Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*
(повна назва)
Кафедра *Системотехніки*
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Дослідження та аналіз процесів взаємодії сервісів в мікро-сервісній архітектурі
(тема)

Виконав:
Студент 2 курсу, групи *СПРм-22-2*
Спеціальність *122 – Комп'ютерні науки*
(код і повна назва напряму)
Тип програми *освітньо-наукова*
(освітньо-професійна або освітньо-наукова)
Освітня програма *Системне проєктування*
(повна назва освітньої програми)

Белименко В.С.
(прізвище, ініціали)

Керівник *к.т.н., проф. Іванов В. Г.*
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ *Гребеннік І. В.*
(підпис) (прізвище, ініціали)

2024 р

Атестаційна робота не містить відомостей заборонених до відкритого опублікування.

Атестаційна робота виконана у відповідності до стандартів, що діють в Україні.

Попередній захист проведений «10» червня 2024 р.

Керівник атестаційної роботи к.т.н., проф. Іванов В.Г.

Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет *Комп'ютерних наук*
(повна назва)

Кафедра *Системотехніки*
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність *122 – Комп'ютерні науки*
(код і повна назва)

Тип програми *освітньо-наукова*
(освітньо-професійна або освітньо-наукова)

Освітня програма *Системне проєктування*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Белименку Вячеславу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Дослідження та аналіз процесів взаємодії сервісів в мікро-сервісній архітектурі*
затверджена наказом по університету від «01» квітня 2024р.№ 259 Ст

2. Термін подання студентом роботи (проекту) 13 червня 2024 р.

3. Вихідні дані до роботи (проекту) інформація про різні архітектури програмного забезпечення, зокрема теоретичні дані про їх типи, такі як монолітна та мікросервісна архітектура, разом з аналізом їх переваг і недоліків. Дані про мікросервісну архітектуру, зокрема її принципи, а також інформація про її застосування, переваги в контексті масштабованості та гнучкості. Інформація про популярні формати серіалізації даних і їх вплив на продуктивність системи. Для обраної програмної системи будуть враховуватися технічні вимоги і специфікації для її розробки, а також вибір і опис технологій, що будуть використані для реалізації. Методологічні дані для тестування включатимуть вимоги до методів тестування продуктивності і надійності, а також параметри для оцінки ефективності міжсервісного зв'язку. Для моделювання продуктивності будуть використовувані вихідні дані, що включають різні сценарії використання та фактори, що впливають на продуктивність.

4. Зміст пояснювальної записки (перелік питань, що потрібно розробити) дослідити і проаналізувати процеси взаємодії сервісів у мікросервісній архітектурі. Провести аналіз різних архітектур

програмного забезпечення, зокрема їх переваг і недоліків, а також розглянути основні принципи мікросервісної архітектури, що забезпечують масштабованість і гнучкість. Зробити оцінку різних технологій міжсервісного зв'язку з огляду на їх переваги та обмеження. Спроектувати програмну систему для тестування міжсервісного зв'язку, включаючи основні компоненти і модулі, а також детально описати обрані технології. Розробити методи тестування, організовати процес тестування продуктивності і надійності системи. Провести оцінки ефективності міжсервісного зв'язку, що включає різні параметри, такі як час відповіді і пропускна здатність. Провести моделювання продуктивності технологій з урахуванням різних факторів, а також сформульовати рекомендації щодо вибору технологій міжсервісного зв'язку, що можуть бути корисними для практичного використання мікросервісної архітектури.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів): Життєві цикли проєктів, Типи архітектури, Структура проєкту, Діаграми класів, Схема роботи месседж брокера, Форми розроблених додатків, Результати теснування.,

6. Консультанти розділів роботи (проекту)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спец. частина	к.т.н., проф. Іванов В. Г.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Отримання завдання на атестаційне проектування	01.04.2024	
2.	Аналіз завдання та пошук літератури з теми роботи	05.04.2024	
3.	Опрацювання літератури та аналіз об'єкту дослідження	12.04.2024	
4.	Вибір апаратного набору та його збірка	26.04.2024	
5.	Розробка алгоритмів	17.05.2024	
6.	Аналіз отриманих результатів	31.05.2024	
7.	Оформлення пояснювальної записки та документації	06.06.2024	
8.	Оформлення презентаційних матеріалів	07.06.2024	
9.	Представлення на рецензування	10.06.2024	
10.	Представлення атестаційної роботи	13.06.2024	

Дата видачі завдання *01 квітня 2024 р.*

Студент _____ *Белименко В.С*

(підпис)

Керівник роботи _____ к.т.н., проф. Іванов В. Г.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 79 с., 8 табл., 39 рис., 4 додаток, 39 джерел інформації

REST, KAFKA, GRAPHQL, C #, ASP, MVC, SOAP, JSON AMQP

Метою цієї роботи є визначення найефективніших засобів і технологій для міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

Об'єктом дослідження є міжсервісні зв'язки в мікросервісній архітектурі програмних систем.

Предметом дослідження є засоби і технології міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

У даній роботі для дослідження ефективності використаних програмних засобів і технологій застосовувалися емпіричні методи програмної інженерії, загальнологічні методи наукового пізнання, а також порівняльний метод.

Наукова новизна полягає в запропонованій методиці оцінки ефективності технологій міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

Практичне значення дослідження полягає в тому, що результати аналізу продуктивності та рекомендації щодо вибору засобів міжсервісної комунікації дозволять найбільш ефективно організувати міжсервісний зв'язок, що сприятиме підвищенню загальної продуктивності системи.

ABSTRACT

Explanatory note to the qualification work: 79p., 8 table., 39 fig., 4 appendices, 39 sources of information

REST, KAFKA, GRAPHQL, C #, ASP, MVC, SOAP, JSON AMQP

The aim of this work is to determine the most effective means and technologies for inter-service communication in the microservices architecture of software systems.

The object of the research is inter-service communication in the microservice architecture of software systems.

The subject of the study is the means and technologies of inter-service communication in the microservices architecture of software systems. In this work, empirical methods of software engineering, general logical methods of scientific cognition, and a comparative method were applied to investigate the effectiveness of the used software tools and technologies.

The scientific novelty lies in the proposed methodology for evaluating the effectiveness of inter-service communication technologies in the microservices architecture of software systems.

The practical significance of the research lies in the fact that the results of performance analysis and recommendations for choosing inter-service communication tools will allow for the most effective organization of inter-service communication, thereby contributing to the overall productivity of the system.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ 10
1.1 Архітектури програмного забезпечення. Переваги і недоліки 10
1.2 Мікросервісна архітектура програмних систем 21
1.3 Проблеми процесу міжсервісної комунікації 22
1.4 Постановка задачі 24
2 ЗАСОБИ І ТЕХНОЛОГІЇ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ 26
2.1 Аналіз критеріїв вибору технології реалізації міжсервісного зв’язку 26
2.2 Формат серіалізації даних 28
2.3 Дослідження засобів і технологій міжсервісного зв’язку 29
3 ОПИС ПРОГРАМНОЇ СИСТЕМИ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ТЕХНОЛОГІЙ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ 32
3.1 Дизайн програмної системи 32
3.2 Опис обраних технологій 36
3.3 Програмна реалізація 48
3.4 Тестування програмної системи 63
4 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ З ВИКОРИСТАННЯМ ОБРАНИХ ТЕХНОЛОГІЙ 72
4.1 Методика оцінювання ефективності 72
4.2 Моделювання продуктивності технологій міжсервісного зв’язку 74
4.3 Рекомендації по використанню технологій міжсервісного зв’язку 78
ВИСНОВКИ 80
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 8
ДОДАТОК А Графічний матеріал кваліфікаційної роботи 84
ДОДАТОК Б Текст програми 94

ВСТУП

На сьогоднішній день IT індустрія розвивається досить швидко та активно. Все більше та більше зароджується нових проектів, стартапів, т.д.

На початку зародження IT індустрії, мало хто розглядав її як бізнес, більшість розробок було створено лише за натхненням та ентузіазмом розробників. Потім виникла потреба в бізнес додатках та у військовій галузі. Якщо говорити про бізнес додатки для внутрішніх потреб, то із-за нестачі досвіду замовників у процесі розробки, вони безпосередньо вели перемовини з розробниками. Виконавці здебільшого не оцінювали свій труд належним чином, і могли або ж переоцінити свій труд та виставити ціну вище реальної, або ж, навпаки - занизити.

Щоб якось конкретизувати та формалізувати бажання замовників складали технічне завдання (далі ТЗ). Та з ним виникало досить багато проблем, оскільки на той час існувало досить мало готових рішень та й взагалі процес розробки був досить тяжкий, оцінювати свої можливості було важко. Здебільшого домовлялися за етапи ТЗ. Наприклад, розбивали продукт на якісь логічні частини котрі оцінювали у певні кошти і встановлювали умовні терміни виконання. Це приносило багато незручностей. Замовники не бачили результат, оскільки важко пояснити що було написана досить велика доля серверної частини, а замовник бачить лише одну графічну форму для автентифікації користувача. Та й розробникам було важко оскільки вони отримували гроші лише по закінченню певного логічного етапу і здебільшого вони хибно оцінювали свої можливості, тобто не вкладалися у визначені терміни. Тому виникало досить багато суперечок, конфліктів тощо. Та й замовники здебільшого не могли грамотно сформулювати свої думки та просто на просто не могли пояснити розробнику свої вимоги.

Звичайно ж ця сфера розвивалась увесь час і почали з’являтися менеджери проектів, бізнес аналітики та інші. Ці люди слугують таким нібито мостом між розробниками та замовниками. Вони досить непогано орієнтуються, як

супроводжувати процес розробки та можуть краще зрозуміти замовників й формалізувати їх вимоги ніж розробник. Перейшли до погодинної оплати роботи, що давало свої успіхи та має досить багато переваг, але недоліки так і залишилися.

Метою цієї роботи є визначення найефективніших засобів і технологій для міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

Об'єктом дослідження є процес міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

Предметом дослідження є засоби і технології міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

У даній роботі для дослідження ефективності використаних програмних засобів і технологій застосовувалися емпіричні методи програмної інженерії, загальнологічні методи наукового пізнання, а також порівняльний метод.

Наукова новизна полягає в запропонованій методиці оцінки ефективності технологій міжсервісного зв'язку в мікросервісній архітектурі програмних систем.

Практичне значення дослідження полягає в тому, що результати аналізу продуктивності та рекомендації щодо вибору засобів міжсервісної комунікації дозволять найбільш ефективно організувати міжсервісний зв'язок, що сприятиме підвищенню загальної продуктивності системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Архітектури програмного забезпечення. Переваги і недоліки

У реальних умовах проектування інформаційних систем йдеться про пошук способів, які забезпечать необхідну функціональність системи за допомогою існуючих технологій, враховуючи встановлені обмеження. Методологія розробки включає набір методів і критеріїв оцінки, які використовуються для постановки завдань, планування, контролю та досягнення поставленої мети. Процес розробки описується моделлю, яка визначає послідовність загальних етапів і очікуваних результатів. Сьогодні існує багато методологій управління проектами та відповідного програмного забезпечення, кожна з яких має свої переваги та недоліки. Розглянемо сім основних методологій, які довели свою ефективність при розробці програмного забезпечення.

Каскадна модель є однією з найтрадиційніших та найпоширеніших методологій для розробки програмного забезпечення.

Ця модель життєвого циклу, як показано на рисунку 1.1, вважається класичним стилем програмної розробки. Вона представляє процес розробки як лінійну послідовність етапів, де кожен етап починається лише після завершення попереднього. Цей підхід не передбачає можливості повернення до попередньої фази для внесення змін у вимоги. Цю методологію використовують, коли вимоги вже чітко визначені, немає проблем із кваліфікованими фахівцями, і проект є відносно невеликим.

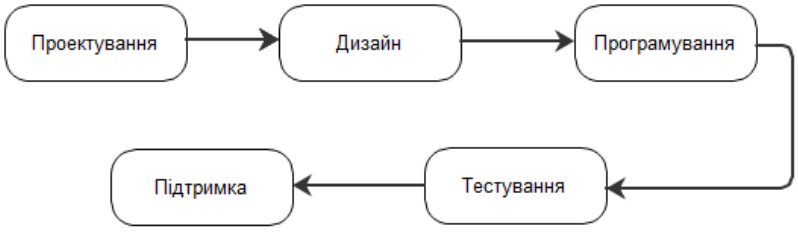

Рисунок 1.1 – Життєвий цикл каскадної моделі

Каскадна модель розробки програмного забезпечення має свої переваги і недоліки. Серед основних переваг можна виділити її простоту і легкість у розумінні та використанні, що особливо важливо для новачків-розробників. Модель також легко керована завдяки своїй жорсткій структурі, оскільки кожна фаза має чітко визначений очікуваний результат і окремий процес перевірки. Крім того, фази в цій моделі виконуються та завершуються лише один раз, що значно економить час. Цей підхід є більш ефективним для невеликих проєктів з добре сформульованими вимогами, а тестування є відносно простим, оскільки воно ґрунтується на сценаріях, визначених у попередніх функціональних специфікаціях.

Однак, каскадна модель має і свої недоліки. Вона може бути використана тільки тоді, коли попередні вимоги чітко визначені, і не підходить для підтримки проєктів. Головним недоліком є те, що під час тестування немає можливості повернутися до попередніх етапів для внесення змін. Також відсутня можливість демонстрації працюючого додатка до завершення останньої стадії циклу, що унеможливлює дізнатися кінцевий результат всього проєкту на ранніх етапах. Ця модель добре підходить для невеликих проєктів, але не є ідеальною для довготривалих та безперервних проєктів, а також для проєктів з недостатньо визначеними вимогами, які потребують частих змін.

У інкрементальній моделі повні вимоги до системи розбиваються на різні збірки. Ця термінологія часто використовується для опису поетапної збірки програмного забезпечення. Ця модель передбачає кілька циклів розробки, які разом утворюють життєвий цикл "мульти-водопад", що зображений на рисунку 1.2. Кожен цикл розділений на менші модулі, які легко створюються. Кожен модуль проходить через фази визначення вимог, проектування, кодування, впровадження та тестування. Згідно з інкрементальною моделлю, процедура розробки передбачає випуск базової функціональності на першому великому етапі продукту, а потім послідовне додавання нових функцій, або "інкрементів". Цей процес триває до тих пір, поки не буде створена повна система.

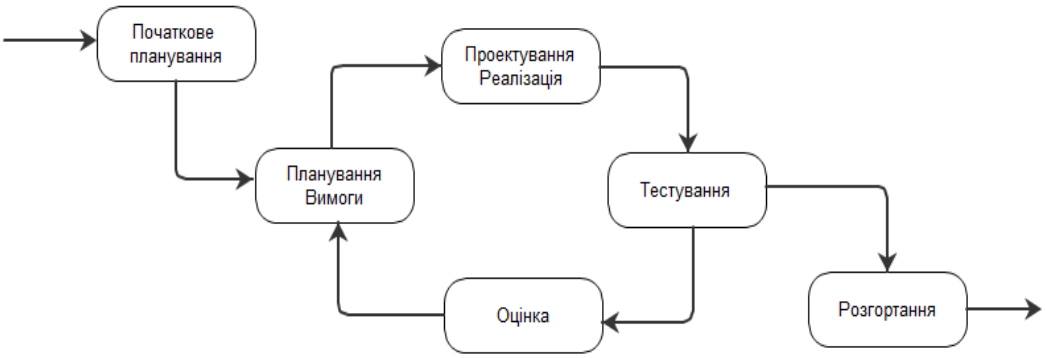

Рисунок 1.2 – Життєвий цикл інкрементальної моделі

Цю модель використовують коли основні вимоги до системи визначені та зрозумілі. У той ж час деякі деталі можуть дороблятися протягом процесу розробки ПЗ. Надає можливість досить швидко випустити продукт на ринок.

Переваги:

– швидко впроваджує нові версії ПЗ;

– надає гнучкість та невеликі затрати при зміні вимог;

– завдяки невеликим ітераціям, тестування та відлагуджування досить легко виконувати;

– досить легко виявляти та зменшувати ризики.

Недоліки:

– потребує кропітливого планування та проектування;

– потребує повного визначення цілої системи;

– більш затратна ніж водопадна модель.

RAD-модель є одним з варіантів інкрементальної моделі. У цій моделі компоненти або функції розробляються кількома висококваліфікованими командами паралельно, ніби це декілька міні-проектів. Часові обмеження одного циклу встановлені жорстко. Після цього створені модулі інтегруються в один робочий прототип. Спільна робота команд дозволяє дуже швидко надати клієнту робочий продукт для огляду, з метою отримання зворотного зв'язку та внесення змін.

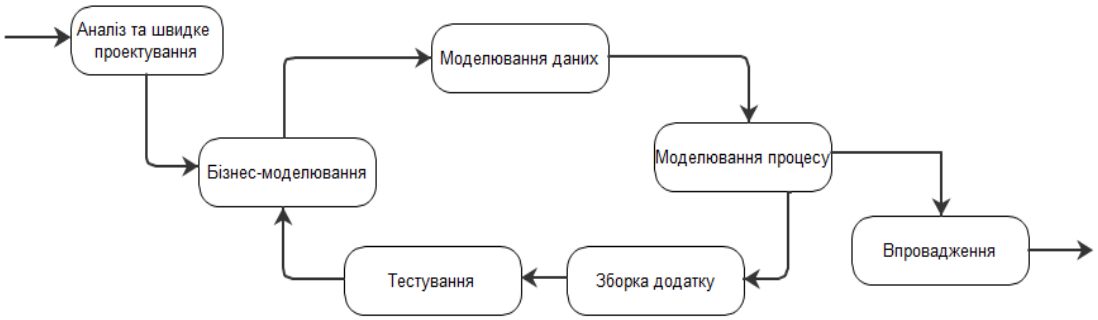

Рисунок 1.3 – Життєвий цикл RAD-моделі

Модель швидкої розробки додатків включає наступні етапи (див. рисунок 1.3):

– бізнес-моделювання: визначає перелік інформаційних потоків між різними підрозділами;

– моделювання даних: використовує інформацію, зібрану на попередньому етапі, для визначення об'єктів і сутностей, необхідних для обміну інформацією;

– моделювання процесу: з'єднує інформаційні потоки для досягнення цілей розробки;

– збірка програми: використовується автоматичне складання для перетворення моделей системи на код;

– тестування: тестуються нові компоненти і інтерфейси.

Переваги:

– швидка модель розробки допомагає знизити ризик та зусилля розробника програмного забезпечення;

– ця модель сприяє частим перевіркам проекту з боку клієнта;

– методика забезпечує зворотний зв'язок з клієнтами, що сприяє поліпшенню будь-якого проекту розробки ПЗ.

Недоліки:

– модель передбачає сильну команду та індивідуальні зустрічі для чіткого визначення вимог бізнесу;

– підходить лише для додатків, які можуть бути розбиті на модулі;

– вимагає висококваліфікованих розробників і команди дизайнерів, що може бути складним для деяких організацій.

Гнучка методологія використовується для проектування системи управління розробкою, що дозволяє постійно вносити зміни в процес розробки [4]. Ця методологія є однією з основних концепцій при створенні різних проектів у галузі програмного забезпечення. Модель застосовується для мінімізації ризику під час розробки продукту у короткі терміни, які називаються ітераціями і зазвичай тривають від одного тижня до одного місяця. Життєвий цикл методології зображено на рисунку 1.4.

