БАКАЛАВРСЬКА РОБОТА

БР.ІП – 41.00.00.000 ПЗ

Група ІП-22-2

Матвієнко Іван-Юрій

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Матвієнко Іван-Юрій Андрійович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Інженерія програмних систем із використанням API-first підходу
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Матвієнко І.-Ю. А._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Гобир Лідія Мирославівна, асистент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Матвієнко Івану-Юрію Андрійовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Інженерія програмних систем із використанням API-first підходу"

керівник проекту (роботи) Гобир Лідія Мирославівна, асистент

затверджені наказом вищого навчального закладу від **“** 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Структура ітерацій (рис. 1.1, ст. 11)

2 Процес проектування мікросервісів (рис. 2.2, ст. 44)

3 Таксономія та супутні артефакти (рис. 1.3, ст. 18)

4 Динамічна візуальна документація API «Absences» у Swagger Editor (рис. 3.3, ст. 62)

5 Додавання нового API шляхом імпорту специфікації OpenAPI (рис. 3.6, ст. 64)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Матвієнко І-Ю.А.
(підпис)

Керівник роботи _____Гобир Л.М.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 79 сторінки, 32 рисунки, 1 таблиця, список використаних джерел із 40 найменувань.

Метою роботи є дослідження принципів інженерії програмних систем із використанням API-first підходу та аналіз методів побудови масштабованих, гнучких і інтегрованих програмних рішень.

Об'єкт дослідження: процеси розробки програмних систем із використанням API-first підходу.

Предмет дослідження: методи, моделі та інструменти проєктування, реалізації та тестування API, а також підходи до інтеграції програмних компонентів.

Результати дослідження: проведено аналіз концепції API-first, досліджено принципи API-орієнтованого проєктування, розглянуто сучасні інструменти та стандарти розробки API, а також реалізовано приклад програмної системи з використанням даного підходу.

У першому розділі розглянуто теоретичні основи API-first підходу, роль API у сучасній розробці програмного забезпечення та обґрунтовано доцільність застосування даної концепції.

У другому розділі досліджено методи та інструменти API-орієнтованого проєктування, підходи до визначення артефактів, побудови таксономій API.

У третьому розділі розглянуто практичну реалізацію API-first підходу, створення API та клієнтського застосунку, використання стандартів і інструментів.

Висновок: застосування API-first підходу забезпечує підвищення гнучкості, масштабованості та інтегрованості програмних систем, спрощує процес розробки та тестування.

КЛЮЧОВІ СЛОВА: API-FIRST, API, REST, МІКРОСЕРВІСИ, ІНТЕГРАЦІЯ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, OPENAPI, ПРОГРАМНА ІНЖЕНЕРІЯ.

ANNOTATION

The bachelor's thesis contains 79 pages, 32 figures, 1 tables, and a list of sources used with 40 names.

The aim of the work is to study the principles of software engineering using the API-first approach and to analyze methods for building scalable, flexible, and integrated software solutions.

Research object: processes of software system development using the API-first approach.

Research subject: methods, models, and tools for API design, implementation, and testing, as well as approaches to integrating software components.

Research results: the concept of API-first was analyzed, principles of API-oriented design were studied, modern tools and standards for API development were reviewed, and a sample software system using this approach was implemented.

The first section describes the theoretical foundations of the API-first approach, the role of APIs in modern software development, and the rationale for applying this concept.

The second section analyzes methods and tools for API-oriented design, approaches to defining artifacts, building API taxonomies, and analyzing user requirements and use cases.

The third section covers the practical implementation of the API-first approach, including API and client application development, use of standards and tools, and evaluation of automated testing effectiveness.

Conclusion: the use of the API-first approach improves flexibility, scalability, and integration of software systems, simplifies development and testing processes, and enhances interaction between system components.

