримує сервіс сповіщень дані при кожному POST-запиті, а також чи коректно він відправляє інформацію через WebSocket і email;

– інтеграційне тестування. Тести проводилися в середовищі, що імітує роботу системи в реальних умовах, щоб перевірити, як мікросервіси взаємодіють один з одним під навантаженням і в нормальних умовах. Для цього були розгорнуті обидва мікросервіси, і проводилися тести на перевірку їхньої взаємодії в реальному часі;

– тестування через WebSocket. Перевірялася здатність сервісу сповіщень отримувати дані через POST-запити і негайно передавати зміни підключеним клієнтам через WebSocket. Була симульована робота кількох клієнтів, які отримують повідомлення про зміни у зустрічах в реальному часі;

– тестування відправки електронних листів. Використовувалося SMTP-серверне тестування для перевірки, чи коректно відправляються електронні листи після створення або редагування зустрічі. Тести перевіряли зміст повідомлення, коректність адреси та своєчасність доставки.

6.3 Результати тестування

6.3.1 Узгодженість даних при створенні та редагуванні зустрічей

При створенні нової зустрічі через POST-запит мікросервіс для керування зустрічами правильно передавав всі дані до сервісу сповіщень. Усі підключені клієнти, що використовували WebSocket, отримували актуальну інформацію в реальному часі. Водночас надсилалася електронна пошта з деталями зустрічі на вказані адреси.

Усі тести підтвердили, що зміни у зустрічах негайно відображалися на стороні клієнта та у сповіщеннях, і дані залишалися узгодженими між сервісами та клієнтами.

6.3.2 Видалення зустрічей

Тести на видалення зустрічей підтвердили, що дані про видалену зустріч коректно передаються на сервіс сповіщень і відповідно надсилаються через WebSocket і email. Усі підключені клієнти отримували сповіщення про видалення, і після цього ці зустрічі не були доступні через GET-запити до API.

6.3.3 Тестування взаємодії через WebSocket

Мікросервіс сповіщень коректно обробляв повідомлення через WebSocket. Усі адміністратори, підключені до WebSocket, отримували повідомлення про створення, редагування або видалення зустрічей без затримок.

Тести підтвердили, що WebSocket-з'єднання стабільне навіть при великій кількості підключень, і дані передаються без втрат.

6.3.4 Надсилання електронних листів

Під час тестування надсилання електронних листів на вказані адреси було підтверджено, що повідомлення формуються та відправляються правильно. Перевірялися різні сценарії — відправка листів при створенні нової зустрічі, зміні її даних або видаленні. Усі листи були доставлені своєчасно, і їхній зміст відповідав даним із системи.

6.3.5 Тестування продуктивності

Тести продуктивності показали, що система здатна обробляти великий обсяг запитів і підключень без значних затримок або втрати даних.

Навантажувальні тести підтвердили, що система залишається стабільною при високих навантаженнях, а узгодженість даних зберігається.

ВИСНОВКИ

Кваліфікаційна робота присвячена дослідженню взаємодії та узгодженості даних у мікросервісах на основі сучасних технологій розробки програмного забезпечення. У роботі було розглянуто всі ключові аспекти мікросервісної архітектури, тестування, а також реалізацію і функціонування двох бекенд мікросервісів на Node.js із використанням фреймворку NestJS.

У ході дослідження було висвітлено наступні теми:

1. Сучасні технології мікросервісної архітектури. Було детально розглянуто принципи мікросервісної архітектури, її переваги у порівнянні з монолітними системами, а також її здатність підвищувати масштабованість, гнучкість і надійність систем за рахунок модульної структури та незалежного розгортання сервісів. У роботі підкреслено важливість самостійності мікросервісів, що дозволяє окремим компонентам функціонувати та еволюціонувати незалежно один від одного.

2. Тестування як основний інструмент забезпечення якості. Показано, що тестування є важливим етапом у життєвому циклі розробки ПЗ, що забезпечує виявлення помилок, перевірку функціональності та надійності системи. Були розглянуті основні принципи тестування, його методи та роль у забезпеченні якості мікросервісних систем. Зроблено акцент на таких техніках, як контрактне тестування, API-тестування, інтеграційне та навантажувальне тестування.