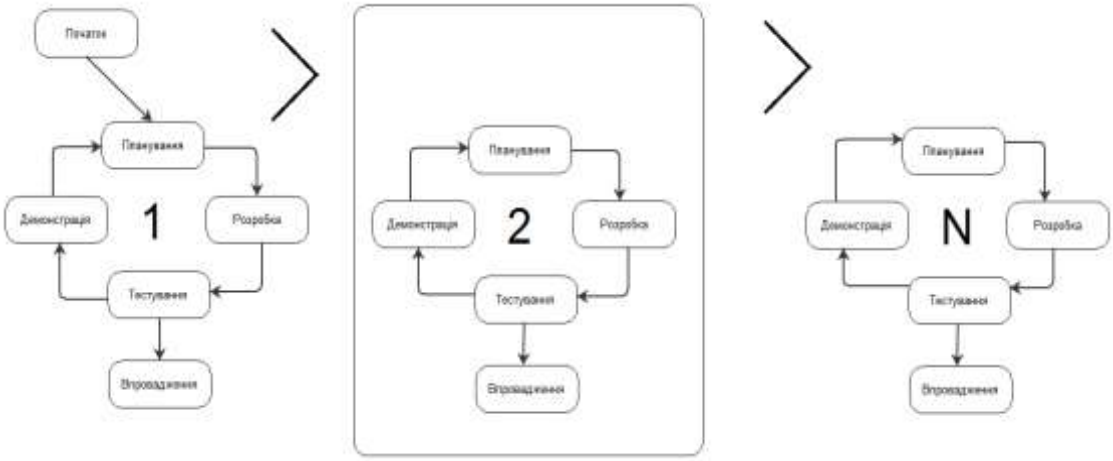

Рисунок 1.4 – Життєвий цикл гнучкої методології

Гнучку методологію слід застосовувати в умовах динамічного бізнесу, де потреби користувачів постійно змінюються. Зміни в гнучкій методології реалізуються за меншу ціну завдяки постійним спринтам. Для початку проекту достатньо невеликого планування, на відміну від каскадної моделі.

Переваги:

– гнучка методологія має адаптивний підхід, що дозволяє змінювати вимоги клієнтів;

– прямий зв'язок та постійний зворотний зв'язок замовників або їх представників усуває невизначеності.

Недоліки:

– ця методологія зосереджена на створенні програмного забезпечення раніше, ніж на документації, що може призвести до нестачі документації;

– процес розробки може вийти з-під контролю, якщо замовник не чітко визначає кінцевий результат проекту.

Спіральна модель є досить складною та концептуальною, спрямованою на раннє виявлення та зменшення ризиків проекту. У цій методології розробники розпочинають з невеликих масштабів, виявляють можливі ризики, складають план їх запобігання, а потім вирішують, чи варто переходити до наступної стадії проекту для здійснення наступної ітерації спіралі. Успіх будь-якого життєвого циклу спіральної моделі залежить від надійного, уважного та грамотного керівництва проекту.

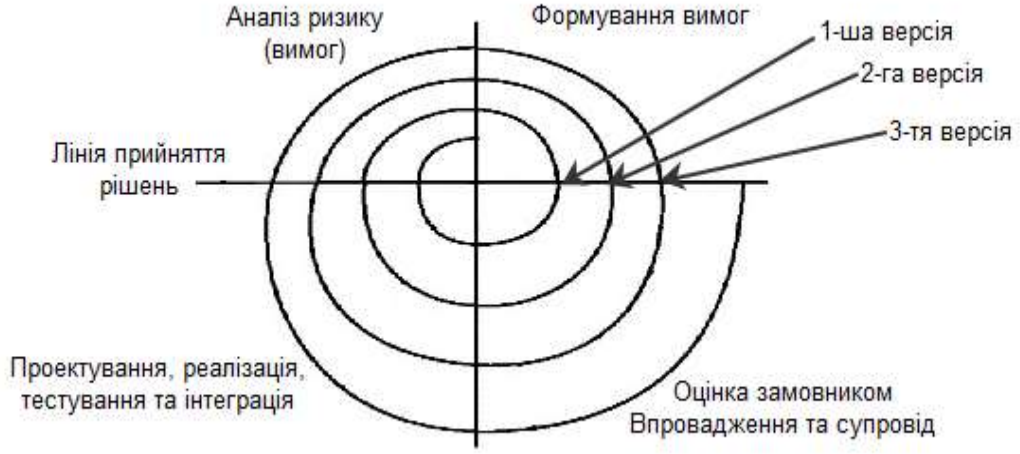

Рисунок 1.5 – Життєвий цикл спіральної методології

Ця модель не підходить для малих проектів, але вона ідеально підходить для складних та витратних проектів, наприклад, розробки системи документообігу для банку, де кожен крок потребує глибокого аналізу перед програмуванням.

Переваги:

– об'ємний аналіз ризиків зазвичай зменшує ймовірність провалу;

– ця модель добре підходить для великих та ризикованих проектів;

– додаткову функціональність можна додати пізніше у спіральній моделі;

– вона задовольняє потреби високоризикованих проектів, де бажання та потреби замовника можуть змінюватися з часом.

Недоліки:

– ця модель дуже витратна з точки зору розробки;

– успіх проекту значно залежить від фази аналізу ризиків; невдача на цьому етапі може призвести до провалу проекту;

– вона не підходить для проектів з низьким ризиком;

– великий ризик полягає в тому, що ця методологія може тривати безкінечно і ніколи не завершитися.

Раціональний уніфікований процес (RUP) є однією з ітераційних методологій розробки програмного забезпечення [6]. Для моделювання використовується мова Unified Modeling Language (UML).

Ітераційна розробка в рамках RUP передбачає розбиття проекту на декілька менших проектів, які виконуються послідовно. Кожна ітерація має чітко визначений набір цілей, які мають бути досягнуті до кінця ітерації. Остання ітерація передбачає, що всі цілі повинні точно відповідати вимогам замовника, тобто всі вимоги повинні бути виконані.

RUP є досить формалізованою методологією, яка приділяє особливу увагу початковим стадіям розробки – аналізу та моделюванню. Це спрямовано на зниження комерційних ризиків шляхом виявлення помилок на ранніх стадіях розробки. Технічні ризики оцінюються та управляються на початкових етапах циклу розробки, а потім переглядаються з часом та з розвитком проекту протягом наступних ітерацій. Нові цілі з'являються залежно від пріоритетів цих ризиків. Релізи версій розподіляються таким чином, що найбільш пріоритетні ризики усуваються першими.

Процес передбачає еволюцію моделей, причому кожна ітерація циклу розробки відповідає певній версії моделі програмного забезпечення. Кожна ітерація включає елементи управління життєвим циклом програмного

забезпечення: аналіз і дизайн (моделювання), реалізація, інтеграція, тестування, впровадження. У цьому сенсі RUP є реалізацією спіральної моделі, хоча часто зображується у вигляді графіка-таблиці..

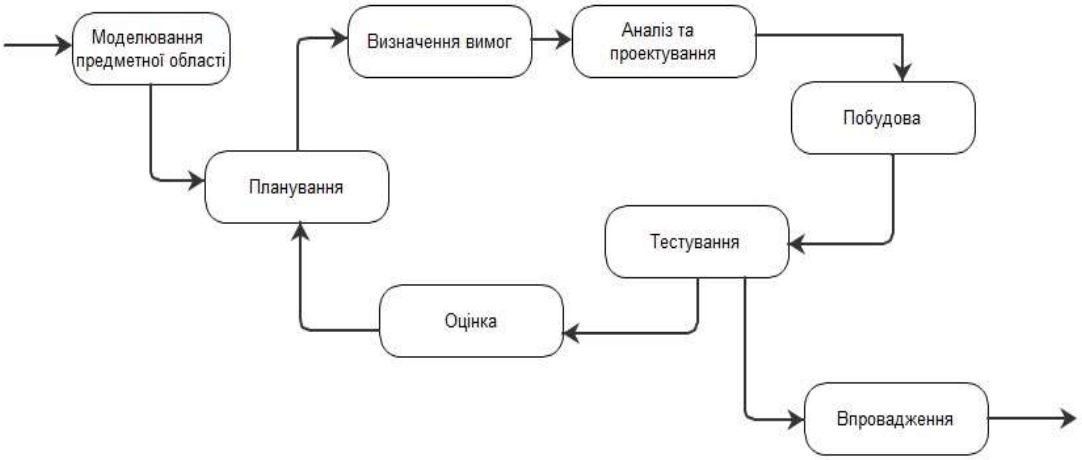

Рисунок 1.6 – Життєвий цикл RUP

В RUP рекомендовано дотримуватися шести практик, що дозволяють успішно розробляти проект:

- ітеративна розробка;
- управління вимогами;
- використання модульних архітектур;
- візуальне моделювання;
- перевірка якості;
- відстеження змін.

Переваги:

- ця методологія робить акцент на точній документації;
- допомагає у вирішенні ризиків, пов'язаних зі зростанням вимог замовників та запитів керівництва;
- зменшує потребу в інтеграції, оскільки процес інтеграції триває протягом всього процесу розробки.

Недоліки:

- розробники повинні бути експертами у своєму ділі в рамках цієї методології;

– процес розробки дуже комплексний і не завжди дуже точно організований;

– інтеграція під час усього процесу розробки може створювати проблеми, що можуть викликати більше несправностей під час тестування;

– цей процес дуже складний, через що може бути важким у розумінні.

Екстремальне програмування (Extreme Programming, XP) – це гнучка методологія розробки програмного забезпечення [7]. Вона використовується для створення додатків у дуже нестабільному середовищі, що додає великої гнучкості під час процесу моделювання. Головна перевага цієї методології – це малозатратність. Проте вартість змін вимог на пізній стадії проекту може бути досить високою. Життєвий цикл XP наведено на рисунку 1.7.

Рисунок 1.7 – Життєвий цикл XP

Переваги методології екстремального програмування:

– залученість замовника у процес розробки, що сприяє кращому розумінню його потреб і вимог;

– можливість визначення доцільних планів і графіків, а також можливість розробників самостійно зафіксувати їх, що є найбільшою перевагою;

– сумісність з більшістю сучасних методів розробки, що дозволяє розробникам створювати якісний продукт.

Недоліки методології екстремального програмування:

– ефективність цієї методології обмежена, якщо розробники не повністю зосереджені та захоплені розробкою;

– потреба у частих зустрічах, що може бути дорогим для замовників;

– велика кількість змін в розробці, що може бути трудомістким для розробників;

– складно визначити точні трудовитрати через невизначеність обсягу робіт та вимог на початку проекту.

Каскадна та Agile методології є найбільш популярними та одними з батьків усіх інших, тому що:

– каскадна методологія планує все до деталей, має жорсткі терміни та фіксований бюджет, але утруднює внесення змін під час розробки;

– Agile методологія набагато гнучкіша, дозволяє легко вносити зміни та пристосовуватися до змінних вимог, але складно встановлює терміни та бюджет.

Обидві методології мають свої переваги і недоліки, і вибір між ними залежить від конкретних вимог проекту та умов його реалізації.

На рисунку 1.8 зображена координатна площина, де розташовані всі розглянуті методології. Ось як вона працює: вісь абсцис відображає спектр від "низькоформалізованих" до "високоформалізованих" методологій, а вісь ординат представляє діапазон від "ітераційних" до "каскадних" методологій. За графіком видно, що дві методології, які стоять у крайніх точках, - це XP та каскадна. Це логічно випливає з розглянутого вище. XP спрямована на швидку розробку з невеликими ітераціями та легкістю внесення змін, тоді як каскадна вимагає чітко визначених умов та має фіксовані етапи, що ускладнює можливість змін.

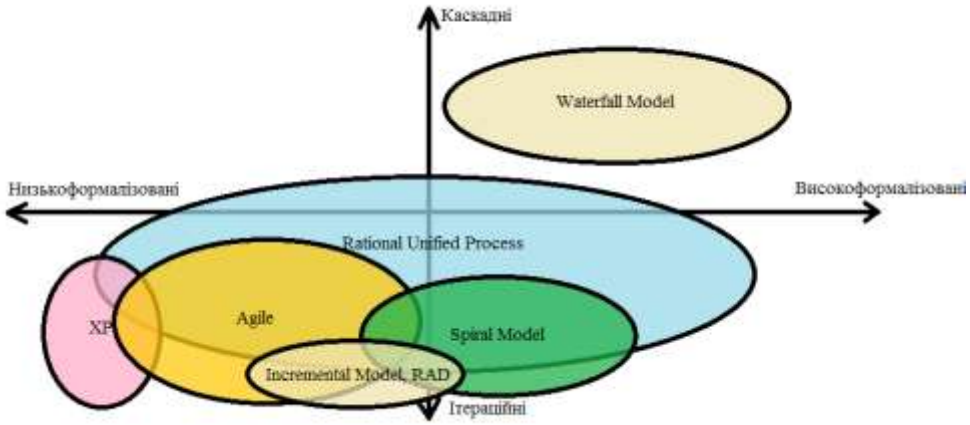

Рисунок 1.8 – Розташування методологій

У таблиці 1.1 наведено порівняння вищезгаданих методологій за певними критеріями. Як видно з таблиці, ітераційні методології володіють такими перевагами, як гарантія успіху та менший рівень ризику, але контроль витрат в них менший. З іншого боку, каскадні методології характеризуються досить високим контролем витрат, вищою детермінованістю вимог, але повністю або частково не надають можливості повернення до попередньої фази.

Таблиця 1.1

Порівняння методологій

	Каскадна	Інкрементальна	RAD	Agile	Спіральна	RUP	XP
Детермінованість вимог	повністю	основні	майже повн.	основні	бажано основні	майже повн.	початкові
Розмір проекту	нижче середн.	середній	малий	великий	великий	вище середн.	нижче середн.
Контроль витрат	високий	середній	вище середн.	низький	нижче середн.	вище середн.	низька
Гарантія успіху	невисока	вище середн.	середня	висока	висока	вище середн.	середня
Рівень ризику	високий	низький	нижче середн.	низький	низький	середній	вище середн.

Залученість замовника	низька	середня	середня	вище середн.	середня	нижче середн.	висока
Простота використання	висока	середня	низька	вище середн.	нижче середн.	нижче середн.	низька
Повернення до попередньої фази	–	–/+	+	+	–/+	–/+	+

1.2 Мікросервісна архітектура програмних систем

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, при якому система складається з невеликих незалежних сервісів, кожен з яких виконує окрему бізнес-функцію. Ці мікросервіси можуть бути розгорнуті, оновлені та масштабовані окремо один від одного. Ось основні принципи та переваги мікросервісної архітектури:

Основні принципи мікросервісної архітектури:

– кожен мікросервіс відповідає за одну конкретну функцію або бізнес-можливість;

– розгортатися незалежно один від одного, що спрощує оновлення та масштабування;

– розподіленість мікросервіси розташовуються в різних середовищах і взаємодіють через добре визначені API;

– кожен мікросервіс має власну базу даних, що дозволяє уникнути централізованого управління даними;

– мікросервіси можуть бути розроблені з використанням різних мов програмування та технологій, що відповідають їхнім вимогам.

Переваги мікросервісної архітектури:

– гнучкість розробки. Команди можуть розробляти та оновлювати мікросервіси незалежно, що прискорює процес розробки;

– масштабованість. Легше масштабувати окремі сервіси, ніж всю систему цілком.

– стійкість. Збій одного мікросервісу не призведе до збою всієї системи;

– простота тестування та розгортання, легше тестувати та розгортати малі сервіси, що зменшує ризик помилок;

– реалізація різних бізнес-вимог. Легко додавати нові функції, створюючи нові мікросервіси.

Виклики мікросервісної архітектури:

– складність управління: велика кількість мікросервісів ускладнює їхнє управління та моніторинг;

– мережеві затримки: комунікація між мікросервісами може призводити до затримок і проблем з продуктивністю;

– транзакційна цілісність: забезпечення транзакційної цілісності між різними мікросервісами є складним завданням;

– сек'юритизація: кожен мікросервіс потребує належного захисту, що ускладнює загальну безпеку системи.

Інструменти та технології для мікросервісів:

– контейнеризація включає в себе Docker і Kubernetes;

– представниками API Gateway є Kong і AWS API Gateway;

– сервісна сітка Istio чи Linkerd;

– моніторинг і логування реалізовані за допомогою Prometheus, ELK Stack (Elasticsearch, Logstash, Kibana);

– CI/CD: Jenkins, GitLab CI, CircleCI.

Мікросервісна архітектура дозволяє створювати більш гнучкі, масштабовані та стійкі програмні системи, але вимагає відповідного підходу до управління, безпеки та моніторингу. Це підхід, який стає все більш популярним у сучасній розробці програмного забезпечення завдяки своїм численним перевагам.

1.3 Проблеми процесу міжсервісної комунікації

Процес міжсервісної комунікації в мікросервісній архітектурі може викликати ряд проблем, які потрібно враховувати для забезпечення стабільності та ефективності системи. Ось основні проблеми, пов'язані з міжсервісною комунікацією.

По-перше, важливо враховувати мережеві затримки і нестабільність. Мережеві з’єднання можуть бути повільними або ненадійними, що може призвести до затримок в обміні даними між мікросервісами або навіть до втрати запитів. Затримки можуть накопичуватися при кожному виклику сервісу, призводячи до значного збільшення часу відгуку системи. Втрати пакетів в мережі потребують механізмів повторної відправки та обробки помилок.

Далі, управління транзакціями є складним завданням у контексті забезпечення консистентності даних між кількома мікросервісами. Традиційні механізми транзакцій, такі як двофазний коміт, можуть бути неефективними або непрактичними в розподілених системах. Підтримка ACID-властивостей (атомарність, консистентність, ізольованість, стійкість) стає складнішою, тому часто використовуються інші моделі, такі як BASE (Basically Available, Soft state, Eventually consistent).

При збільшенні кількості запитів на міжсервісну комунікацію виникають проблеми масштабованості і навантаження. Потрібно правильно розподіляти запити між сервісами, щоб уникнути перевантаження окремих сервісів, забезпечуючи балансування навантаження. Високе навантаження на мережу може призводити до додаткових витрат на інфраструктуру та зниження продуктивності.

Забезпечення безпеки міжсервісної комунікації є критично важливим аспектом. Необхідно забезпечити аутентифікацію і авторизацію, щоб тільки авторизовані сервіси могли взаємодіяти один з одним. Передача даних повинна бути захищена за допомогою шифрування для запобігання перехоплення та несанкціонованого доступу.

Моніторинг і налагодження є ще одним важливим завданням. Відстеження і налагодження проблем в міжсервісній комунікації є складним завданням, тому

потрібно мати централізовані системи логування і трасування для відстеження запитів через всі мікросервіси. Важливо моніторити продуктивність кожного сервісу та комунікаційних каналів для виявлення вузьких місць і проблем.

Залежності між сервісами також можуть ускладнити підтримку та розвиток системи. Велика кількість залежностей може призвести до складнощів у розгортанні та оновленні сервісів. Виникнення циклічних залежностей може ускладнити ці процеси, а зміна в одному сервісі може вплинути на інші сервіси, що ускладнює тестування та розгортання.

Для ефективного управління міжсервісною комунікацією в мікросервісній архітектурі необхідно використовувати надійні інструменти та підходи для балансування навантаження, забезпечення безпеки, моніторингу та управління залежностями. Використання технологій, таких як сервісні сітки (service meshes), API-шлюзи (API gateways) та інші інструменти, може значно полегшити ці завдання.