KEY WORDS: API-FIRST, API, REST, MICROSERVICES, INTEGRATION, AUTOMATED TESTING, OPENAPI, SOFTWARE ENGINEERING.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у розвитку цифрової економіки, а програмні системи стають дедалі складнішими, розподіленими та інтегрованими. Зростання кількості веб- та мобільних застосунків, розвиток мікросервісної архітектури, а також необхідність швидкої взаємодії між різними системами обумовлюють важливість ефективних підходів до проєктування програмного забезпечення. Одним із таких підходів є API-first, який передбачає проєктування інтерфейсів програмування застосунків (API) на ранніх етапах розробки як основи всієї системи. Сучасні виклики, пов’язані з інтеграцією різнорідних сервісів, забезпеченням масштабованості, безпеки та зручності підтримки програмних продуктів, вказують на необхідність використання новітніх методологій інженерії програмних систем.

Актуальність теми зумовлена потребою у створенні ефективних, масштабованих і гнучких програмних рішень, які забезпечують швидку інтеграцію між компонентами системи та зовнішніми сервісами. Традиційні підходи до розробки програмного забезпечення часто не забезпечують належного рівня узгодженості між фронтендом і бекендом, що призводить до збільшення витрат часу на розробку та тестування. У свою чергу, API-first підхід дозволяє стандартизувати взаємодію між компонентами системи, підвищити якість документації та спростити процес командної розробки. У контексті сучасних вимог до швидкого випуску продуктів і їх безперервного вдосконалення застосування API-first підходу є важливим кроком до підвищення ефективності програмної інженерії.

Метою роботи є дослідження та розробка програмної системи із використанням API-first підходу, яка забезпечує ефективну взаємодію між компонентами, підтримує масштабованість, зручність інтеграції та можливість автоматизованого тестування.

Завданнями дослідження є аналіз предметної області та існуючих підходів до API-орієнтованої розробки, визначення функціональних і

нефункціональних вимог до системи, дослідження методів проєктування API та вибір відповідних інструментів, розробка архітектури програмної системи, реалізація API та клієнтського застосунку, проведення тестування та оцінка ефективності застосованих рішень. Також у процесі дослідження буде розглянуто питання автоматизації тестування, документування API та забезпечення узгодженості між компонентами системи.

Об’єктом дослідження є процеси інженерії програмних систем, що включають аналіз, проєктування, реалізацію та тестування програмних продуктів із використанням API-first підходу.

Предметом дослідження є методи, моделі та інструменти розробки програмних систем із використанням API-first підходу, зокрема проєктування API, використання стандартів (OpenAPI), інтеграція компонентів, а також автоматизація процесів тестування та розгортання.

Методи дослідження включають аналіз наукових і технічних джерел для визначення сучасного стану проблеми, порівняльний аналіз існуючих технологій і підходів, моделювання архітектури програмної системи, експериментальний метод для реалізації та тестування API, а також аналіз отриманих результатів для оцінки ефективності розробленого рішення.

Розроблена програмна система передбачає створення API як центрального елемента взаємодії між компонентами, реалізацію клієнтського застосунку, а також використання сучасних інструментів для документування та тестування. Особлива увага приділяється забезпеченню масштабованості, безпеки, узгодженості даних і зручності використання, що сприятиме підвищенню якості програмного продукту та ефективності процесу розробки.

Бакалаврська робота містить 79 сторінок, 32 рисунки, 1 таблиця, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ API-FIRST ПІДХОДУ

1.1 Сутність та принципи API-first підходу в інженерії програмних систем

Оскільки економіка API сьогодні є зростаючим бізнесом, важливо враховувати, які дії вжити організації, щоб взяти в ній участь . Пошук відповідного методу створення API як продуктів та правильного набору інструментів і платформ є відправною точкою для досягнення цієї мети. Початок застосування методу проектування API, який називається API-first design, може бути кроком до власної екосистеми API організації.

Предметом цього дослідження є API-first проектування та інструменти, які можна використовувати для підтримки цього методу проектування. API-first проектування – це відносно новий метод проектування API, який ще не був предметом багатьох досліджень. Це дослідження має на меті розкрити значення API-first проектування та показати його процес на практиці шляхом створення демонстраційного API та застосунку відповідно до цього методу. Воно може служити вступним матеріалом до API-first проектування, а також оцінює інструменти, що стосуються цього підходу до проектування.

Існує безліч інструментів, які можуть допомогти в проектуванні та розробці API на кожному етапі їх розробки. У цій роботі представлено стандарт, що визначає API, та пару інструментів, які використовуються для підтримки розробки API у хмарному середовищі Azure. Вони були обрані для відповідності API-first-проектуванню в цьому середовищі.