3. Опис функціоналу та архітектури мікросервісів. Було представлено архітектуру двох бекенд мікросервісів, де один з них обробляє HTTP-запити для керування зустрічами (створення, редагування, видалення та отримання зустрічей), а другий приймає ці дані та надсилає сповіщення через WebSocket і електронну пошту. Описано деталі взаємодії між сервісами, які забезпечують узгодженість даних та інформування користувачів в режимі реального часу.

4. Опис реалізації мікросервісів. Було розроблено та описано реалізацію двох мікросервісів на платформі Node.js з використанням фреймворку NestJS. Наведено переваги використання NestJS для мікросервісної архітектури, зокрема його модульність, потужну систему обробки запитів, підтримку WebSocket і інтеграцію з email-сервісами. Також було показано, як ці мікросервіси взаємодіють між собою через HTTP та WebSocket, забезпечуючи користувачам швидкий доступ до актуальної інформації про зустрічі.

5. Тестування взаємодії мікросервісів. Було детально описано процес тестування взаємодії між мікросервісами. Використано такі підходи, як контрактне тестування для перевірки відповідності API сервісів, інтеграційне тестування для перевірки реальної взаємодії в тестовому середовищі, а також навантажувальне тестування для оцінки продуктивності системи під трафіком. Окремо було розглянуто тестування WebSocket і email-сервісів, що забезпечує оповіщення користувачів про зміни.

У підсумку, кваліфікаційна робота показала важливість тестування як основного елементу забезпечення якості в мікросервісній архітектурі. Реалізація двох мікросервісів та їх тестування дозволяють продемонструвати, як за допомогою сучасних інструментів можна створити ефективну систему для керування даними та сповіщення користувачів, забезпечуючи при цьому узгодженість і надійність взаємодії між компонентами системи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Офіційна документація NestJS.NestJS - A progressive Node.js framework. Офіційна документація.https://docs.nestjs.com
2. Офіційна документація Node.js. Node.js Documentation. Офіційний сайт. https://nodejs.org/en/docs
3. Мартін Фаулер - Мікросервіси: Погляд на програмну архітектуру Fowler, M. "Microservices: A Definition of This New Architectural Term." https://martinfowler.com/articles/microservices.html
4. YouTube-канал Traversy Media - NestJS Crash Course. Traversy Media. "NestJS Crash Course". Відео на YouTube. https://www.youtube.com/watch?v=wqhNoDE6pb4
5. YouTube-канал Academind - Microservices Crash Course Academind. "Microservices Crash Course – Understanding the Basics". https://www.youtube.com/watch?v=vvhC64hQZMk
6. Офіційна документація Postman. Postman Learning Center - API Development Tools. https://learning.postman.com
7. Тестування мікросервісів з використанням Jest та Supertest. Article: "Microservice Testing Using Jest & Supertest" – Blog by LogRocket.. https://blog.logrocket.com/testing-nodejs-express-jest-supertest
8. Документація по WebSocket. MDN Web Docs. "Introduction to WebSockets". https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
9. Архітектура мікросервісів - стаття на Medium. Bozhekulov, A. "Microservices Architecture: Key Concepts & Benefits." Medium. https://medium.com/swlh/microservices-architecture-key-concepts-benefits-8f56f823befe

10. YouTube-канал Fireship - Microservices Explained. Fireship. "Microservices Explained in 100 Seconds". https://www.youtube.com/watch?v=jZ4gTmfo4lA

11. Тестування REST API з використанням Postman. Medium article: "How to Test REST API with Postman". https://medium.com/@paulrajudesigner/how-to-test-rest-api-with-postman-57f8d7eac419

12. Офіційна документація для SMTP-серверів (email). Nodemailer - Send Emails with Node.js. https://nodemailer.com

13. Документація Docker. Docker - Build, Share, and Run Any App, Anywhere. https://docs.docker.com

14. YouTube-канал TechWorld with Nana - Docker Tutorial for Beginners. TechWorld with Nana. "Docker Tutorial for Beginners – A Full DevOps Course". https://www.youtube.com/watch?v=3c-iBn73dDE