1.4 Постановка задачі

В умовах сучасного розвитку програмного забезпечення та IT-інфраструктури, дослідження та аналіз процесів взаємодії сервісів у мікро-сервісній архітектурі є надзвичайно важливими завданнями. Зростаюча складність систем, підвищені вимоги до масштабованості та надійності, а також необхідність швидкого впровадження нових функцій вимагають від розробників впровадження новітніх технологій та методів для забезпечення ефективної роботи мікросервісів.

Основна задача роботи полягає в аналізі існуючих методів та інструментів взаємодії між мікросервісами. Для досягнення визначеної мети необхідно:

– провести порівняльну характеристику технологій, які використовуються для взаємодії мікросервісів, включаючи їхні переваги та недоліки;

– провести огляд існуючих систем та виявити їх недоліки, що впливають на ефективність та надійність;

– провести емпіричні вимірювання для оцінки ефективності запропонованих методів на основі реальних даних про взаємодії сервісів;

– виявити, які технології підходять для вирішення конкретних завдань на основі розробленого додатку;

– розробити рекомендації щодо впровадження запропонованих методів в реальні системи для підвищення їх ефективності та надійності.

– розробити програму для тестування та аналізу різних механізмів і видів зв'язку між мікросервісами, що забезпечують ефективну, надійну та безпечну роботу мікро-сервісної архітектури.

2 ЗАСОБИ І ТЕХНОЛОГІЇ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ

2.1 Аналіз критеріїв вибору технології реалізації міжсервісного зв’язку

Вибір технології реалізації міжсервісного зв'язку в мікросервісній архітектурі є важливим етапом, який може суттєво вплинути на продуктивність, масштабованість та надійність системи. Основні критерії, які варто враховувати при виборі технології, включають наступні.

По-перше, потрібно визначити тип зв'язку: синхронний чи асинхронний. Синхронний зв'язок (наприклад, REST, gRPC) зручний для запит-відповідь сценаріїв, де один сервіс викликає інший і очікує негайної відповіді. У той же час, асинхронний зв'язок (наприклад, RabbitMQ, Kafka) підходить для сценаріїв, де сервіси взаємодіють через повідомлення або події, і не потребують негайної відповіді.

Далі варто враховувати продуктивність і затримку. Технології, такі як gRPC, можуть забезпечити вищу продуктивність та меншу затримку порівняно з REST через використання протоколу HTTP/2 та ефективніші схеми серіалізації даних (наприклад, Protocol Buffers).

Ще один важливий аспект — це надійність та гарантії доставки. Для критичних систем важливо мати гарантії доставки повідомлень. Наприклад, Apache Kafka надає надійні механізми доставки та можливості відновлення.

Масштабованість також є ключовим фактором. Необхідно враховувати можливості масштабування обраної технології. Наприклад, Kafka легко масштабується горизонтально і може обробляти великий обсяг повідомлень.

Важливо також оцінити складність впровадження і обслуговування. Вибір технології залежить від наявного досвіду команди та складності інтеграції. Наприклад, налаштування RabbitMQ може бути складнішим порівняно з використанням REST API.

Модель даних та протокол передачі мають значення. Технології, які використовують бінарні протоколи (наприклад, gRPC), зазвичай ефективніші з точки зору використання мережі та швидкості серіалізації/десеріалізації.

Безпека є ще одним важливим критерієм. Потрібно враховувати можливості захисту даних та аутентифікації/авторизації, які надає технологія. Наприклад, gRPC підтримує TLS з коробки.

Інтеграція з наявною інфраструктурою також є важливою. Технологія має добре інтегруватися з іншими частинами системи та інструментами, які вже використовуються (наприклад, системи моніторингу, логування, трасування).

Нарешті, варто враховувати вартість впровадження та експлуатації. Вибір технології може залежати від наявних фінансових ресурсів та бюджету на підтримку і розвиток системи.

Популярні технології для міжсервісного зв'язку в мікросервісній архітектурі можуть суттєво впливати на ефективність, продуктивність та масштабованість системи. Розглянемо декілька з них.

REST (Representational State Transfer) є однією з найпоширеніших технологій. Її основні переваги включають простоту, широку підтримку та зрозумілість для розробників. Однак, REST має деякі недоліки, такі як високі накладні витрати через використання HTTP, що робить його менш ефективним для високонавантажених систем.

gRPC (Google Remote Procedure Call) є ще однією популярною технологією, яка забезпечує високу продуктивність завдяки підтримці бінарного протоколу. Вона добре підходить для поліморфних і типізованих даних. Однак, gRPC має свої мінуси: високу складність налаштування та необхідність використання специфічних інструментів для генерації кодів.

Message Brokers, такі як RabbitMQ та Apache Kafka, підтримують асинхронну взаємодію між сервісами. Вони забезпечують високу надійність та можливості відновлення. Проте, ці технології можуть бути складними у налаштуванні і управлінні, а також потребують додаткових накладних витрат на інфраструктуру.

GraphQL пропонує гнучкість у запитах, дозволяючи отримувати тільки необхідні дані. Це може значно підвищити ефективність взаємодії між сервісами. Однак, налаштування схеми може бути складним, а великі запити можуть створювати проблеми з продуктивністю.

Вибір конкретної технології для міжсервісного зв'язку залежить від вимог проекту, архітектурних рішень, наявних ресурсів та компетенцій команди.

2.2 Формат серіалізації даних

Формат серіалізації даних є важливим аспектом при виборі технології для міжсервісного зв'язку. Вибір формату серіалізації впливає на продуктивність, ефективність використання мережі, сумісність між мовами програмування та простоту інтеграції. Розглянемо аналіз основних форматів серіалізації даних.

По-перше, популярним вибором є JSON (JavaScript Object Notation). Його основні переваги включають легкість читання та розуміння, оскільки це людиночитабельний текстовий формат. Крім того, JSON підтримується практично у всіх мовах програмування і забезпечує гнучкість, підтримуючи складні структури даних, такі як об'єкти та масиви. Проте, JSON має і свої недоліки: він менш ефективний з точки зору продуктивності та використання мережі порівняно з бінарними форматами, а розмір даних у текстовому форматі більший. Також відсутність явної типізації може призводити до помилок.

Іншим популярним форматом є XML (eXtensible Markup Language). Його переваги включають гнучкість для складних і структурованих даних, можливість розширення за допомогою атрибутів і тегів, а також широке використання в різних галузях та стандартах. Проте, XML має високі накладні витрати на серіалізацію та десеріалізацію, великий обсяг даних через надлишкові теги і складність обробки порівняно з JSON.

Protocol Buffers (protobuf) є ефективним з точки зору швидкості та використання мережі. Його переваги включають високу продуктивність, компактність завдяки бінарному формату та явну типізацію, що забезпечує

надійність даних. Однак, protobuf вимагає попереднього визначення схем (schema) і генерації коду, а бінарний формат складний для читання та налагодження людиною.

Avro також є ефективним бінарним форматом. Його переваги включають високу продуктивність, сумісність завдяки включенню схеми в повідомлення, що полегшує еволюцію даних, та гнучкість з підтримкою складних типів даних і динамічної типізації. Однак, Avro потребує попереднього визначення схеми та певної конфігурації, а його бінарний формат ускладнює налагодження.

MessagePack пропонує менший розмір даних у порівнянні з JSON і високу швидкість серіалізації/десеріалізації. Він підтримується багатьма мовами програмування. Недоліками MessagePack є складність налагодження через бінарний формат та менш явна типізація порівняно з protobuf чи Avro.

Thrift забезпечує ефективний бінарний формат, явну типізацію для надійності даних та підтримку багатьох мов програмування. Однак, як і protobuf, Thrift вимагає попереднього визначення схем і генерації коду, а його бінарний формат складний для читання людиною.

Вибір формату серіалізації залежить від конкретних вимог системи. Для простих та універсальних задач, де важлива сумісність і легкість налагодження, підійдуть текстові формати, такі як JSON або XML. Для високопродуктивних систем, де важлива ефективність використання ресурсів та мережі, краще обирати бінарні формати, такі як Protocol Buffers, Avro чи Thrift. Правильний вибір формату серіалізації допоможе забезпечити оптимальну продуктивність, масштабованість та надійність міжсервісного зв'язку в мікросервісній архітектурі.

2.3 Дослідження засобів і технологій міжсервісного зв’язку

Дослідження засобів і технологій міжсервісного зв’язку охоплює аналіз та порівняння різних підходів і технологій, які використовуються для забезпечення ефективного, надійного і безпечного обміну даними між сервісами в сучасних програмних системах. У контексті мікросервісної архітектури, де системи

складаються з незалежних, дрібнозернистих сервісів, які взаємодіють між собою через визначені інтерфейси, вибір засобів і технологій міжсервісного зв’язку стає критичним аспектом розробки. Розглянемо основні засоби та технології міжсервісного зв’язку.

Першою технологією є HTTP/REST. Цей підхід використовує HTTP протокол для комунікації між сервісами. Його переваги включають простоту, широке розповсюдження та підтримку багатьох мов програмування. Однак, REST має порівняно низьку продуктивність і відсутність підтримки асинхронних викликів.

Наступним є gRPC, який використовує HTTP/2 та Protocol Buffers для ефективної передачі даних. Основні переваги gRPC включають високу продуктивність, підтримку асинхронної комунікації та стиснення даних. Недоліками є складність у налаштуванні та менш поширене використання у порівнянні з REST.

GraphQL є ще однією технологією, де клієнти запитують тільки ті дані, які їм необхідні, через єдиний кінцевий точковий інтерфейс. Це забезпечує гнучкість запитів і зменшення надлишковості даних. Проте, GraphQL може бути складним у реалізації та підтримці, і можуть виникати проблеми з продуктивністю при складних запитах.

Для асинхронної комунікації між сервісами часто використовуються Message Queues (Черги повідомлень). Технології, такі як RabbitMQ, Apache Kafka, та Amazon SQS, забезпечують високу надійність, асинхронність та підтримку горизонтального масштабування. Однак, налаштування цих систем може бути складним, і необхідно обробляти ідентифікатори повідомлень.

Ще одним підходом є Service Mesh, представлений технологіями Istio та Linkerd. Це використання окремого шару інфраструктури для керування комунікацією між сервісами. Переваги Service Mesh включають управління трафіком, безпеку та спостережуваність. Проте, розгортання та підтримка Service Mesh можуть бути досить складними.

Останнім є підхід Event-Driven Architecture (Архітектура, орієнтована на події). Технології, такі як Apache Kafka та AWS EventBridge, використовують події для ініціювання дій в інших сервісах. Це забезпечує високу гнучкість, асинхронність та підтримку реактивних програм. Водночас, налаштування цієї архітектури є складним, і вимагає ретельного планування потоків даних.

Вибір конкретної технології міжсервісного зв’язку залежить від вимог проекту, архітектурних рішень, наявних ресурсів та компетенцій команди.

Таблиця 2.1

Порівняльна таблиця

Технологія	**Переваги**	**Недоліки**
HTTP/REST	Простота, сумісність	Низька продуктивність, синхронність
gRPC	Висока продуктивність, асинхронність	Складність налаштування
GraphQL	Гнучкість запитів	Складність реалізації
Message Queues	Надійність, асинхронність	Складність налаштування
Service Mesh	Управління трафіком, безпека	Складність розгортання
Event-Driven	Гнучкість, асинхронність	Складність налаштування

Вибір конкретної технології залежить від вимог вашої системи, таких як продуктивність, масштабованість, простота в реалізації та підтримці, а також специфікації бізнес-процесів. Іноді оптимальним рішенням є поєднання декількох технологій для досягнення найкращих результатів.

3 ОПИС ПРОГРАМНОЇ СИСТЕМИ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ТЕХНОЛОГІЙ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

3.1 Дизайн програмної системи

На рисунку 3.1 показана загальна архітектура системи, яка використовує різні типи архітектур, такі як клієнт-серверна, тришарова та мікросервісна, що взаємодіють між собою за допомогою шини повідомлень [9].

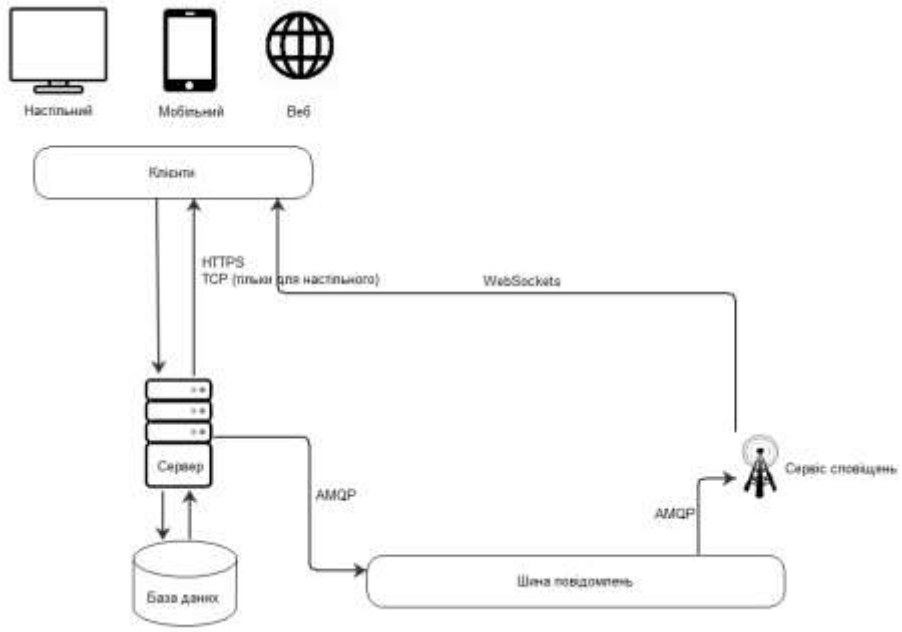

Рисунок 3.1 – Загальна архітектура системи

На рисунку 3.1 видно, що система має дві основні взаємодіючі сторони: клієнт та сервер. Серверна частина системи відповідає за обробку запитів від клієнтів, обробку та зберігання даних, а також взаємодію з іншими модулями. Клієнтська частина відповідає за здійснення запитів до сервера та відображення даних, що отримані від сервера. Крім того, серверна частина взаємодіє з іншими сервісами, такими як сервіс сповіщень.

Для взаємодії між клієнтом та сервером обрано два протоколи: HTTPS та TCP. Обрано два протоколи з огляду на наявність двох версій системи: публічної та корпоративної.

Публічна версія системи дозволяє будь-кому створювати проекти та запрошувати до участі користувачів. Цей сервіс розгортається на потужностях компанії.

Корпоративна версія системи є аналогічною до публічної, але розгортається в корпоративній мережі. Це обумовлено збільшеним запитом на безпеку даних та бажанням обмежити доступ до даних.

Для публічної версії обрано протокол HTTPS через його можливість легкого доступу до даних через Web API у стилі REST-архітектури.

Для корпоративної версії можна використовувати як HTTPS, так і TCP. HTTPS забезпечує більшу гнучкість, але TCP може забезпечити більшу швидкість передачі даних.

Архітектура сервера виконана за тришаровою архітектурою, яка буде розглянута детальніше на рисунку 3.2.

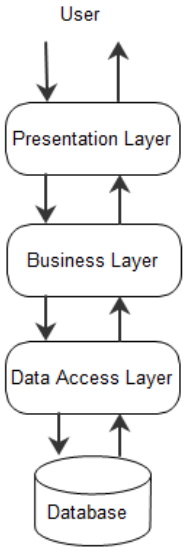

Рисунок 3.2 – Трьохшарова архітектура

Ця архітектура складається з трьох основних шарів:

– шар доступу до даних (Data Access Layer, DAL): на цьому рівні відбувається отримання та збереження даних. Сюди входять реляційні та нереляційні бази даних, веб-сервіси та інші джерела даних;

– шар бізнес-логіки (Business Logic Layer, BLL): на цьому рівні відбувається обробка даних, отриманих з рівня доступу до даних, і передача їх

до рівня презентації та навпаки. Тут також здійснюються перевірки на відповідність бізнес-правилам;

– шар презентації (Presentation Layer): на цьому рівні відбувається отримання даних з нижчих рівнів та від користувача. Ці дані можуть бути представлені у вигляді настільного додатку, веб-додатку, Web API та інших інтерфейсів.

У нашій системі для збереження даних використовується реляційна база даних. Відповідно, для рівня доступу до даних використовується відповідний провайдер БД.

Рівень представлення даних реалізується через Web API для публічної версії, а також через RPC (Remote Procedure Call) для корпоративної. Використання Web API у стилі REST дозволяє нам створювати клієнтські додатки для будь-яких платформ. Формат даних JSON використовується для REST, а XML - для RPC.

Клієнт і сервер виконані за трьошаровою архітектурою:

– шар доступу до даних: для клієнта це включає REST-клієнт для взаємодії з Web API через протокол HTTPS та RPC-клієнт для взаємодії з RPC сервером через протокол TCP;

– шар бізнес-логіки: виконується на клієнті або сервері, залежно від того, де відбувається обробка даних та логіка програми;

– шар презентації: для публічної версії може бути виконаний у вигляді настільного, мобільного або веб-додатку. Для корпоративної версії - лише у вигляді настільного додатку.

Оскільки система має багато модулів, для певних видів задач прийнято рішення використовувати архітектуру мікросервісів.

Мікросервісна архітектура передбачає розбиття додатку на невеликі, незалежні мікросервіси, кожен з яких відповідає за виконання конкретної функціональності. Це дозволяє підтримувати гнучкість, масштабованість та швидкість розгортання.

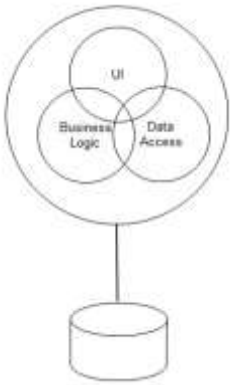

Рисунок 3.3 – Монолітна архітектура

Отже, замість того, щоб мати один додаток на сервері, який вирішує всі обмежені контексти, ми застосовуємо кілька менших додатків (див. рисунок 2.4), кожен з яких спеціалізується на певному обмеженому контексті. Ці додатки функціонують на окремих серверах і взаємодіють між собою через мережу.

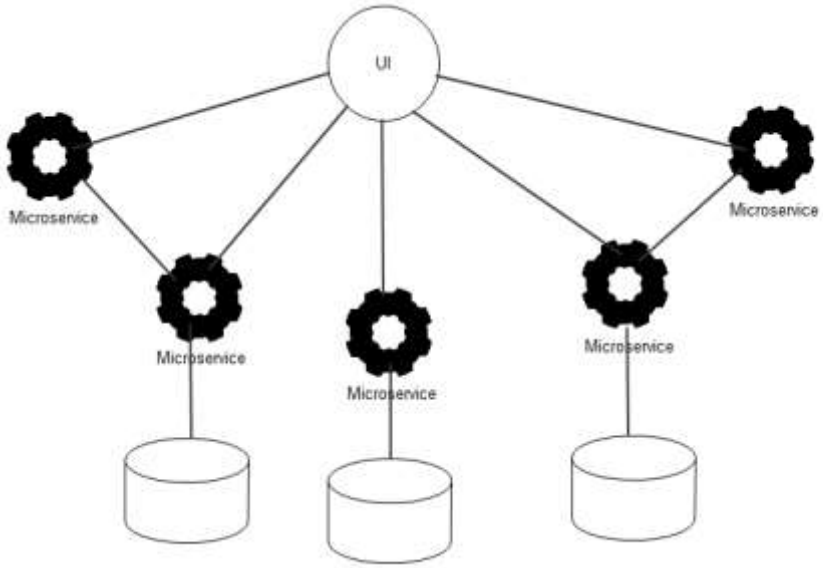

Рисунок 3.4 – Архітектура мікросервісів

У нашому випадку, додаток з обмеженим контекстом - це сервіс сповіщень. Основне завдання цього сервісу полягає в тому, щоб повідомляти користувача в реальному часі про події, які відбулися і потребують його уваги з мінімальною затримкою. Існують різні методи доставки повідомлень користувачеві, такі як polling, long polling, SSE та використання WebSockets.