Головною метою цієї дослідницької роботи є з'ясування того, що таке API-first проектування та які його переваги , а також як застосувати метод API-first проектування в середовищі Azure API Management за допомогою сторонніх інструментів, що зробить цей процес ефективним.

Це дослідження має на меті з'ясувати, наскільки процес створення API можна прискорити за допомогою відповідних інструментів.

Для досягнення цієї мети, ця робота складається з дослідження API-first проектування, щоб зрозуміти, як застосовувати цей метод, представити відповідні інструменти та стандарти для підтримки API-first проектування та створити демонстраційний API та невеликий клієнтський застосунок з використанням цього підходу. Демонстрація створена лише для тестування та вступних цілей і виконується однією людиною. Її документація може бути використана як посібник з API-first проектування. Інструменти оцінюються щодо того, наскільки добре вони підтримують підхід API-first проектування та чи можуть вони зробити цей процес ефективним у середовищі Azure.

Проблема дослідження пов'язана з проблемою поєднання ручної розробки API зі зростанням обсягу документації, що перешкоджає швидкому розгортанню нових API. Хочеться розглянути можливість розробки API за допомогою попередньо визначених інструментів, мінімізуючи таким чином потребу в ручному введенні, отримуючи при цьому якісні API. Проектування з урахуванням API може бути одним із рішень цієї проблеми, тому воно є предметом дослідження цієї роботи.

Тему цього дослідження запропонувала Tieto Finland Oy. Tieto Education, галузь промисловості Tieto, створює цифрові рішення для догляду за дітьми та освіти. Tieto Education зацікавлена в економіці API та пошуку нових гнучких методів для створення та управління API. Тема демонстраційного API пов'язана з галуззю Tieto Educations.

Методологія дослідження, що використовується в цій роботі, - це дизайн-дослідження (прикладне дослідження дій). Дизайн-дослідження спрямоване на вирішення дослідницької проблеми або покращення поточної ситуації шляхом розробки практичних рішень. Що відрізняє дизайн-дослідження від будь-якої розробницької роботи, спрямованої на покращення або вирішення проблеми, так це документування роботи, а також її публікація та презентація.

Для цієї роботи було обрано дизайн-дослідження, оскільки його метою є покращення поточної ситуації. Замовник роботи хоче зробити свій поточний спосіб створення API швидшим та ефективнішим. Ця робота дотримується

ітеративної схеми дизайн-дослідження, що означає, що коли рішення дослідницької проблеми тестується, воно змінюється відповідно до результатів та тестується повторно.

Ітерації дотримуються структури, яка має чотири частини: початок, проектування, впровадження та оцінювання, як показано на рисунку 1.1. Ці чотири кроки виконуються в кожній ітерації з необхідними змінами у використовуваних інструментах та практиках. Відправною точкою в першій ітерації є використання набору інструментів та стандартів, які цікавлять замовника.

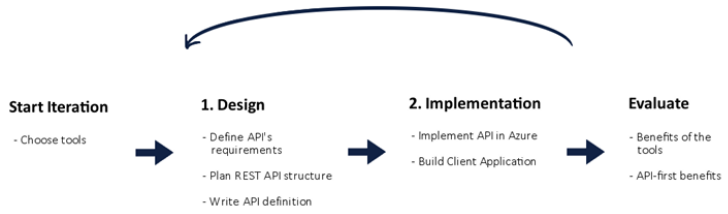

Рисунок 1.1 - Структура ітерацій

У кожній ітерації аналізуються два основні питання: переваги інструментів, що використовуються в ітерації, та загальні переваги API-first-проектування.

Результати кожної ітерації оцінюються з урахуванням того, наскільки добре використані інструменти сприяють процесу API-first проектування в середовищі Azure, і чи можна використовувати інструменти ефективніше порівняно з попередніми ітераціями. Переваги інструментів оцінюються за тим, як вони відповідають робочому середовищу Azure та чи приносять вони достатню цінність процесу порівняно з ресурсами (часом, грошима тощо), які вони потребують. Ітерації повторюються, доки не буде задокументовано та представлено задовільний API-first процес як результат цієї роботи.