У методі polling клієнт періодично запитує сервер, чи є для нього нові повідомлення. Long polling працює на тому ж принципі, але сервер не надсилає

відповідь до тих пір, поки не з'явиться нове повідомлення або не мине максимальний час очікування. У методі SSE (Server-Sent Events) клієнт встановлює постійне однонаправлене з'єднання з сервером, через яке сервер надсилає повідомлення користувачеві. З використанням WebSockets клієнт та сервер підтримують постійне двонаправлене з'єднання, що дозволяє обмінюватися повідомленнями в обидва напрямки.

Ми обрали використання WebSockets для доставки повідомлень до клієнта. Навіть якщо наразі нам потрібно лише відправляти сповіщення від сервера до клієнта, ми обрали двонаправлене з'єднання, щоб розширити можливості у майбутньому, наприклад, для отримання підтвердження про доставку повідомлення.

3.2 Опис обраних технологій

Для організації взаємодії між мікросервісами був обраний протокол AMQP [11]. AMQP (Advanced Message Queuing Protocol) – це публічний протокол для передачі повідомлень між компонентами систем. Основна ідея полягає в тому, що окремі підсистеми або незалежні додатки можуть обмінюватися повідомленнями через AMQP-брокер, який відповідає за маршрутизацію, забезпечує гарантовану доставку, розподіл потоків даних та підписку на необхідні типи повідомлень.

AMQP включає три основні поняття:

- повідомлення (message) - це одиниця даних, яка надсилається. Головна його частина (зміст) не інтерпретується сервером. До повідомлення можна додавати структуровані заголовки;
- точка обміну (exchange) - це місце, куди надходять повідомлення. Точка обміну розподіляє повідомлення в одну чи декілька черг. При цьому в точці обміну повідомлення не зберігаються;
- черга (queue) - це місце, де повідомлення зберігаються до тих пір, поки їх не забере клієнт.

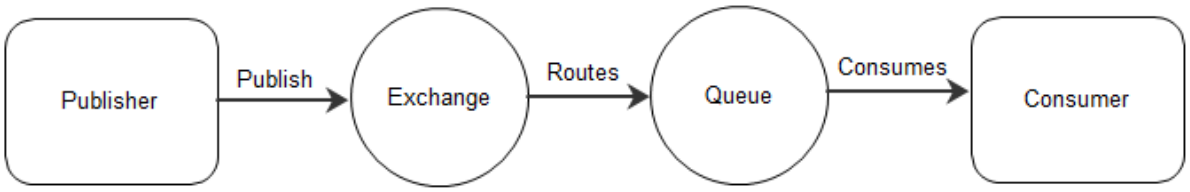

Рисунок 3.5 – Головні компоненти AMQP протокола

З рисунку 3.5 видно, що протокол AMQP має дві взаємодіючі сторони: видавця (publisher), який створює повідомлення та передає його до точки обміну, і споживача (consumer), який забирає повідомлення з черги та обробляє їх.

У нашій системі, згідно з архітектурою, зображеною на рисунку 3.1, у ролі видавця виступає «Сервер», який створює повідомлення та надсилає їх до точки обміну. Один з можливих споживачів - це сервіс сповіщень, який обробляє повідомлення, пов'язані з надсиланням користувачам сповіщень про певні події.

Для забезпечення слабкого зв'язку між модулями, їхньої легкої інтеграції та написання модульних та інтеграційних тестів, ми застосували один із принципів ООП - інверсію управління.

Інверсія управління (Inversion of Control, IoC) - важливий принцип об'єктно-орієнтованого програмування, що використовується для підвищення слабкого зв'язку між модулями. Це архітектурне рішення для інтеграції, що полегшує розширення можливостей системи, при якому контроль над потоком управління програмою залишається за каркасом.

Одним з реалізацій IoC щодо управління залежностями є впровадження залежностей.

Впровадження залежностей (Dependency Injection, DI) - це процес предоставлення зовнішньої залежності програмному компоненту. Цей підхід використовується в багатьох фреймворках, які називаються IoC-контейнерами.

Ця система побудована з урахуванням можливості користувача самостійно створювати нові ролі та надавати їм відповідні функції та права доступу. Щоб спростити користування та заощадити час, передбачено шість "типових" ролей:

замовник, менеджер проекту, сервіс менеджер, керівник команди, розробник і тестувальник.

Детальний опис кожної ролі:

– замовник створює вимоги до програмного продукту, визначає ціну та кінцеві строки;

– менеджер проекту узгоджує ціну та кінцеві строки, створює задачі для вимог, назначає відповідальних за задачі та контролює їх виконання;

– сервіс менеджер надає сервіс незалежним замовникам, шукає виконавців, створює ролі та контролює робочий процес;

– керівник команди розбиває задачі на підзадачі, призначає відповідальних розробників та звітує про процес виконання задач;

– розробник виконує задачі якісно та вчасно, звітує про процес виконання підзадач;

– тестувальник затверджує можливість використання нового функціоналу, виконує планові тести та знаходить недоліки.

Кожна роль потребує певного візуального супроводу, що може включати в себе різні візуальні компоненти залежно від їх функцій та прав доступу. Для досягнення цієї мети використовується АРМ (автоматизована система керування).

Ми використовуємо принцип інверсії управління при написанні модулів, що забезпечує слабкозв'язність. Це дозволяє нам динамічно завантажувати різні АРМ відповідно до ролі та прав користувача. Нижче перераховано основні АРМ. За діаграмою прецедентів, зображеною на рисунку 3.6, можна виділити основні можливості замовника, такі як створення та робота з вимогами, виставлення термінів виконання та узгодження ціни.

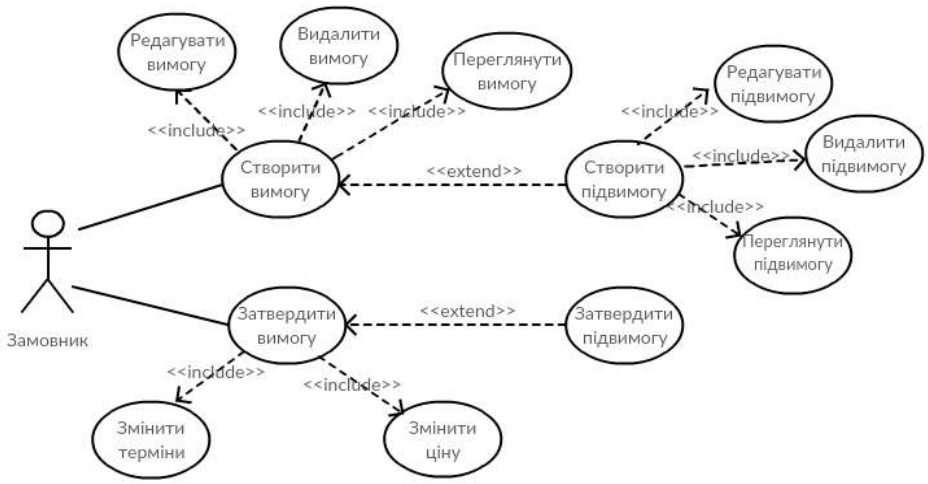

Рисунок 3.6 – Діаграма прецендентів для роботи з вимогами по відношенню до замовника

АРМ замовника використовує динамічний модуль для роботи з вимогами у повному обсязі, включаючи створення, редагування та видалення вимог, виставлення кінцевих термінів реалізації, проведення перемов щодо ціни, термінів виконання та умов, а також підтвердження вимоги зі сторони замовника. За діаграмою прецедентів, зображеною на рисунку 3.7, можна виділити основні можливості менеджера проекту, такі як перегляд існуючих вимог, створення задач та призначення керівників команд на ці задачі.

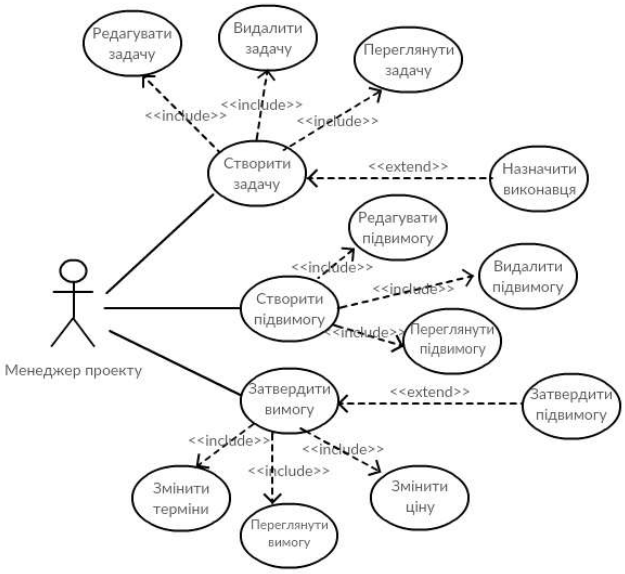

Рисунок 3.7 – Діаграма прецендентів для роботи з вимогами та задачами по відношенню до менеджера проекту

АРМ менеджера використовує динамічний модуль для роботи з вимогами з правами перегляду існуючих вимог, редагування ціни, надання підтвердження про ціну, а також динамічний модуль для роботи з задачами, що дозволяє створювати, редагувати та видаляти задачі, назначати зв’язки між задачами та вимогами, призначати керівників команд для задач, переглядати відсоток виконання задачі, встановлювати кінцеві терміни виконання задачі, що не суперечать кінцевим термінам виконання вимог. За діаграмою прецедентів, зображеною на рисунку 3.8, можна виділити можливості керівника команди, такі як перегляд існуючих задач, створення підзадач та призначення розробників на ці підзадачі.

Рисунок 3.8 – Діаграма прецендентів для роботи з задачами та підзадачами по відношенню до керівника команди

АРМ керівника використовує динамічний модуль для роботи з задачами з правами перегляду існуючих назначених задач, а також створення, редагування та видалення підзадач до задачі. Керівник може назначати кінцеві терміни виконання підзадач, що не суперечать кінцевим срокам задачі, і призначати розробників до підзадач. За діаграмою прецедентів, зображеною на рисунку 3.9,

можна виділити основні можливості розробника, такі як перегляд назначених підзадач та звітування про відсоток виконання підзадачі.

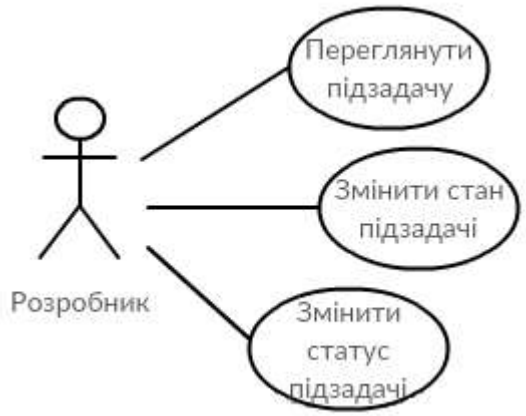

Рисунок 3.9 – Діаграма прецендентів для роботи з підзадачами по відношенню до розробника

АРМ розробника використовує динамічний модуль для роботи з задачами з правами перегляду назначених підзадач та їх задач, звітування про відсоток виконання підзадачі та відмітки про завершення задачі як виконаної.

Щодо підсистеми роботи з вимогами та "Аукціон вимог", основні можливості включають:

- створення, редагування та видалення вимог та підвимог;
- міграція з записок до вимог або підвимог;
- сповіщення про стан виконання вимоги або підвимоги, де станом розуміється процент закінченості;
- можливість логічної декомпозиції вимоги на підвимоги;
- створення набору задач на кожну підвимогу менеджером проекту;
- можливість створення задач на вимогу, але не обов'язково;
- зміна стану вимоги чи підвимоги автоматично (з урахуванням вагових коефіцієнтів) або вручну.

При автоматичній зміні стану вимоги розраховується з урахуванням вагових коефіцієнтів підвимог та задач, які задаються при їх створенні.

Наприклад, якщо вимога складається з трьох підвимог та однієї задачі, і кожен елемент має свій ваговий коефіцієнт, то стан вимоги обчислюється з урахуванням відповідних відсотків виконання кожного елемента.

У випадку ручного проставлення стану підвимоги, менеджер проекту може вручну встановлювати процент виконання з певним інтервалом у часі та додавати робочі записи до підвимоги. При цьому стан вимоги перераховується автоматично.

"Аукціон вимог" є частиною функціоналу системи, відповідальність за яку належить менеджеру проекту. В рамках цієї функціональності замовник та менеджер проводять перемовини про умови вимоги, деталізуючи та уточнюючи їх. Перед прийняттям вимоги до опрацювання вона потребує двостороннього погодження.

На прикладі: замовник створює вимогу, менеджер вносить свої пропозиції щодо ціни та термінів, після чого відбувається обговорення та узгодження умов. Коли обидві сторони приймають умови, вимога затверджується і переходить до стадії виконання.

Життєвий цикл вимоги включає наступні етапи: «To Discuss» (створена), «In Discuss» (обговорюється), «Accepted» (обговорення завершено), «In Progress» (почалася розробка) та «Completed» (завершено виконання вимоги). Також передбачено два додаткових етапи, доступних для переходу з етапів «In Discuss», «Accepted», «In Progress»: «Frozen» (обговорення чи виконання відкладено до невизначених термінів) та «Canceled» (обговорення чи виконання відмінено).

Для роботи з вимогами та підвимогами передбачено дві додаткові дії: «Branching» (розгалуження) та «Merging» (злиття). Під розгалуженням розуміється процес декомпозиції вимоги чи підвимоги на дві або більше. Під злиттям розуміється об'єднання двох або більше вимог чи підвимог. Виконання цих операцій рекомендується до етапу «Accepted», оскільки після цього етапу потребує додаткових обговорень та можливо витрат і зміщень кінцевих термінів.

Підсистема обліку задач включає наступні можливості: створення, редагування, видалення задач та підзадач, а також сповіщення про стан

виконання. Задача представляє собою набір завдань, що задовольняють вимогам чи підвимогам, або декілька задач на одну вимогу чи підвимогу. Керівник команди розбиває задачу на підзадачі та назначає розробників. Під підзадачею розуміється набір робіт, який виконує розробник.

В системі існує можливість встановлення термінів виконання для задач та підзадач. Під час виконання підзадачі розробник сповіщує про її стан виконання та відсоток завершення. Крім того, розробник має можливість створювати робочі записи, які описують хід виконання підзадачі.

Після розбиття задачі на підзадачі та їх взяття до виконання розробники сповіщають про стан виконання підзадач. Стан виконання задачі може змінюватися автоматично з урахуванням вагових коефіцієнтів або вручну керівником команди.

Автоматична зміна стану виконання задачі з урахуванням вагових коефіцієнтів відбувається аналогічно до зміни стану вимоги.

Також можливе автоматичне змінення стану виконання задачі чи підзадачі з використанням попередньо визначеного часового розподілу. Наприклад, можна налаштувати автозміну стану задачі погодинно або подобово, а також використовувати лінійний або експоненціальний розподіл зростання стану.

У випадку ручного встановлення стану задачі, керівник команди вручну встановлює відсоток виконання з певним інтервалом часу. В обох випадках, як автоматичного, так і ручного, керівник може створювати робочі записи для задачі, аналогічно до підзадач.

Коли стан підзадачі змінюється, керівник проекту отримує сповіщення про ці зміни, а щодо змін у задачі, сповіщення надходить до менеджера проекту (якщо використовуються вбудовані ролі). Фіксація цих змін та їх сповіщення здійснюються за допомогою сервісу сповіщень, який буде описаний нижче.

Життєвий цикл задачі та підзадачі складається з таких етапів: створена («To Do») – прийнята до виконання («In Progress») – виконання завершено («Completed») – успішно пройдено тестування («Accepted»).

Підсистема обліку дефектів

Основні можливості підсистеми обліку дефектів включають створення, редагування та видалення дефектів. Дефект описує недолік у системі, виявлений під час тестування.

Дефект може бути виявлений під час тестування задачі або підзадачі, які знаходяться у стані «Completed», або під час планових тестів, таких як стрес-тестування чи системні тести. Після виявлення недоліку, тестувальник створює дефект, який призначається розробнику, що відповідав за відповідну підзадачу.

Підсистема роботи з динамічними ролями та АРМ

Підсистема динамічних ролей

Основними можливостями підсистеми динамічних ролей є створення, редагування, видалення ролей та їх назначення користувачам. Динамічна роль визначає права користувача на функції, такі як перегляд, створення, редагування та видалення інших об'єктів. АРМ користувача будується динамічно відповідно до його прав на кожному конкретному проекті. Якщо користувач бере участь у різних проектах з різними ролями, система перебудовує його АРМ відповідно до прав, що належать йому в межах кожного проекту.

Підсистема сповіщень та нагадувань

Основні можливості цієї підсистеми включають сповіщення користувачів про зміни в певних об'єктах (наприклад, зміну статусу задачі) та нагадування про заплановані дії. Для доставки сповіщень використовується сервіс сповіщень. Після автентифікації користувача він автоматично підписується на отримання сповіщень, які відповідають проектам, в яких він бере участь, та його правам. Наприклад, якщо користувач має роль «Розробник», він не отримує сповіщення про зміни вимог.

Під нагадуванням розуміється сповіщення, яке надсилається користувачеві, якщо він запланував певну дію та встановив час нагадування. Наприклад, якщо менеджер проекту створює запис «Мітинг о 17:00 10.12» і налаштовує нагадування з 19:00 09.12 до 16:00 10.12, він отримуватиме нагадування на вказані часи, крім періоду «годин відпочинку», коли сповіщення не надсилаються.

Підсистема спілкування

Для ефективної комунікації між користувачами в рамках одного проекту передбачена підсистема спілкування. Вона забезпечує обмін текстовими повідомленнями з можливістю прикріплення додатків між користувачами. Для початку спілкування необхідно бути учасником відповідної групи, яка є сутністю, що об'єднує користувачів. Користувач може бути членом декількох груп і отримувати повідомлення тільки з тих, до яких він належить. Для додавання користувачів до проекту передбачена можливість генерації запрошень. Після їх підтвердження надсилається відповідне сповіщення.

Система управління версіями

Для організації спільного доступу до вихідних кодів та фіксації змін застосовуються системи управління версіями. Вони полегшують роботу з інформацією, що змінюється, надаючи можливість зберігати різні версії документів та відстежувати їх зміни. Згідно зі статистикою від RhodeCode на 2023 рік, найпопулярнішими системами є Git та Mercurial. Найбільш відомі веб-сервіси для Git - GitHub, для Mercurial - Bitbucket.

Інтеграція з веб-сервісами

При реєстрації у системі користувач може зв'язати свій обліковий запис на веб-сервісах, таких як GitHub або Bitbucket. Інтеграція полягає у можливості автоматичного відображення комітів у підзадачах та отриманні сповіщень про нові коміти в репозиторії. Зазвичай керівник команди отримує сповіщення від розробників, що є членами його команди.

Фінансова підсистема

Система надає засоби для перегляду виконаної роботи, побудови статистики та оцінки роботи для визначення суми оплати. Для зручності оплати та розподілу коштів між виконавцями передбачено інтеграцію з популярними сервісами оплати, такими як Authorize.Net, PayPal, SecurePay.com. Користувачі можуть вказати свої дані під час реєстрації або редагування облікового запису для здійснення оплати та отримання виплат за виконану роботу.

Юридична підсистема

Система має юридичну силу, і її дані вважаються достовірними для використання під час судових розглядів. Для цього передбачено можливість затвердження інформації цифровим підписом. Це вимагає залучення фахівця з юридичної галузі для консультації та реалізації цієї можливості.

Масштабування системи

При зростанні попиту на продукт збільшуються навантаження на систему. Для уникнення перевищення часу відповіді або повного краху використовуються стратегії масштабування. Один із підходів - горизонтальне масштабування, коли нові хости придбаються для розподілу навантаження. Цей підхід є перспективнішим, оскільки дозволяє легше розширювати систему залежно від потреб.

Горизонтальне масштабування системи відкриває широкі можливості для збільшення її продуктивності та надійності. З огляду на цю можливість, при подальшій розробці та розгортанні системи можна використовувати хмарні засоби, зокрема Microsoft Azure.

Microsoft Azure надає комплекс рішень, які ідеально відповідають вимогам вашої системи:

– Azure Service Bus - це потужний і надійний сервіс обміну повідомленнями між різними компонентами системи. Використання асинхронної комунікації через Azure Service Bus дозволить ефективно обмінюватися повідомленнями між мікросервісами вашої системи;

– Azure Blob Storage - це ідеальне рішення для зберігання неструктурованих даних, таких як документи, файли чи медіа-контент. Використання Azure Blob Storage спростить процес зберігання та управління великими обсягами даних в хмарі;

– Azure Notification Hubs - це масштабована платформа для відправки push-сповіщень на мобільні пристрої на різних платформах. Використання Azure Notification Hubs полегшить процес відправки сповіщень користувачам вашого мобільного додатку та допоможе забезпечити швидку доставку повідомлень.

Ці сервіси забезпечать надійну, масштабовану та ефективну інфраструктуру для вашої системи, дозволяючи зосередитися на розробці функціоналу та покращенні користувацького досвіду.

Крім того, використання хмарних обчислень дозволяє легко розгортати систему завдяки підходу PaaS (Platform-as-a-Service): всього кількома клацаннями можна розгорнути всю інфраструктуру. У періоди великого навантаження на нашу систему ми можемо за кілька хвилин підняти декілька серверів для балансування навантаження, що є однією з головних переваг хмарних сервісів.

Рекомендовані характеристики для комплексу програмно-технічного забезпечення:

а) для серверів;

1) Процесор Intel Core i5 з тактовою частотою не менше 2.5 ГГц;
2) Об'єм оперативної пам'яті від 16 ГБ;
3) Мінімум 1 ГБ вільного місця на жорсткому диску;
4) Наявність відеокарти;
5) Операційна система Windows Server 2012R2 або новіша;
6) Встановлений .NET Core версії 1.1.0;
7) Встановлений PostgreSQL починаючи з версії 9.6.

б) для шини повідомлень;

1) процесор Intel Core i5 з тактовою частотою не менше 2.5 ГГц;
2) об'єм оперативної пам'яті від 16 ГБ;
3) мінімум 4 ГБ вільного місця на жорсткому диску;
4) наявність відеокарти;
5) операційна система Windows Server 2012R2 або новіша;
6) встановлений RabbitMQ версії 3.6.2 або новішої.

в) для настільних клієнтів;

1) процесор Intel Core i3 з тактовою частотою не менше 1.8 ГГц;
2) об'єм оперативної пам'яті від 4 ГБ;
3) мінімум 256 МБ вільного місця на жорсткому диску;

4) наявність відеокарти;
5) операційна система Windows 7 або новіша;
6) встановлений .NET Framework версії 4.6.2 або новішої.

г) для мобільних клієнтів;

1) об'єм оперативної пам'яті від 2 ГБ;
2) мінімум 128 МБ фізичного вільного місця;
3) операційна система iOS версії 10 або Android версії 7 або новіші.

д) для веб-клієнтів;

1) процесор Intel Core i3 з тактовою частотою не менше 1.8 ГГц;
2) об'єм оперативної пам'яті від 2 ГБ;
3) наявність відеокарти;
4) браузер з підтримкою WebSocket та ECMAScript 6.

3.3 Програмна реалізація

Система розроблялася з використанням таких технологій та інструментів, як C#, ASP .NET Core, ASP .NET SignalR, Entity Framework Core та WPF.

C# - це мова програмування, призначена для розробки різноманітних додатків в середовищі .NET Framework. Мова C# є простою, типобезпечною та об'єктно-орієнтованою. Завдяки багатьом нововведенням, C# забезпечує можливість швидкої розробки додатків, при цьому зберігаючи елегантність мови, схожої на C. За останніми тенденціями та нововведеннями, такими як .NET Core та .NET Standard, можна здійснювати кросплатформенну розробку як під ОС Windows.

ASP .NET Core - це новий кросплатформний фреймворк з відкритим вихідним кодом для розробки сучасних хмаро-орієнтованих додатків, таких як веб-застосунки, IoT (Internet of Things) додатки та серверні частини для мобільних додатків. Він надає наступні основні можливості:

- розробка графічного веб-інтерфейсу та Web API як єдиного цілого;
- інтеграція з сучасними клієнтськими фреймворками;

- система конфігурацій, яка забезпечує підтримку хмарних обчислень;
- вбудований контейнер IoC;
- нову модульну обробку HTTP запитів;
- новий набір інструментів, які полегшують веб-розробку;
- можливість запуску додатків ASP .NET на таких операційних системах, як Windows, Mac, Linux.

ASP .NET SignalR - це бібліотека для ASP .NET, яка дозволяє розробляти "real-time" веб-додатки. SignalR забезпечує двонаправлену комунікацію між сервером та клієнтом, дозволяючи серверу відправляти контент безпосередньо користувачам, які з ним з'єднані. Вона підтримує Web Sockets та інші підходи, такі як SSE та pooling. SignalR дозволяє забезпечити відправку сповіщень користувачеві у нашій системі.

Entity Framework (EF) Core - це легка та кросплатформна версія популярного EF, яка надає доступ до даних. EF Core - це об'єктно-реляційне відображення, яке дозволяє розробникам .NET працювати з базами даних. Вона відображує реляційні таблиці на об'єкти та зменшує кількість коду для взаємодії з БД, надаючи можливість вибірки даних та виконання операцій додавання/оновлення/видалення.

Windows Presentation Foundation (WPF) - це система для побудови клієнтських додатків Windows з візуальними можливостями комунікації з користувачем. WPF базується на векторній системі візуалізації, незалежній від приладу виводу та створеній з урахуванням можливостей сучасного графічного обладнання. Вона надає інструменти для створення візуального інтерфейсу з використанням мови XAML, елементів управління, прив'язки даних, графіки, анімації та ін.

Для бази даних ми вибрали PostgreSQL через її кросплатформенність та вбудовані інструменти для горизонтального масштабування, такі як шардинг та "мультимастер". При проектуванні таблиць ми використовували третю нормальну форму, щоб уникнути дублювання даних та надмірні функціональні залежності між атрибутами таблиць.

Таблиця "UserRoleProject" служить для вирішення багаторазових зв'язків між користувачами, проектами та ролями. У цій таблиці містяться три зовнішні ключі: UserId, ProjectId та RoleId, які посилаються на відповідні записи в таблицях "User", "Project" та "Role".

Ця таблиця дозволяє встановити зв'язок між користувачами та проектами, визначаючи при цьому ролі, які вони виконують у кожному проекті. Наприклад, один користувач може мати роль "Адміністратор" в одному проекті та роль "Користувач" в іншому. Такий підхід дозволяє гнучко налаштовувати права доступу користувачів до різних проектів з різними ролями.

Використання таблиці "UserRoleProject" дозволяє ефективно керувати доступом користувачів до проектів і управляти їхніми ролями в цих проектах, забезпечуючи при цьому надійність та консистентність даних.

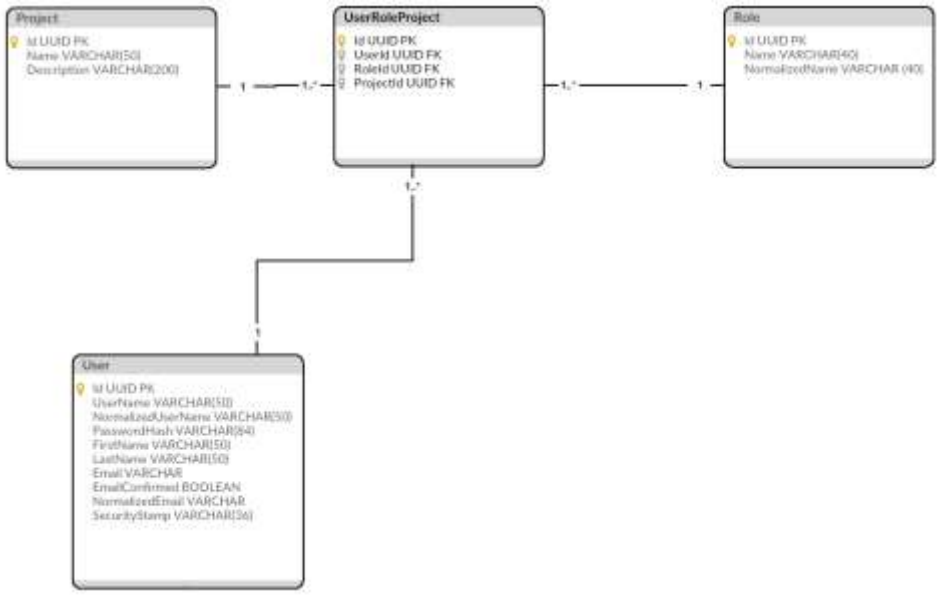

Рисунок 3.10 – Відношення між таблицями «User», «Project», «Role»

Ми забезпечуємо можливість запрошувати інших користувачів до проекту за допомогою таблиці "Invite" (див. Рисунок 3.11). Ця таблиця містить два зовнішні ключі до таблиці "User", що обумовлено необхідністю визначення автора запрошення (CreatorId) та особи, яка була запрошена (InviteeId).

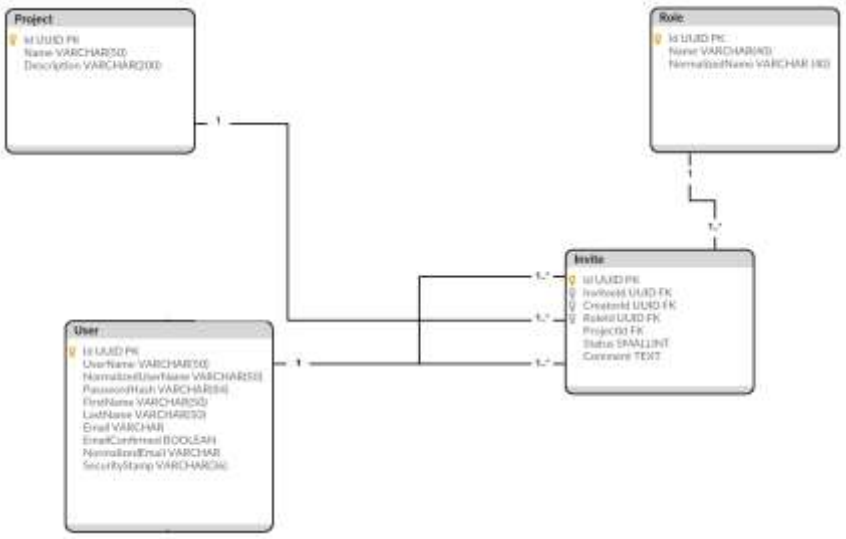

Рисунок 3.11 – Відношення таблиць «User», «Invite», «Role», «Project»

Для зберігання бінарних даних була створена таблиця "Attachment". Файли можуть бути прикріплені як до вимог, так і до задач. Для економії місця та прискорення часу очікування ми надаємо можливість вибрати вже завантажені файли. Отже, ми маємо відповідні таблиці зв'язку "багато до багатьох": "RequirementAttachment" та "TaskAttachment".

Серверна частина реалізована у трьохшаровій архітектурі. Давайте розглянемо реалізацію кожного шару.

Шар доступу до даних реалізовано з використанням таких шаблонів як UnitOfWork та Repository (див. додаток А). Шаблон Repository (див. Рисунок 3.12) забезпечує роботу з даними бази даних, таку як вибірка, додавання, оновлення, видалення. Для кожної сутності створюється свій репозиторій. Для перетворення реляційних даних (дані з бази даних) у об'єкти використовується Entity Framework Core. Він надає можливість перетворення C# лямбда-виразів у SQL запити, що спрощує процес розробки.

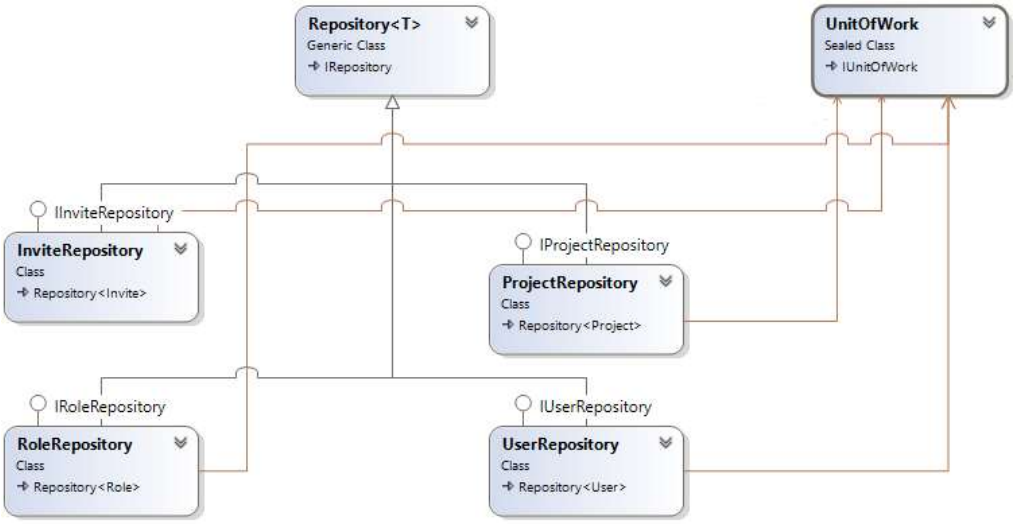

Рисунок 3.12 – Діаграма класів UnitOfWork та Repository

Шаблон UnitOfWork відіграє роль контексту взаємодії з базою даних. У нашій серверній частині, під контекстом слід розуміти період часу від моменту надходження запиту до його обробки (час між надсиланням запиту та отриманням відповіді). Тобто, контекст існує протягом цього періоду часу та є індивідуальним для кожної сесії запит-відповідь. Він об'єднує всі репозиторії, тобто, дозволяє виконувати операції вибірки, створення, оновлення та видалення даних для будь-якої сутності, а потім, після виклику методу SaveChanges (який також є членом даного класу), у разі потреби фіксує зміни одним або кількома запитами.

Важливо відзначити, що метод SaveChanges підтримує асинхронну модель взаємодії з використанням CancellationToken, призначеного для відміни дії, якщо це можливо, користувачем під час виконання.

Шар бізнес-логіки відповідає за такі основні операції як перевірка прав доступу, перевірка надісланих даних на відповідність бізнес-правилам та взаємодію між шарами презентації та доступу до даних. Під перевіркою прав слід розуміти процес, що визначає, чи може користувач виконувати певні дії. Наприклад, користувач з роллю "Замовник", який намагається відредагувати задачу, отримає помилку, оскільки не має на це прав. Під перевіркою даних на відповідність бізнес-правилам слід розуміти процес, що визначає, чи відповідають отримані дані з шару презентації бізнес-правилам. Наприклад,

якщо користувач бажає зареєструватися в системі та вводить більше 20 символів у поле "Username", він отримає помилку, оскільки для цього поля дозволено лише 20 символів.

Кожен клас, призначений для комунікації з шаром презентації, має суфікс "Manager". Ці сутності інкапсулюють роботу з шаром доступу до даних та виконують вищезгадані операції. Також, за необхідності, вони здійснюють конвертацію сутностей. Наприклад, якщо є сутність "User", що має поле "PasswordHash", і користувач не повинен отримувати це поле, для цього використовується сутність "UserInfo", що не містить цього поля, а "Manager" перетворює об'єкти з "User" на "UserInfo" та передає їх шару презентації.

Шар презентації реалізовано за допомогою ASP .NET Core MVC з використанням Web API. Раніше Microsoft мав два окремі фреймворки для Web API та MVC: ASP .NET Web API та ASP .NET MVC, але зараз ми маємо один фреймворк ASP .NET Core MVC, який об'єднав їх.

Для побудови Web API ми використовували REST архітектуру, яка використовує чотири основні HTTP методи:

- GET – для отримання даних;
- POST – для створення даних;
- PUT – для зміни даних;
- DELETE – для видалення даних.

За замовчуванням, URL складається з IP-адреси, номера порту, сегмента "api", імені контролера, а для виклику певного методу використовуються HTTP методи. Пізніше, після випуску першої версії системи, буде куплено DNS ім'я, яке замінить IP адресу та номер порту.

Усі контролери у системі наслідуються від BaseController, який в свою чергу наслідується від Controller з фреймворку ASP .NET Core MVC. Базовий контролер містить загальну логіку для всіх дочірніх контролерів.

За замовчуванням, всі користувачі повинні бути автентифіковані у системі, і для цього використовується атрибут "AuthorizeAttribute", який застосовується до базового контролера. Для деяких методів, до яких можуть мати доступ

незареєстровані користувачі, використовується атрибут "AllowAnonymousAttribute".

До базового контролера застосовано також два основні атрибути: "RouteAttribute" та "HandleExceptionFilterAttribute". Перший використовується для задання шаблону маршруту, а другий - для обробки помилок, що виникають під час обробки запитів. Ці атрибути допомагають забезпечити коректну обробку помилок та повернення інформативних повідомлень користувачам.

Для автентифікації ми використовуємо токени, а саме JSON Web Token (JWT). Це безпечний спосіб передачі інформації про користувача між двома сторонами. Інформація у JWT кодується у вигляді JSON об'єкта, який може бути підписаний для перевірки цілісності або зашифрований для забезпечення конфіденційності.

Для роботи з JWT токенами використовується бібліотека Microsoft.AspNetCore.Authentication.JwtBearer, яка дозволяє створювати токени з різними параметрами шифрування та іншими налаштуваннями.

Однією з переваг використання токенів є їх безстандартність (statelessness), що спрощує процес горизонтального масштабування веб-додатків.

Для реалізації функцій, таких як реєстрація користувачів, присвоєння ролей тощо, ми використовуємо бібліотеку ASP .NET Core Identity. Вона надає вбудовану систему автентифікації та авторизації, що дозволяє користувачам створювати облікові записи, аутентифікуватися, керувати своїми обліковими записами та використовувати зовнішні провайдери для входу, такі як Facebook, Google, Microsoft, Twitter та інші.

ASP .NET Core Identity має реалізацію, яка працює з базою даних через Entity Framework Core, але ми перевизначили необхідні інтерфейси з використанням нашої реалізації UnitOfWork, що підходить до архітектури нашого додатку.