У цьому дослідженні також оцінюються переваги проектування, орієнтованого на API. Результати ітерацій та кінцевий процес порівнюються зі

списком переваг, отриманих з точки зору системи відліку, та перевагами, виявленими в рамках цього дослідження. Очікується, що не всі переваги вихідного матеріалу можуть бути перевірені через характер цього дослідження. Такі переваги

це може стосуватися командної роботи, виробничого середовища та роботи з кількома проектами розробки, що не входить до предмета цієї роботи.

API-First Design – це підхід до архітектури та дизайну програмного забезпечення, який часто обговорюють та на який посилаються наші галузеві колеги, і він є новою темою в академічних колах та рецензованій дослідницькій галузі. Наше раннє розуміння сучасного стану цього підходу до дизайну випливає з наступних дослідницьких ініціатив:

- Короткий огляд [1]. Огляд стану API-FD в академічній та галузевій літературі повністю включено до наступного розділу. Публікація містить короткий огляд літератури з цієї теми та підкреслює галузево-орієнтований характер API-FD. Ця робота закладає основу для майбутніх досліджень у цій роботі та має на меті розширити базу знань, що підтримує API-FD.
- Концептуальний застосунок для розробки фронтенд-дизайну [2]. У роботі, яка також включена повністю, з точки зору програмної інженерії обговорюється створений прототип, демонструючи життєздатність застосунку, орієнтованого на інтерфейс користувача, з підтримкою API-FD. Концептуальний застосунок продемонстрував потенційні переваги автоматизованої інтеграції корисного навантаження API з точки зору WYSIWYG-інтерфейсу користувача та фронтенд-дизайну.

1.2 Роль API в сучасній розробці програмного забезпечення

API (інтерфейс прикладного програмування) – це сполучний елемент між програмами або компонентами, який дозволяє їм взаємодіяти один з одним. API дозволяють отримувати, надсилати, змінювати та видаляти дані за допомогою визначених викликів функцій з однієї програми до іншої. Виклик цих функцій

дає змогу використовувати програму або її дані в іншій програмі. Оскільки API – це інтерфейси, що викликаються через програми та використовуються програмістами, кінцевий користувач програми ніколи безпосередньо не контактує з API. Це код , метою якого є використання іншими веб-програмами. Хоча API функціонально завжди є частиною іншої програми, сьогодні API стали центральним елементом та джерелом бізнесу для багатьох програмних компаній.

Традиційне розуміння API полягає в тому, що вони потрібні лише для підтримки вже створених програм та задоволення потреби у використанні даних цієї програми. Зовсім недавно API стали помітною частиною бізнесу з двох причин: вони пропонують ефективнішу масштабованість програм для різних пристроїв всередині компанії та дозволяють пропонувати API як продукт зовні для інших компаній. Компанії можуть мати API як свій основний або навіть єдиний продукт.

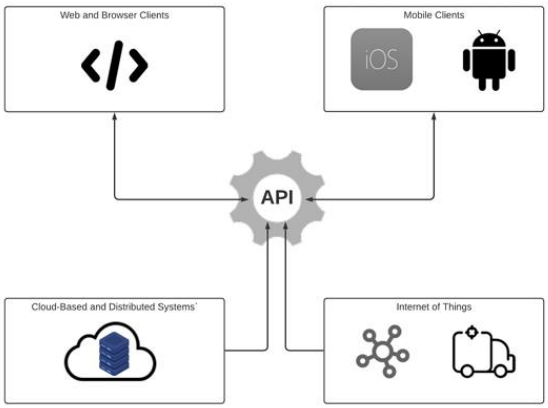

Рисунок 1.2 - Складність сучасної екосистеми API

API-економіка – це широке поняття, яке включає створення та використання власних внутрішніх API компанії, створення зовнішніх API для використання іншими та використання зовнішніх API від іншої компанії. Кожен із цих підходів приносить компанії користь, або безпосередньо фінансово, залучаючи нове джерело доходу чи нових клієнтів, або опосередковано, підвищуючи ефективність розробки. API-економіка – це постійно зростаюча галузь бізнесу, яка продовжує набирати популярності.