ASP .NET Core має вбудований контейнер інверсії управління (IoC), який дозволяє встановлювати відповідності між абстракціями та їх реалізаціями з вибором стратегії життєвого циклу об'єкта. Виділяють три основні стратегії:

- transient – створюється новий об'єкт при кожному запиті до контейнера;
- scoped – створюється один об'єкт на кожен запит до контейнера в рамках однієї сесії;
- singleton – об'єкт створюється лише один раз і використовується для всіх запитів.

У нашому випадку зазвичай використовується стратегія scoped, оскільки наші менеджери та контексти до БД існують в рамках запиту-відповіді.

До розгортання додатків ASP .NET на веб-серверах IIS, але оскільки ASP .NET Core кросплатформений, виникла необхідність відв'язати його від IIS та Windows. Наразі ми маємо можливість розгортати додатки на стандартних веб-серверах IIS або IIS Express, або запускати їх без IIS, використовуючи WebListener або Kestrel.

WebListener працює лише на Windows, тоді як Kestrel є кросплатформеним. Оскільки ми акцентуємо на кросплатформеності, ми використовуємо Kestrel.

Клієнтська частина нашого додатку реалізована під ОС Windows та використовує трьохшарову архітектуру. Шар доступу до даних представляє собою реалізацію REST-клієнта, який взаємодіє з сервером. Він включає три основні сутності: RestClient, CleanSlateRestClient та сервіси для взаємодії з контролерами.

RestClient використовується для реалізації основних HTTP методів, таких як GET, POST, PUT, DELETE, і має різні варіації використання. Для його реалізації використовується бібліотека RestSharp, яка надає можливість створення HTTP запитів.

CleanSlateRestClient агрегує усі сервіси та створює їх реалізації. Порівняно з реалізацією шару доступу до даних на серверній частині, CleanSlateRestClient схожий на шаблон UnitOfWork.

Сервіси використовуються для виклику відповідного методу в контролері на сервері та передачі готових об'єктів на шар бізнес-логіки. Наприклад, якщо у нас є контролер AccountController з методом Token, який потрібно викликати за

допомогою HTTP методу POST, на клієнтській стороні ми маємо AccountService з методом GetToken. Завдання методу GetToken полягає в отриманні імені користувача («username») та паролю, виклику методу POST у RestClient, отриманні даних та їх передачі на шар бізнес-логіки. Кожен клас сервісу, призначений для комунікації з верхнім шаром, має закінчення «Service».

Шар бізнес-логіки відповідає за первинну перевірку відповідності отриманих даних з відповідністю бізнес-правилам та виклик необхідного методу з шару доступу до даних, а також за передачу отриманих даних на верхній шар.

Первинна перевірка відповідності отриманих даних здійснюється для зменшення кількості запитів до сервера та скорочення часу відповіді. Якщо надійшли невірні дані, користувач негайно отримає результат, оскільки запит до сервера не буде створено. Кожен клас, призначений для комунікації з шаром презентації, має закінчення «Manager».

Шар презентації реалізований як клієнтський додаток під операційною системою Windows з використанням технології WPF (Windows Presentation Foundation). Цей шар базується на шаблоні MVVM (Model-View-ViewModel) (див. рисунок 3.13), який складається з трьох основних частин:

– модель (Model): модель представляє собою бізнес-логіку та фундаментальні дані, що необхідні для роботи додатку. Вона відповідає за обробку даних та логіку програми;

– представлення (View): представлення відображає графічний інтерфейс додатку, такий як вікна, кнопки і т. д. Воно підписується на події зміни значень властивостей чи команд, які надає Модель Представлення. Якщо в Моделі Представлення змінилась будь-яка властивість, то всі підписані на цю подію об'єкти отримають сповіщення, а Представлення зробить запит на оновлення значення властивості з Моделі Представлення;

– модель Представлення (ViewModel): модель Представлення є абстракцією Представлення та містить обгортку даних із Моделі, яку необхідно прив'язати. Вона містить в собі дані, які будуть відображені на екрані, і логіку,

яка дозволяє взаємодіяти з цими даними. Також Модель Представлення містить команди, які може виконати Представлення для зміни Моделі.

За допомогою шаблону MVVM забезпечується розділення логіки додатку на рівні даних (Модель), представлення (Представлення) та логіки представлення (Модель Представлення), що сприяє покращенню підтримуваності та розширюваності коду.

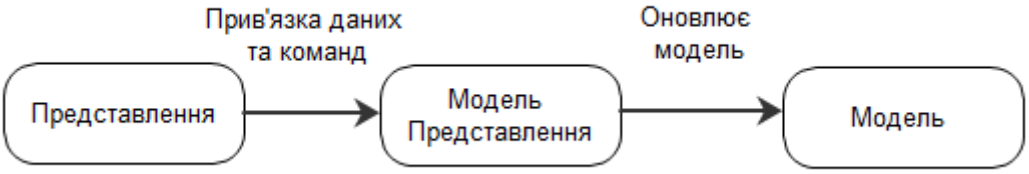

Рисунок 3.13 – Шаблон MVVM

У нашому виконанні Модель виступає як шар бізнес-логіки, Представлення - це XAML розмітка вікна або користувацького компонента (UserControl), а Модель Представлення - це зв'язувальна ланка між ними. Для підтримки шаблону MVVM та забезпечення прив'язки даних використовувалася бібліотека MVVMLight. Вона має достатній набір інструментів, що потрібні для реалізації шаблону MVVM. Для забезпечення комунікації між ViewModel використовувалися повідомлення. З одного боку, є видавець - той, хто створює повідомлення, а з іншого - споживач - той, хто отримує повідомлення того типу, на який він підписався. Для реалізації цієї схеми використовувався Messenger, який надається бібліотекою MVVMLight. Він має два основні методи: Register та Send. Перший реєструє підписку повідомлень з вказанням їх типу, другий відправляє повідомлення.

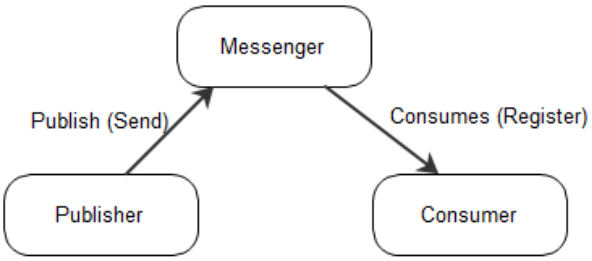

Рисунок 3.14 – Схема роботи Messenger

Для забезпечення зміни графічних форм був розроблений власний навігаційний модуль, що складається з чотирьох класів (див. рисунок 3.15):

– NavigationViewResolver – його завданням є реєстрація (метод "Register") графічної форми й, за необхідності, її контексту даних, а також надання її при запиті (метод "Resolve");

– NavigationController – він відповідає за зміну графічних форм у NavigationControl та, за потреби, контексту даних;

– NavigationControl – його функція - показувати користувачу графічні форми відповідно до вказівок NavigationController;

– NavigationService – цей клас надає інтерфейс для зміни контексту відображення, зміни графічних форм і можливості повернення до попередньої форми. Вихідний код наведено у додатку В.

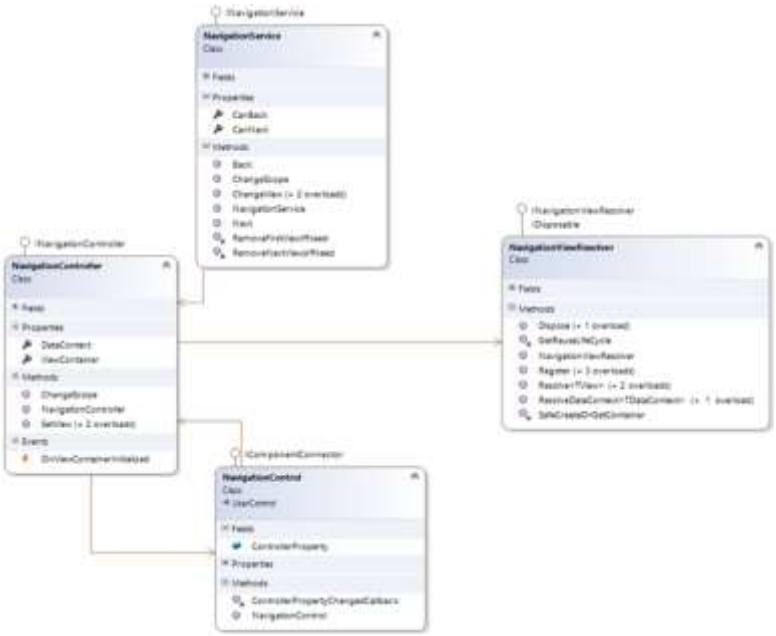

Рисунок 3.15 – Діаграма класів навігаційного модуля

Контекст відображення визначає, які екземпляри форм та їх контекст даних віддаються. Наприклад, у нашому додатку користувач може брати участь у декількох проектах з різними ролями і, відповідно, різною інформацією, що повинна відображатися. Таким чином, у нашому випадку контекстом відображення є ідентифікатор проекту.

Для реєстрації графічної форми та її контексту даних був використаний IoC-контейнер DryIoc, який підтримує три стратегії створення об'єктів. Цей контейнер також використовувався для співставлення абстракцій з їх реалізаціями, наприклад, IAccountManager та AccountManager.

Графічний інтерфейс був реалізований у стилі «матеріальний дизайн», розробленому компанією Google. Цей стиль є популярним і використовується у таких відомих додатках, як Google Play, Gmail, YouTube, Telegram, ВКонтакте та інших.

Для реалізації сервісу сповіщень використовувався ASP.NET SignalR, який забезпечує двонаправлений зв'язок між клієнтом та сервером через WebSocket, що дозволяє сервісу надсилати сповіщення клієнту в будь-який момент.

Комунікація між сервером та сервісами здійснювалася за допомогою RabbitMQ - платформи, яка реалізує систему обміну повідомленнями між компонентами програмної системи на основі стандарту AMQP. Важливо відзначити, що RabbitMQ має вбудовані можливості для горизонтального масштабування, наприклад, створення декількох черг чи підключення більшої кількості споживачів на один тип повідомлення.

Для розробки системи використовувалось середовище Microsoft Visual Studio. Це інтегроване середовище розробки, створене компанією Microsoft, надає інструменти для створення різних типів програм, таких як консольні, графічні (WPF, Windows Forms), веб-сайти та веб-додатки (ASP.NET, ASP.NET Core), з використанням різних мов програмування. Останні тенденції у хмарних обчисленнях дозволяють Visual Studio інтегруватися з Microsoft Azure.

На рисунку 3.16 показано структуру модулів у Visual Studio. Усі проекти логічно розділені на Client (клієнтська частина), Middleware (загальна частина), Server (серверна частина) та Services (мікросервісна частина).

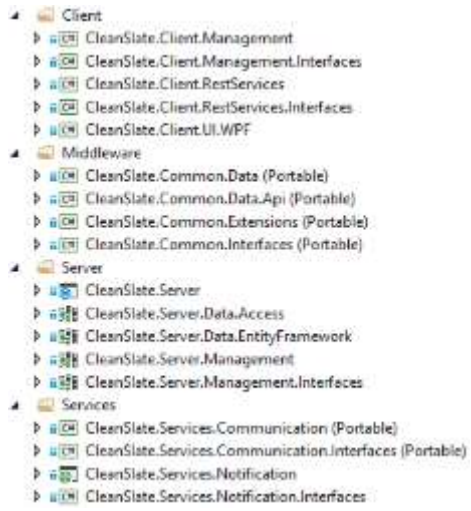

Рисунок 3.16 – Структура проектів

Client містить у собі п’ять проектів:

– CleanSlate.Client.RestServices.Interfaces – містить у собі абстракції для клієнтського шару доступу до даних: IRestClient, ICleanSlateRestClient, IRestClientFactory та абстракції сервісів, наприклад, IAccountService;

– CleanSlate.Client.RestServices – містить у собі реалізації усіх абстракцій для для клієнтського шару доступа до даних;

– CleanSlate.Client.Management.Interfaces – містить у собі абстракції для клієнтського шару бізнес-логіки, наприклад, IAccountManager;

– CleanSlate.Client.Management – містить у собі реалізації усіх абстракцій для клієнтського шару бізнес-логіки;

– CleanSlate.Client.UI.WPF – містить у собі реалізацію клієнтського шару презентації з використанням WPF.

Middleware містить у собі чотири проекти:

– CleanSlate.Common.Data – містить у собі модельні класи, які є об’єктним представленням реляційних даних;

– CleanSlate.Common.Data.Api – містить у собі модельні класи, які використовуються для взаємодії з Web Api;

– CleanSlate.Common.Extensions – містить у собі класи, які надають методи розширення («Extension Methods») для вбудованих типів;

– CleanSlate.Common.Interfaces – містить у собі абстракції, які можуть використовуватися як для клієнтської так і для серверної частин.

Server містить у собі п’ять проектів:

– CleanSlate.Server.Data.Access – містить у собі абстракції для серверного шару доступу до даних: IRepository, IUnitOfWork та дочірні абстракції IRepository для кожної сутності яка є об’єктним представленням реляційних даних, наприклад, IUserRepository;

– CleanSlate.Server.Data.EntityFramework – містить у собі реалізації абстракцій для серверного шара доступу до даних з використанням EF Core;

– CleanSlate.Server.Management.Interfaces – містить у собі абстракції для серверного шару бізнес-логіки, наприклад, IAccountManager;

– CleanSlate.Server.Management – містить у собі реалізації усіх абстракцій для серверного шару бізнес-логіки;

– CleanSlate.Server – містить у собі реалізацію серверного шару презентації з використання ASP .NET Core MVC.

Services містить у собі чотири проекти:

– CleanSlate.Services.Communication.Interfaces – містить у собі абстракції для комунікації між сервером та сервісами;

– CleanSlate.Services.Communication – містить у собі реалізацію абстракцій для комунікації між сервером та сервісами;

– CleanSlate.Services.Notification.Interfaces – містить у собі абстракції для сервісу сповіщень;

– CleanSlate.Services.Notification – містить у собі реалізації абстракцій сервіса сповіщень з використання ASP .NET SignalR.

При розробці системи були застосовані такі підходи та принципи: ООП, SOLID, DRY, KISS, YAGNI, а також coding guidelines, рекомендовані Microsoft [30].

Об’єктно-орієнтоване програмування (ООП) – це методологія програмування, яка представляє програму як сукупність об'єктів, кожен з яких є екземпляром певного класу, а класи формують ієрархії наслідування. Основні парадигми ООП включають:

– інкапсуляція – об'єднання даних та методів, які працюють з ними, в одному класі, приховуючи деталі реалізації від користувача;

– наслідування – можливість створення нового класу на основі існуючого з частковою або повною функціональністю. Базовий клас називається суперкласом, а новий клас – нащадком;

– поліморфізм – здатність системи використовувати об'єкти з однаковими інтерфейсами без знання їх типу та внутрішньої структури.

SOLID – акронім, що складається з перших літер принципів, спрямованих на створення гнучкого та адаптивного програмного коду:

– Single Resposibility Principle (Принцим єдиного обов’язку) – кожен клас має виконувати лише одну задачу і мати одну причину для змін;

– Open/Closed Principle (Принцип відкритості/закритості) – програмні сутності повинні бути відкриті для розширення, але закриті для зміни. Нові функціональні можливості додаються шляхом розширення існуючого коду, а не його зміни;

– Liskov Substitution Principle (Принцип підстановки Лісков) – об'єкти базового класу повинні мати можливість заміщатися об'єктами його підкласів без порушення роботи програми;

– Interface Segregation Principle (Принцип поділу інтерфейсів) – клієнти не повинні залежати від методів, якими вони не користуються;

– Dependency Inversion Principle (Принцип інверсій залежностей) – модулі верхнього рівня не повинні залежати від модулів нижнього рівня; обидва повинні залежати від абстракцій.

DRY (Don't Repeat Yourself) – принцип, який закликає уникати повторення одного і того ж коду.

KISS (Keep It Simple, Stupid) – принцип, що рекомендує робити архітектуру максимально простою та зрозумілою, використовуючи шаблони проєктування і уникаючи зайвої складності.

YAGNI (You Ain’t Gonna Need It) – принцип, який закликає реалізовувати лише необхідні функціональні можливості і уникати надлишкового функціоналу.

При розробці використовувалися керівні принципи програмування, рекомендовані Microsoft. Наприклад, використовувався camelCase для імен змінних, всі інтерфейси починаються з префіксу "I", а назви властивостей класу є іменниками.

3.4 Тестування програмної системи

У даному розглянуто основні функції системи такі як реєстрація, автентифікація, створення та підтвердження запрошень, створення проекту, створення та перегляд вимог.

Для початку роботи з системою, користувач повинен автентифікуватися у системі. На рисунку 3.17 зображено форму автентифікації користувача.

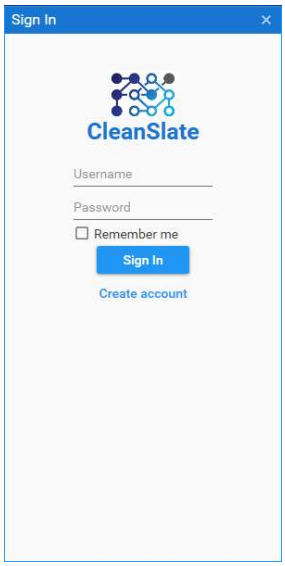

Рисунок 3.17 – Графічна форма автентифікації користувача

Якщо користувач не зареєстрований у системі, він може перейти до форми реєстрації (рисунок 3.18), натиснувши кнопку «Create account», де потрібно ввести необхідні поля для реєстрації.

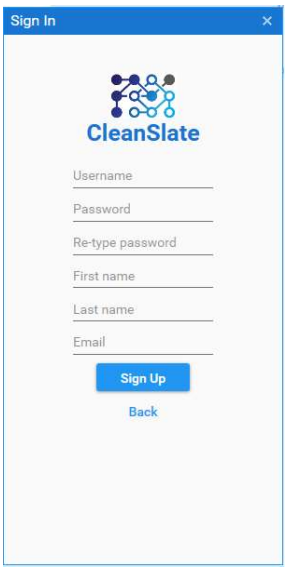

Рисунок 3.18 – Графічна форма реєстрації користувача

Після автентифікації відкривається головна форма (рисунок 3.19).

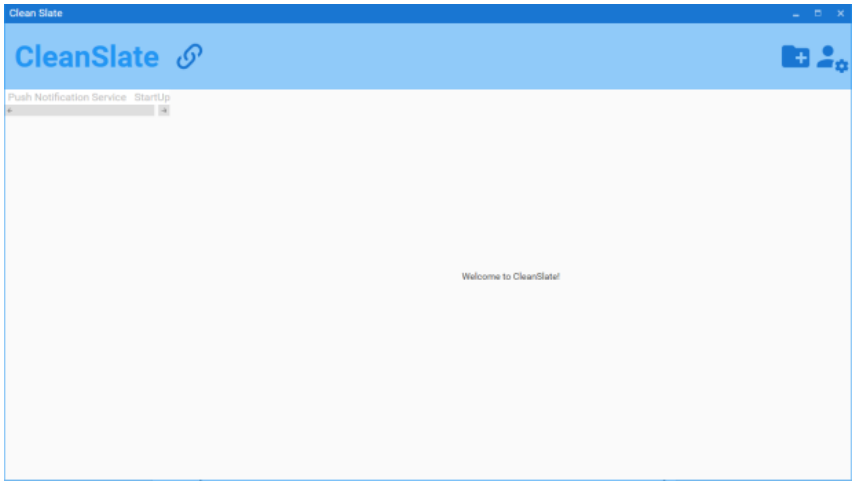

Рисунок 3.19 – Головна форма

Користувач має можливість створити новий проект (рисунок 3.20).