API-економіка близька до іншої концепції, яка називається монетизацією API. Існує багато бізнес-моделей, які можна використовувати для того, щоб почати отримувати прибуток від API. Важливо вибрати правильну модель для свого API, виходячи саме з мети та використання.

API можна розділити на три категорії: внутрішні або приватні API, партнерські API та зовнішні API. Усі категорії мають свої власні способи отримання переваг. API з кожної категорії можуть співіснувати всередині компанії або навіть певною мірою перетинатися, оскільки зовнішні та партнерські API також можуть використовуватися всередині компанії. Інший спосіб категоризації API полягає в тому, що вони поділяються на захищені та відкриті API. Захищені API вимагають авторизації користувача, тоді як відкриті API доступні для всіх.

Існують різні думки щодо того, скільки категорій API існує, оскільки деякі експерти з цього питання визначають внутрішні та приватні API як різні поняття, а інші – ні. Різниця між ними полягає в тому, що приватні API використовуються в публічному Інтернеті, а внутрішні API – у внутрішній мережі організації.

Внутрішні та приватні API використовуються всередині компанії для розробки власних сервісів. Зазвичай їх використовують розробники або люди, які працюють поруч з ними. Внутрішні API часто обробляють дані, що відрізняються від типу даних, що використовуються зовні, наприклад, дані, які не дозволено показувати клієнтам або конкурентам.

Партнерські API також використовуються іншими організаціями, проте лише тими, які мають угоду з постачальником API. Вони захищені від користувачів поза угодою та призначені для користі всіх пов'язаних сторін. Партнерські API захищені, наприклад, ключами API або підписки.

На відміну від партнерських API, зовнішні API відкриті для використання всіма та часто обробляють відкриті дані. Зовнішні API можуть відкрити нові бізнес-перспективи з двох точок зору: компанія може або пропонувати API для зовнішнього використання, або використовувати зовнішні API із зовнішнього

джерела . Зовнішні API, що пропонуються для публічного використання, також можуть використовуватися всередині організації. Це навіть рекомендується робити, оскільки таким чином API оновлюється регулярніше, а проблеми виявляються та виправляються швидше.

Навіть попри те, що зовнішні API доступні для всіх, можна стягувати плату за їх використання залежно від рівня використання та надавати пріоритет певним групам користувачів, щоб забезпечити їм кращий рівень обслуговування. Можуть існувати різні типи підписок та рівні ціноутворення. Зовнішні API часто використовують бізнес-модель *Freemium* , де базовий рівень використання є безкоштовним; однак для професійного використання потрібна платна преміум-підписка.

Іноді найкращим варіантом є використання зовнішнього API, створеного кимось іншим. Це економить час і позбавляє необхідності виконувати ту саму роботу, яку вже виконав інший постачальник API. Використання зовнішніх API, створених іншими компаніями, пов'язане з ризиком втрати підтримки цього API.

Кількість публічних API постійно зростає. Існують каталоги API, такі як *Programmableweb.com,* які надають платформу для публікації та пошуку зовнішніх API.

Платформи керування API містять функції для моніторингу та керування API. Зазвичай вони включають інструменти для розгортання та публікації API, а також їх обмеження, документування та керування версіями. Окрім цих найпоширеніших функцій керування, платформи керування API можуть мати інші інструменти, такі як керування користувачами, ціноутворенням та політиками.

Платформа керування API також може мати шлюз API для керування трафіком та портал розробника, який містить документацію API.

Наше флагманське дослідження, присвячене API-First Design, виникло завдяки галузевому досвіду створення хмарних додатків для індустрії фінансових технологій та бажанню вплинути на архітекторів та розробників програмного забезпечення щодо послідовного документування API. Зі

зростанням нашої кодової бази та команди також збільшувався час, необхідний експертам у предметній області, які писали наш код, для відповіді на запитання щодо функцій програмного забезпечення. Документування існуючих API для оптимізації комунікації щодо використання та споживання функцій у наших мікросервісах стало очевидною потребою. У пошуках інструментів, які автоматизують документування API, неодноразово з'являлися два терміни: API-First Design та API-Driven Design.

Байдуже розмежовуючи академічні кола та промисловість, автор шукав інформацію з двох каналів: рецензованої традиційної літератури та «сірої» літератури, створеної галуззю. Початкове дослідження рецензованих та академічних публікацій виявило невелику кількість статей, що обговорюють API-First Design, і ще менше тих, що чітко визначали мету підходу до розробки програмного забезпечення. Натомість, галузевий контент на цю тему був численним для обох термінів.

У наступному підрозділі досліджується стан публікацій API-First Design в академічному та рецензованому контексті на момент публікації [1].

У нашому попередньому пошуку публікацій про API-First Design у традиційній літературі ми виявили три статті, які надали огляд теми з цієї спільноти. Ми виявили, що доступні публікації зазвичай розглядають API-First Design лише як другорядну тему, а не обговорюють атрибути API-FD або не надають основи для впровадження. Наприклад, обговорюють роль, яку API відіграють у сучасному цифровому ландшафті. Вони наголошують на важливості розкриття можливостей системи через API у децентралізованих системах та бізнесі, для якого вони розроблені. Вони також припускають, що організації, що зосереджені на API, також повинні прагнути створити культуру API. Визнаючи, що розробка API є складним завданням, робота зосереджена переважно на розробці та управлінні API. Вважають, що API-First Design є кращим підходом для розкриття бізнес-можливостей через API. Специфіка API-First Design не обговорюється, що залишає простір для подальшого вивчення в майбутніх дослідженнях.

Обговорюють тематичне дослідження, проведене з Saxo Bank. У дослідженні розглядалося визначення набору тестів та покриття API, що надаються за допомогою підходу API-First до проектування сервісів. Проектування API-First є другорядною темою статті, оскільки основна увага в статті приділяється стабільності набору тестів та функціональному покриттю. Автори виявляють, що проблеми, пов'язані з тестуванням, часто виникають через відсутність початкового контексту (ПК), концепції, поширеної в розробці, орієнтованій на поведінку [21]. Вони роблять висновок, що підхід API-First покращує можливості та практики тестування, забезпечуючи посилену співпрацю у визначенні плану тестування за допомогою методології проектування, орієнтованої на API.

У вартих уваги прикладах дослідження API-First, представляють підхід API-First до проектування платформи Backend-as-a-Service (BaaS) [22]. Приклади платформи BaaS включають зберігання даних, керування користувачами та зберігання файлів, що використовуються сторонніми розробниками додатків. У поєднанні розробниками додатків, backend-сервіси дозволяють пришвидшити розробку повнофункціональних платформ. Окрім огляду методологій проектування компонентів BaaS, новизна цього дослідження полягає в підході API-First, прийнятому для проектування архітектури системи та встановлення основи платформи. Як обговорювалося у статті, переваги цього підходу включають слабо пов'язані послуги та швидший вихід на ринок продуктів, що використовують функції BaaS. Автори цієї публікації визнають, що процес розробки в підході API-First починається з визначення та впровадження API, які описують платформу; фреймворки API-First Design не включені до обговорення. Однак, на відміну від статей, які розглядають API-First Design як другорядну тему, Дуджак та ін. підкреслити його переваги, включаючи сприяння слабкому зв'язку між сервісами та розкриття функцій сервісів через єдиний API. Цілі та переваги, викладені в цій статті, збігаються з тими, що визначені в галузевій та сірій літературі, що є позитивним кроком до подолання розриву між двома секторами цієї галузі.

Хоча вищезазначені статті не є вичерпними, вони представляють літературу, опубліковану академічною спільнотою на момент написання. Мінімальна кількість публікацій, що обговорюють API-First Design, свідчить про новизну дослідження в академічних колах. Це відкриває можливість ознайомитися з роботою наших колег у галузі.

На відміну від обмежених досліджень API-First Design в академічному середовищі, існує безліч офіційних документів, блогів, журналів та галузевих відеоконтентів, що обговорюють API-First Design. Широкий контент надає можливість зрозуміти принципи, переваги та недоліки API-First Design.