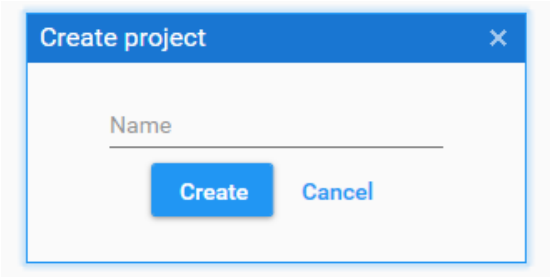

Рисунок 3.20 – Створення нового проекту

На рисунку 3.21 зображено форму для створення запрошення для іншого користувача до проекту.

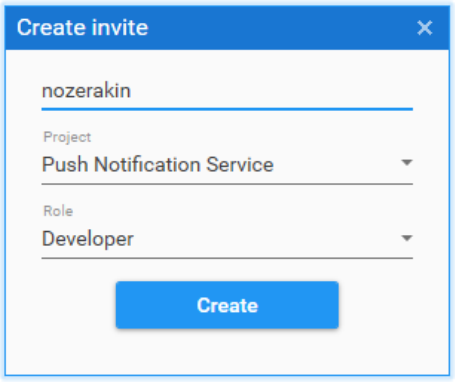

Рисунок 3.21 – Форма створення запрошення

Користувач може переглянути вхідні запрошення та відправлені запрошення на вкладках «Incoming» та «Outgoing», відповідно (рисунок 3.22).

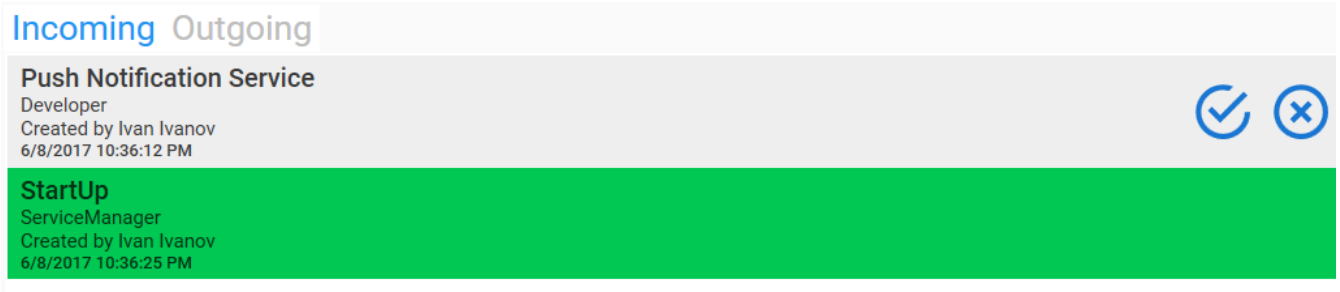

Рисунок 3.22 – Графічна форма з запрошеннями

На рисунку 3.23 зображено графічну форму для створення вимог.

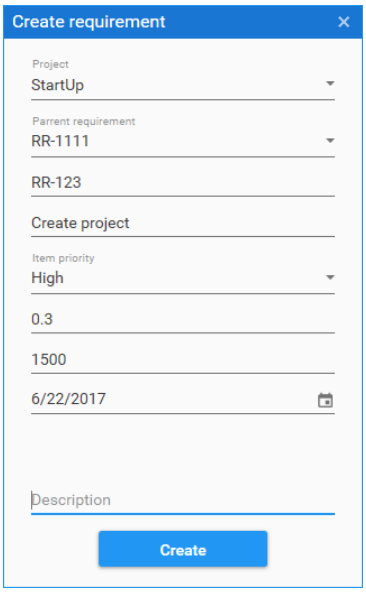

Рисунок 3.23 – Створення нової вимоги

Усі вимоги можна переглянути натиснувши кнопку «Requirements». Вимоги будуються у древоподібній структурі, натиснувши на елемент у дереві, можна детально переглянути саму вимогу (рисунок 3.24).

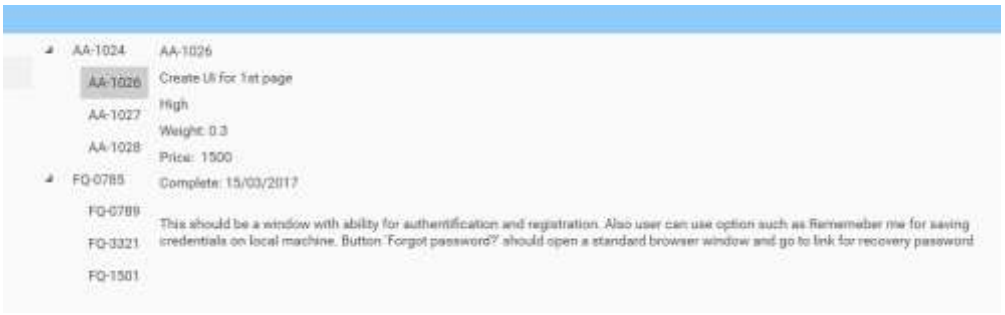

Рисунок 3.24 – Перегляд вимог

Під час мануального тестування було перевірено основні функції системи. Нижче наведено перелік випробувань та отриманий результат.

Таблиця 3.1

Тестовий сценарій «Автентифікація у системі»

Назва	Тест для перевірки графічної форми автентифікації у системі	
Use case	Автентифікація користувача у системі	
Дія	Очікуваний результат	Результат тесту
Передумова		
Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Кроки тесту		
Заповнити поля такими значеннями: «Username» : CustomerUsr «Password» : Qwerty123	Немає помилок про невірні дані. Кнопка «Sign In» доступна.	Пройдений
Натиснути кнопку «Sign In»	Користувач автентифікований. Відкрилася головна графічна форма	Пройдений

Таблиця 3.2

Тестовий сценарій «Реєстрація користувача у системі»

Назва	Тест для перевірки графічної форми автентифікації у системі	
Use case	Реєстрація нового користувача у системі	
Дія	Очікуваний результат	Результат тесту
Передумова		

Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Натиснути кнопку «Create Account»	Форма реєстрації відкрита. Усі текстові поля видимі.	Пройдений
Заповнити поля наступними значеннями: «Username» : CustomerUsr «Password» : Qwerty123 «Re-type password» : Qwerty123, «First name» : John «Last name» : Smith «Email» : john.sth@e.com	Немає помилок про невірні дані. Кнопка «Sign Up» доступна.	Пройдений
Натиснути кнопку «Sign Up»	Користувач зареєстрований. Відкрилася форма про успішну реєстрацію користувача	Пройдений

Таблиця 3.3

Тестовий сценарій «Створення нового проекту»

Назва	Тест для перевірки графічної форми створення нового проекту	
Use case	Створення нового проекту у системі	
Дія	Очікуваний результат	Результат тесту
Передумова		
Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Заповнити поля такими значеннями: «Username» : CustomerUsr «Password» : Qwerty123	Немає помилок про невірні дані. Кнопка «Sign In» доступна.	Пройдений
Натиснути кнопку «Sign In»	Користувач автентифікований. Відкрилася головна графічна форма	Пройдений
Натиснути кнопку створення нового проекту	Відкрилася форма створення нового проекту. Усі текстові поля доступні	Пройдений
Заповнити поля такими значеннями: «Name» : StartUp	Немає помилок про невірні дані. Кнопка «Create» доступна.	Пройдений
Натиснути кнопку «Create»	Створено новий проект. Закрилася графічна форма створення проекту. На головній формі, зліва, оновився список проектів	Пройдений

Таблиця 3.4

Тестовий сценарій «Створення запрошення»

Назва	Тест для перевірки графічної форми створення запрошення	
Use case	Створення запрошення	
Дія	Очікуваний результат	Результат тесту
Передумова		
Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Заповнити поля такими значеннями: «Username» : CustomerUsr «Password» : Qwerty123	Немає помилок про невірні дані. Кнопка «Sign In» доступна.	Пройдений
Натиснути кнопку «Sign In»	Користувач автентифікований. Відкрилася головна графічна форма	Пройдений
Натиснути кнопку створення запрошення	Відкрилася форма створення запрошення. Усі текстові поля доступні	Пройдений
Заповнити поля такими значеннями: «Username» : ivanovdev «Project» : StartUp «Role» : Developer	Немає помилок про невірні дані. Кнопка «Create» доступна.	Пройдений
Натиснути кнопку «Create»	Створено запрошення. Закрилася графічна форма створення запрошення. На головній формі, у спику вихідних запрошень, оновився список запрошень	Пройдений

Таблиця 3.5

Тестовий сценарій «Підтвердження запрошення»

Назва	Тест для перевірки графічної форми підтвердження запрошення	
Use case	Підтвердження запрошення	
Дія	Очікуваний результат	Результат тесту
Передумова		
Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Заповнити поля такими значеннями: «Username» : ivanovdev «Password» : Qwerty123	Немає помилок про невірні дані. Кнопка «Sign In» доступна.	Пройдений

Натиснути кнопку «Sign In»	Користувач автентифікований. Відкрилася головна графічна форма	Пройдений
Кроки тесту		
Натиснути кнопку перегляду запрошень	Відкрилася форма зі списком вхідних запрошень.	Пройдений
Натиснути кнопку підтвердження запрошення	Запрошення у списку змінило колір на зелений, що свідчить про успішне підтвердження запрошення. На головній формі, зліва, оновився список проектів.	Пройдений

Таблиця 3.6

Тестовий сценарій «Створення вимоги»

Назва	Тест для перевірки графічної форми створення вимоги	
Use case	Створення вимоги	
Дія	Очікуваний результат	Результат тесту
Передумова		
Запустити головний виконавчий файл	Форма автентифікації відкрита	Пройдений
Заповнити поля такими значеннями: «Username» : CustomerUsr «Password» : Qwerty123	Немає помилок про невірні дані. Кнопка «Sign In» доступна.	Пройдений
Натиснути кнопку «Sign In»	Користувач автентифікований. Відкрилася головна графічна форма	Пройдений
Натиснути кнопку створення нової вимоги	Відкрилася графічна форма для створення нової вимоги. Усі текстові поля доступні	Пройдений
Заповнити поля такими значеннями: «Project» : Startup «Code name» : FR-10 «Name» : Create UI «Item priority» : High «Weight» : 0.3 «Price» : 1500 «Description» : Create user interface for register new user	Немає помилок про невірні дані. Кнопка «Create» доступна.	Пройдений

Натиснути кнопку «Create»	Створилася нова вимога. Графічна форма для створення вимог закрилася. У дереві вимог з’явилася створена вимога	Пройдений

Модульні тести (Unit-тести) дозволяють швидко та автоматично перевіряти окремі частини коду незалежно від решти програми. При правильному підході, модульні тести можуть покрити більшу частину коду додатка. Основні характеристики модульних тестів включають тестування невеликих фрагментів коду ("юнітів"), тестування в ізоляції, тестування лише загальнодоступних кінцевих точок та автоматизацію тестування. Розглянемо ці характеристики детальніше.

При створенні модульних тестів обираються невеликі частини коду, які потрібно перевірити. Зазвичай це менше класу, а в більшості випадків тестується окремий метод класу. Це забезпечує швидкість написання модульних тестів.

Ізоляція тестованого коду від іншої програми є важливою для чіткого визначення можливих помилок саме в цьому фрагменті. Це підвищує контроль над окремими компонентами програми.

Невеликі зміни в класі можуть призвести до провалу багатьох модульних тестів через зміну реалізації. Щоб уникнути цього, модульні тести обмежуються лише загальнодоступними кінцевими точками, що дозволяє ізолювати їх від деталей внутрішньої реалізації.

Оскільки модульні тести охоплюють невеликі фрагменти коду, їхня кількість може бути значною. Без автоматизації процесу отримання результатів і виконання тестів це може знизити продуктивність розробки. Тому важливо, щоб модульні тести були простими у використанні та забезпечували чітку інформацію про результати. Для автоматизації розробники зазвичай використовують готові фреймворки.

Розробка через тестування (Test-Driven Development, TDD) – це підхід, при якому спочатку пишуться модульні тести, а потім код, достатній для їх проходження. Використання TDD знижує кількість потенційних помилок у

програмі. Створюючи модульні тести перед написанням коду, ми описуємо поведінку майбутніх компонентів, не обмежуючись конкретною реалізацією. Таким чином, тести допомагають визначити API майбутніх компонентів.Моделі тестів Arrange-Act-Assert являє собою не лише особливість тестування в Visual Studio, а й цілу парадигму тестування:

– arrange – підготовка середовища, в якому виконується код;
– act – тестування коду
– assert – переконуємося, що результат тесту саме той, який ми очікували.

Під час тестування часто використовуються фіктивні об’єкти (mock-об’єкти). Mock-об’єкт реалізує задані аспекти програмного середовища, яке моделюється, і використовується виключно для тестування взаємодії.

Для написання модульних тестів ми обрали фреймворк NUnit. Класи, призначені для юніт-тестів, позначають атрибутом TestFixtureAttribute, а методи тестів – атрибутом TestAttribute. Метод для базових налаштувань середовища позначають атрибутом SetupAttribute і зазвичай називають Setup. На рисунку 3.25 наведено результат виконання тестів для RequirementController.

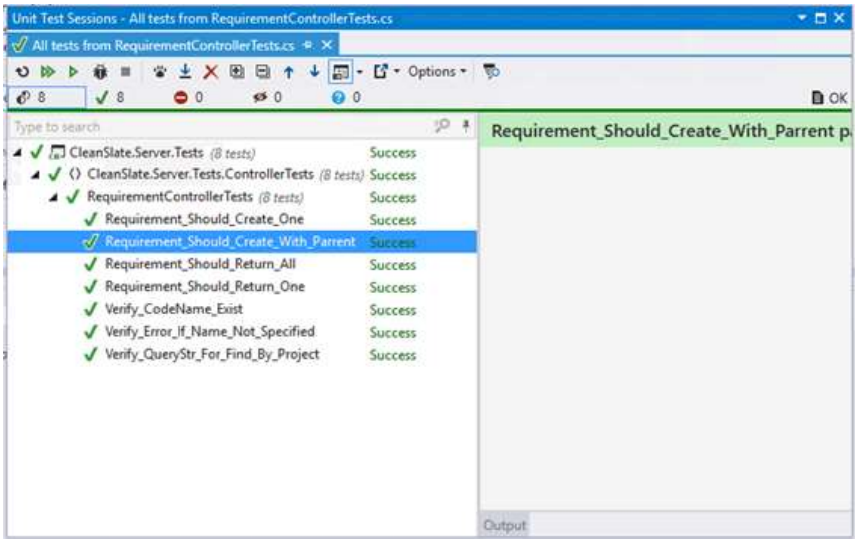

Рисунок 3.25 Результат виконання модульних тестів

4 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ З ВИКОРИСТАННЯМ ОБРАНИХ ТЕХНОЛОГІЙ

4.1 Методика оцінювання ефективності

Методика оцінювання ефективності засобів і технологій міжсервісного зв’язку полягає у наступній послідовності дій:

– визначення критеріїв оцінювання ефективності засобів і технологій міжсервісного зв’язку;

– проведення моделювання продуктивності для обраних технологій на основі визначених критеріїв;

– розробка рекомендацій щодо використання розглянутих технологій міжсервісного зв’язку;

– вибір технології на основі значень найбільш вагомих критеріїв для системи.

В результаті дослідження засобів і технологій міжсервісної комунікації визначається перелік основних критеріїв, за якими їх можна оцінювати. На рисунку 4.1 наочно показано відповідність кожної з технологій обраним критеріям, враховуючи відсутність додаткових втручань у реалізацію. Це дозволяє виділити найкращі технології, спираючись на задоволення обраним критеріям.

Відповідність критеріям встановлюється в залежності від того, чи підтримуються вони технологією без додаткових зусиль з боку розробки та проектування. Кожен з критеріїв може мати різну вагу при виборі технології в залежності від вимог до системи. Наприклад, якщо одним з вимог до системи є жорстка стандартизація, а продуктивність не є першочерговою вимогою, то використання XML є прийнятним. У такому випадку SOAP буде найкращим кандидатом, оскільки пропонує структурований спосіб відправки повідомлень між службами зі строгими правилами, що визначають структуру повідомлень і способи їх обробки.

	REST	GraphQL	SOAP	Apache Thrift	gRPC
Мультиплатформність	+	+	+	—	—
Незалежність від формату даних	+	+	—	+	—
Стандартизація	—	—	+	—	—
Когнітивна зрозумілість	+	+	—	+	+
Простота використання	+	—	—	—	+
Розповсюдженість	+	+	—	—	+
Версіонування	+	—	+	+	—
Кешування	+	—	+	—	—

Рисунок 4.1 — Задовільнення технологій обраним критеріям

Якщо продуктивність є важливою вимогою, SOAP не слід обирати, оскільки він прив'язаний до використання XML, який має найгіршу продуктивність серед усіх розглянутих форматів серіалізації. Високу продуктивність забезпечують RPC технології, такі як Apache Thrift і gRPC. Обидві технології прості у використанні, але gRPC дозволяє використовувати Protocol Buffers не тільки як формат серіалізації та десеріалізації, але й як мову визначення інтерфейсу. Проте Apache Thrift не підтримує стрімінг великих обсягів даних. Вибір між gRPC і Apache Thrift залежить від необхідності використання Protocol Buffers та роботи з великими обсягами даних. Незважаючи на високу продуктивність, RPC протоколи мають недоліки, зокрема тісну зв'язаність і слабку когнітивну зрозумілість.

Якщо важливі зрозумілість використання та мультиплатформеність, REST може бути використаний для встановлення міжсервісного зв’язку. REST став однією з найбільш популярних альтернатив комунікації між сервісами завдяки своїй легкості та підтримці будь-якою мовою програмування. GraphQL пропонує новий спосіб запиту даних у форматі, який легко зрозуміти. Він дозволяє передавати лише необхідні дані, що зменшує обсяг трафіку.

GraphQL представляє ресурси даних як вузли в графі, де будь-які вузли можуть бути об'єднані для формування запиту. Він був розроблений для об'єднання даних в єдиний сервіс і добре відповідає шаблону API шлюзу (API gateway), але додає небажаних залежностей до системи, оскільки вимагає структури для розбору запитів і формування відповідей. Якщо фреймворк GraphQL зможе аналізувати і серіалізувати дані без значних втрат продуктивності порівняно з архітектурою REST із використанням JSON як формату серіалізації та десеріалізації, він може бути життєздатною альтернативою.

4.2 Моделювання продуктивності технологій міжсервісного зв’язку

На основі аналізу відповідності технологій критеріям, наведених на рисунку 4.1, можна зробити висновок, що найкращими для використання є GraphQL та REST. Моделювання продуктивності цих технологій виконується за допомогою розробленої програмної системи. Для порівняння продуктивності обраних засобів комунікації необхідно провести моделювання, яке включає набір тестів з різними обсягами даних [22].