У 2002 році Джефф Безос, засновник і генеральний директор Amazon, розіслав своїм співробітникам службову записку, яка зобов'язувала визначати та використовувати сервісні інтерфейси для технологій та міжкомандної комунікації [23]. Цей Мандат API є раннім втіленням підходу API-First до розробки програмного забезпечення [24] та побудови культури API. Майже 20 років тому Amazon пропонує повний набір послуг та хмарних продуктів. Показовими для еволюції Amazon як технологічної компанії, що орієнтована на API, з моменту Мандату 2002 року є Amazon Marketplace Webservice (MWS) [25] та Amazon Web Services (AWS) [26], пакети API, які полегшують використання торговельного майданчика Amazon та хмарних технологій сторонніми по відношенню до самої компанії особами. Ще одним показником важливості моделі API-як-товару є пакети API, опубліковані для використання розробниками іншим гігантом Big 5 технологій, Google [27]. До його набору активів BaaS з доступом через API входять платформа Maps Platform, Google Ads та Google Analytics [27].

В результаті цифрової трансформації, спричиненої COVID-19, автори Ван та Макларті [28] обговорюють зміну ментальної моделі, яку бізнес-лідери повинні прийняти для підтримки технологічного та бізнес-оновлення, необхідного для переживання такої трансформації. Автори пояснюють, що сектори охорони здоров'я, банківської справи та логістики можуть отримати вигоду від розробки та поширення API. Вони також стверджують, що менші

компанії можуть досягти успіху навіть серед великих компаній, надаючи свої дані через API. Як і у випадку з Мандатом Безоса щодо API та зміною філософії розробки та комунікації, автори зазначають, що компанії повинні навчитися виявляти можливості для розкриття бізнес-потенціалів через

набір чітко визначених API [28].

У дописі в блозі 2020 року [29] автор Кріс Тоцці обговорює позитивний вплив розробки на основі API на стратегію розвитку підприємства. Тоцці надає безцінний огляд переваг дизайну на основі API. Переваги використання API як основи архітектурної стратегії є значними та актуальними. Цей підхід забезпечує виявлення компонентів, які могли бути проігноровані в альтернативних стратегіях API, що призводить до модульних, хмарно-сумісних додатків та послуг, покращених конвеєрів CI/CD та гарантії сумісності API завдяки початковій зосередженості на розробці API. Відсутній в академічних колах, Тоцці підсумовує кроки, які можуть допомогти успішно впровадити дизайн на основі API; цей процес показано на рисунку 1.2. Перші кроки включають оцінку існуючих, нових та необхідних API, а також цільових та життєздатних архітектурних шаблонів. Наступні кроки зосереджені на розробці, тестуванні та моніторингу API, інтегрованих у конвеєр CI/CD. Останнім і важливим кроком є визначення та дотримання проактивного циклу зворотного зв'язку API.

На основі результатів опитування громадської думки 2018 року, показаних на рисунку 1.3, що серед сучасних архітектурних підходів до проектування розподілених систем архітектори найбільше підтримують API-First Architectures (67%). У статті Лазар визначає деякі переваги API-First Design, які можуть сприяти уподобанню опитаних архітектурних елементів. Розділення фронтенд-компонентів та бекенд-компонентів системи можна досягти за допомогою чітко визначеного API.