Перший тест полягає в отриманні невеликої кількості даних від кожного сервісу через інтерфейси REST і GraphQL. Таблиці містять по 5 записів. Затримка під час отримання даних від системи, розміщеної в хмарному середовищі, може значно варіюватися від запиту до запиту. Для отримання достовірних результатів було надіслано 100 запитів поспіль із секундною затримкою між ними. Затримка впроваджена для мінімізації впливу навантаження на систему на результати. Одна секунда обрана як порогове значення для часу виконання запиту. Час відповіді вимірюється як загальний час перебування запиту в системі, включаючи обробку запиту мікросервісом та обробку даних з бази даних. Час відповіді для кожного запиту до кожного з інтерфейсів зображено на рисунку 4.2.

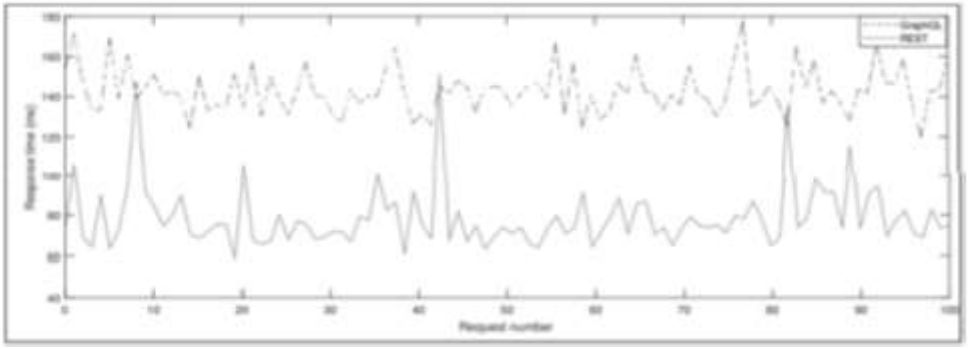

Рисунок 4.2 — Час відповіді при запиті даних за допомого GraphQL та REST інтерфейсів

По горизонтальній осі розташовані запити у порядку їхнього виконання, а по вертикальній осі – час виконання кожного запиту у мілісекундах. Суцільна лінія відображає запити REST, а пунктирна – запити GraphQL. Комунікація через REST показала менший час відповіді порівняно з GraphQL. Найшвидший час відповіді для REST становив 58,53 мс, а найповільніший – 151,08 мс. Взаємодія за допомогою GraphQL продемонструвала гірші результати: найшвидший час відгуку склав 119,86 мс, а найповільніший – 177,72 мс.

Порівняння середнього часу відгуку для REST і GraphQL показано на рисунку 4.3.

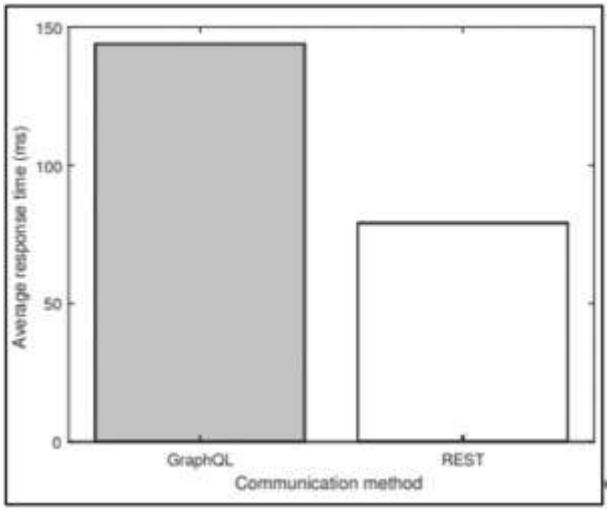

Рисунок 4.3 — Порівняння середнього часу відгуку для GraphQL та REST

По горизонтальній осі розміщені методи взаємодії, а по вертикальній осі – відповідний середній час виконання запиту. REST показав середній час відгуку 78,94 мс, тоді як GraphQL має середнє значення 142,92 мс, що робить GraphQL на 81,05% повільнішим в середньому.

Другий тестовий випадок використовує те саме тестове оточення, що й попередній, але в цьому випадку вимірювалась продуктивність при обробці запитів з більшим обсягом даних. Кожна з чотирьох таблиць, до яких звертаються мікросервіси, розширена до 25 записів.

Виміряний час відгуку в цьому тесті показав, що для GraphQL затримка помітно зростає зі збільшенням обсягу даних. Заміряний час для 100 виконаних запитів з використанням кожного з інтерфейсів, що брали участь у моделюванні, зображено на рисунку 4.4.

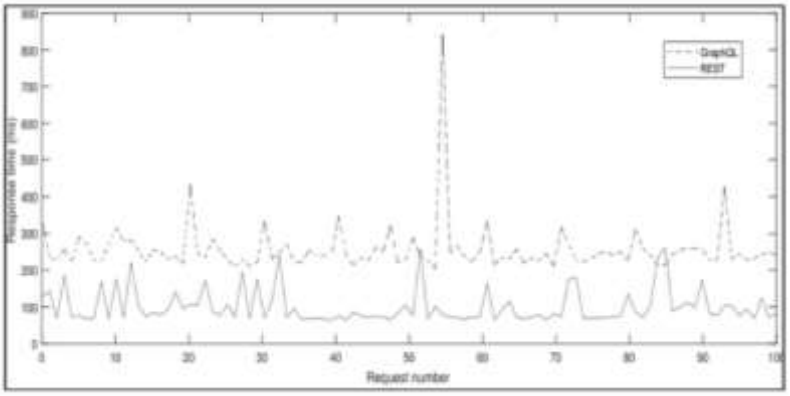

Рисунок 4.4 — Час відповіді при запиті даних для другого тестового випадку

Найшвидший час відповіді при використанні REST становив 63,04 мс, а найповільніший – 260,39 мс. У випадку GraphQL найшвидший час відгуку склав 203,00 мс, а найповільніший – 840,56 мс. Порівняння середнього часу відгуку при виконанні запитів для REST і GraphQL показано на рисунку 4.5.

Отримання даних через інтерфейс REST зайняло в середньому 100,35 мс, тоді як вибірка даних за допомогою GraphQL в середньому склала 256,80 мс. Таким чином, отримання даних за допомогою GraphQL у другому тестовому випадку було в середньому на 256% повільніше, ніж вибірка даних з використанням REST.

Для третього тестового випадку кількість тестових даних збільшено до 50 записів у кожній з таблиць, до яких звертаються мікросервіси. Процедура виконання викликів та оточення залишаються такими ж, як і в попередніх сценаріях моделювання.

.

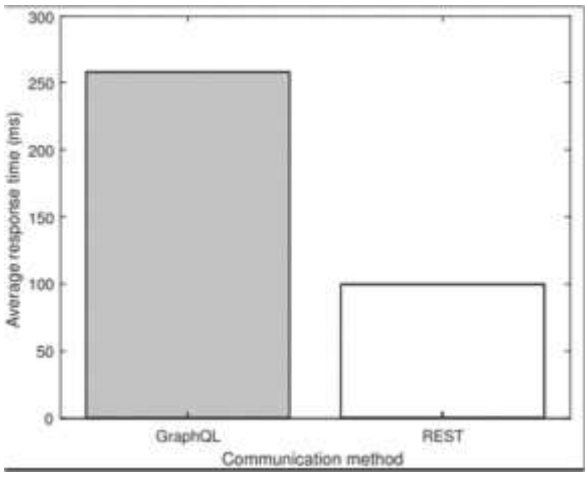

Рисунок 4.5 — Порівняння середнього часу відгуку GraphQL та REST для другого тестового випадку

Результати запитів для третього тестового випадку доводять, що фреймворк GraphQL не може конкурувати з архітектурою REST за продуктивністю, як і у попередніх сценаріях. Час відповіді для GraphQL та REST відображений на рисунку 4.6.

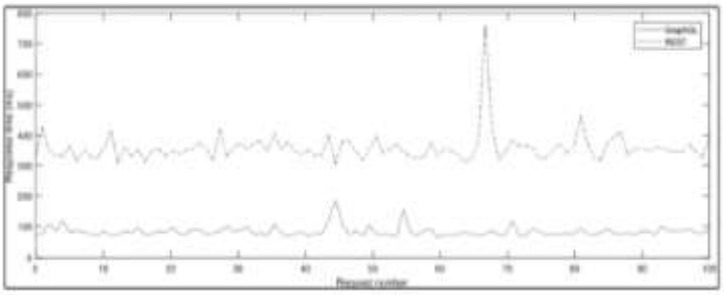

Рисунок 4.6 — Час відповіді при запиті даних для третього тестового випадку

Для REST мінімальний час відповіді становить 65,25 мс, максимальний — 185,94 мс. Час для GraphQL значно гірший: мінімальна затримка — 301,14 мс, максимальна затримка — 759,84 мс. Порівняння середнього часу відгуку для REST і GraphQL у третьому тестовому випадку показано на рисунку 4.7.

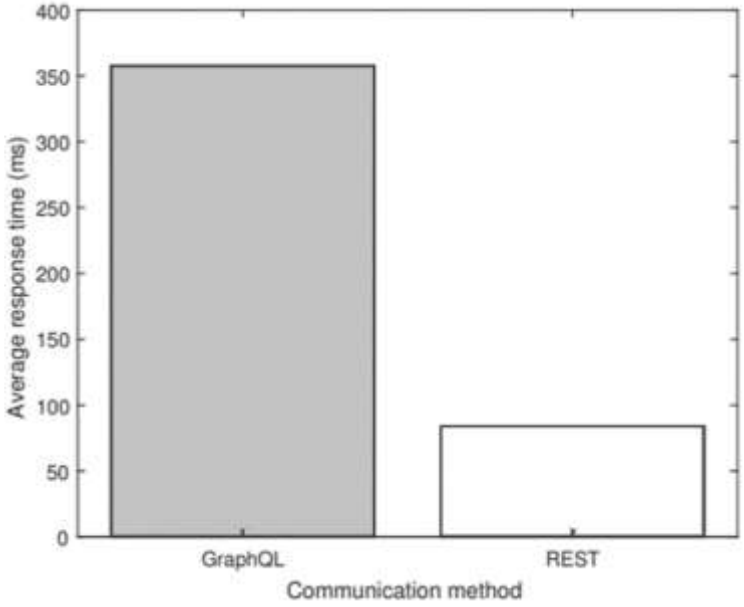

Рисунок 4.7 — Порівняння середнього часу відгуку для третього тестового випадку

Помітне зниження продуктивності при використанні REST у цьому тестовому випадку порівняно з другим тестовим випадком, однак середній час відповіді REST фактично швидший, ніж у другому тестовому випадку — 84 мс. Середній час відповіді для GraphQL у цьому тесті склав 357,89 мс. Таким чином, середній час відповіді для GraphQL на 426% більший, ніж у REST, та на 179% більший порівняно з тестом 2.

4.3 Рекомендації по використанню технологій міжсервісного зв’язку

Результати моделювання продуктивності показують, що GraphQL поступається архітектурі REST з використанням JSON у плані часу відгуку. Навіть при отриманні невеликого обсягу даних, обробка запитів через GraphQL викликає збільшення часу відповіді. Оскільки GraphQL використовує JSON як формат повідомлень, його можливості серіалізації обмежені. GraphQL пропонує простіший інтерфейс для вибору необхідних даних, включаючи у відповідь лише запитані поля.

Проблема REST інтерфейсу полягає в тому, що кожен ресурс відображається на окремий URI, і для кожного випадку, коли різні клієнти потребують специфічної структури даних, необхідно створювати новий REST

ендпоінт. Натомість, при використанні GraphQL, потрібно лише створити детальну структуру, що описує наявні типи і поля. Кожен клієнт може точно вказати, які поля він бажає отримати, що усуває великі інтерфейси, які складно підтримувати та обробляти у різних версіях одного API.

При запиті невеликого обсягу даних, GraphQL є гарним варіантом. Однак, якщо продуктивність є ключовим критерієм, GraphQL не є найкращим вибором. Архітектура REST може забезпечити кращу продуктивність завдяки меншому часу відгуку та розміру повідомлень, особливо при використанні двійкового протоколу серіалізації замість JSON. Проте, використання RPC фреймворків, таких як gRPC або Thrift, та бінарної серіалізації в архітектурі мікросервісів обмежує кросплатформову сумісність мікросервісів.

Вибір технології для взаємодії в мікросервісній архітектурі є компромісом між обмеженням використання конкретної мови та інструментів, ефективністю комунікації та складністю реалізації.

ВИСНОВКИ

У кваліфікаційній роботі було спроектовано систему для підтримки життєвого циклу розробки ПЗ та створено підсистему для взаємодії замовників і розробників. Було розглянуто сім методологій: каскадна, інкрементальна, Agile, RAD, спіральна, RUP, XP, із зазначенням їхніх переваг та недоліків. Порівняльний аналіз допоміг розробити власну концепцію розробки ПЗ – концепцію СПЗ. Визначено переваги та недоліки різних технологій для взаємодії мікросервісів, що дозволило виявити найефективніші рішення для конкретних завдань. Проведено аналіз існуючих систем і виявлено їхні недоліки, що впливають на ефективність та надійність мікросервісної архітектури.

Було спроектовано систему, яка підтримує життєвий цикл СПЗ. Вона пропонує дві версії: публічну та корпоративну. Завдяки використанню трьохшарової та мікросервісної архітектури, система легко масштабується горизонтально. Розглянуто переваги використання хмарних обчислень, зокрема Microsoft Azure. Система також передбачає можливість набуття юридичної сили, що допоможе відмовитись від зайвих паперів при обговоренні термінів, умов виконання та оплати.

Було реалізовано підсистему для взаємодії замовників і розробників під час створення ПЗ, а також "Аукціон вимог", що є частиною концепції СПЗ. Для реалізації використовувалися сучасні технології програмування. Створено кросплатформений сервер, що надає широкий вибір ОС для розгортання. Він розроблений з використанням найновішого фреймворку від Microsoft – ASP.NET Core. Дані зберігаються у PostgreSQL, оскільки він підтримує горизонтальне масштабування та є кросплатформеним. Для комунікації між мікросервісами обрано RabbitMQ. З використанням WebSocket було реалізовано сервіс сповіщень, який в режимі реального часу доставляє користувачу сповіщення про події, які відбулися. Для синхронної взаємодії між сервісами був використана REST архітектура на базі JSON.

На основі розробленого додатку виявлено, які технології найкраще підходять для вирішення конкретних завдань, та розроблено рекомендації щодо впровадження запропонованих методів у реальні системи для підвищення їх ефективності та надійності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бабаєв В.М. Прийняття рішень: Конспект лекцій. Харків: ХНАМГ, 2007. 185 с.

2. Верес О.М. Технології підтримання прийняття рішень: Навч. Посібник. Львів: Видавництво Національного університету “Львівська політехніка”, 2010. 252 с.

3. Види серверів і їх ролі. 2021. URL: https://www.myvin.com.ua/news/12371-vydy-serveriv-i-ikh-roli.

4. Катренко А.В. Системний аналіз: підручник. Львів: “Новий світ – 2000”, 2009. 396 с.

5. Катренко А.В. Теорія прийняття рішень: підручник з грифом МОН. Львів: “Магнолія 2006”, 2010. 352с.

6. Катренко А.В. Управління ІТ-проектами. [Книга 1. Стандарти, моделі та методи управління проектами]: [підручник]. Львів: «Новий Світ-2000», 2011. 550 с.

7. Колесников О. С., Войтенко О. Є., Харченко І. В. Комп'ютерні мережі і телекомунікації. Київ: Київський університет, 2016. 288 с.

8. Курковская Г. І., Рябко Б. Я., Силаєва О. С. Основи мережевих технологій. Київ: Видавничий дім «Слово», 2014. 368 с

9. Литвин В.В. Інтелектуальні системи. Львів: Новий Світ-2000, 2008. 408с.

10. Литвин В.В. Технології менеджменту знань: навч. посібник. Львів: Видавництво Львівської політехніки, 2010. 260 с.

11. Литвин В.В. Проектування інформаційних систем: навч. посібник. Львів: “Магнолія 2006”, 2010. 352с.

12. Пасічник В.В. Організація баз даних та знань. Київ: BHV „ПИТЕР”, 2006. 460с.

13. Сервери і системи зберігання даних. 2019. URL: http://www.itnt.net.ua/lang/ua/servers.html

14. Сервер – що це таке, як працює і навіщо потрібен. Види, функції та приклади.. 2022. URL: https://termin.in.ua/server/

15. Синкевич М., Лєсна Н. Дослідження засобів і технологій міжсервісного зв'язку в мікросервісній архітектурі. International Electronic Scientific Journal «Science Online», 2019

16. Топологія комп'ютерних мереж. URL: https://uk.wikipedia.org/wiki/Топологія_комп%27ютерних_мереж

17. Що таке сервер і для чого він потрібний. 2022. URL: https://blogchain.com.ua/shcho-take-server-i-dliachoho-vin-potribnyj/.

18. Al-Shaer, E., Hamed, H. S. Security Design of Network Topologies: Theory and Practice. CRC Press, 2015. 200 p.

19. Apache thrift documentation. URL: https://thrift.apache.org

20. Cisco Networking Academy. CCNA Routing and Switching: Introduction to Networks. Cisco Press, 2013. 720 p.

21. Comer, D. E. Computer Networks and Internets. Upper Saddle River, NJ, Pearson, 2014. 672 p.

22. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003. p. 560

23. Extensible markup language (xml) 1.0 (fifth edition). URL: https://www.w3.org/TR/REC-xml

24. GitHub - Netflix/zuul: Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more. URL: https://github.com/Netflix/zuul

25. GitHub - spring-cloud/spring-cloud-config: External configuration (server and client) for Spring Cloud. URL: https://github.com/spring-cloud/spring-cloud-config

26. Hartpence B. Network Topology and Its Engineering Considerations. CreateSpace Independent Publishing Platform, 2016. 174 p.

27. Hypertext Transfer Protocol Bis (httpbis). URL: https://datatracker.ietf.org/wg/httpbis/charter

28. Kami Makki S. A. Sumaray. A comparison of data serialization formats for optimal efficiency on a mobile platform. In ICUIMC ’12 Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication, 2012.

29. Kurose J. F., Ross, K. W. Computer Networking: A Top-Down Approach. Boston: Pearson, 2016. 864 p.

30. Lammle, T. CCNA Routing and Switching Study Guide: Exams 100-105, 200-105, and 200-125. Wiley, 2016. 1152 p.

31. Maeda K. Performance evaluation of object serialization libraries in xml, json and binary formats. In Digital Information and Communication Technology and it’s Applications (DICTAP), 2012 Second International Conference on, 2012.

32. Odom W. CCENT/CCNA ICND1 100-105 Official Cert Guide. Cisco Press, 2016. 1024 p.

33. Peterson, L. L., Davie, B. S. Computer Networks: A Systems Approach. San Francisco: Morgan Kaufmann, 2011. 920 p.

34. Popic et al. Performance evaluation of using protocol buffers in the internet of things communication. In 2016 International Conference on Smart Systems and Technologies (SST), 2016.

35. Protocol buffers documentation. URL: https://developers.google.com/ protocol-buffers

36. Richardson Maturity Model. URL: https://martinfowler.com/articles/richardsonMaturityModel.html

37. Spring Framework. URL: http://springprojects.ru/projects/spring-framework

38. Stallings, W. Data and Computer Communications. Upper Saddle River, NJ: Pearson, 2013. 912 p.

39. Белименко В.С. Дослідження та аналіз процесів взаємодії сервісів в мікро-сервісній архітектурі, 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 6., – Харків: ХНУРЕ. 2024. – 780 с.