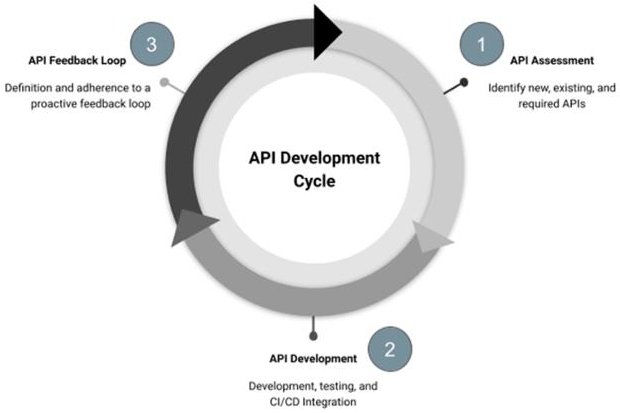

Рисунок 1.3 - Цикл розробки API

З повсюдним поширенням хмарних, розподілених та ресурсомістких систем, SOA стала стандартом для проектування програмного забезпечення для підприємств у галузі [6]. Зокрема, системи інкапсулюють бізнес-функціональність, правила та логіку домену у слабо пов'язаних, незалежно розгортаних додатках, відомих як мікросервіси. Незважаючи на велику документацію та дослідження методологій проектування та чітко визначених рекомендацій щодо впровадження мікросервісів, залишаються проблеми з визначенням меж мікросервісів та досягненням належної гранулярності [8, 35, 7]. Крім того, незважаючи на наслідки проектування мікросервісів, яке позиціонує явний інтерфейс як основу проектування [4], залишаються проблеми з визначенням точних, опублікованих інтерфейсів [6, 8], що є критичним аспектом проектування, який часто розглядається наприкінці циклу розробки.

Важливість API для сучасних програмних систем привернула увагу до дизайну API та його ролі в загальному дизайні систем, що призвело до появи API-First Design. Однак у нашому дослідженні ми виявили лише обмежену кількість рецензованих статей, що обговорюють цю тему. Крім того, ми виявили, що API-FD найчастіше є другорядною темою в дослідженні. Контент, який безпосередньо визначає або обговорює API-FD та його атрибути, або практики, що допомагають його впровадженню чи розгортанню, зустрічається рідко або взагалі відсутній. На відміну від мінімального контенту, доступного в академічному та рецензованому середовищі, існує безліч галузевих офіційних

документів, блогів, журналів та відеоконтенту. Дослідження підтверджують важливість включення голосів галузевих партнерів до академічних досліджень [36, 37, 38, 13]. Використання «сірої» та галузевої літератури підвищує актуальність досліджень у галузі програмної інженерії, особливо для сучасних тем [14, 39]. Таким чином, великий контент надає можливість зрозуміти принципи, переваги та недоліки API-First Design та зробити свій внесок у спільноту програмної інженерії, що є основою цієї роботи.

Далі наведено початкові відкриття та фундаментальні принципи цього нового підходу до архітектури та дизайну програмного забезпечення, а також зростаючу важливість API в цифрову епоху.

- Фахівці з програмної інженерії сходяться на думці про важливість якісних API та переваги підходу API-First Design.
- Заявлені переваги API-First Design, окрім чітко визначеного API, включають розширений доступ до бізнес-можливостей, покращену ідентифікацію системних компонентів, чіткі межі між логікою фронтенду та бекенду та самими сервісами, покращене масштабування та збільшення швидкості розробки завдяки паралелізації зусиль [40, 35, 29].
- Важливість добре розроблених API зростає, оскільки компанії розглядають API як активи та товари, і тому прагнуть максимізувати можливості, що надаються через набір API.

Найбільш захопливим висновком, зробленим з цього короткого огляду, є необхідність додаткових досліджень API-First Design. Спираючись на це усвідомлення, автор прагнув визначити існуючі інструменти, які могли б допомогти покращити впровадження API-FD. Обговорення цих зусиль наведено в наступному розділі.

1.3 Концепція та обґрунтування використання API-first дизайну

Сервісно-орієнтована архітектура (SOA) та, точніше, мікросервісна архітектура (MSA) – це стандартні підходи до проектування сучасних

розподілених систем та систем, що інтенсивно працюють з даними. Загальні методології проектування програмного забезпечення та мікросервісів, що застосовуються до SOA та MSA, включають проектування на основі даних, моделей та проектування на основі предметної області. Зрілість цих методологій демонструється великою документацією, що обговорює теми та доступність інструментів моделювання програмного забезпечення та генерації коду у контексті відкритого коду, академічному та комерційному контекстах.

Розвиток хмарних та розподілених програмних систем дозволяє компаніям надавати своїм клієнтам контент у новому світі розподілених технологій. Однак конкуренція в цьому світі вимагає зміни мислення; здатність компанії надавати контент має розглядатися як пріоритет розробки продукту, а інтерфейс прикладного програмування (API), через який пропонується контент, є товаром. На підтримку цієї еволюції в дизайні, API є центральним елементом дизайну та архітектури програмного забезпечення. Крім того, як обговорювалося раніше, фахівці галузі сходяться навколо методології дизайну, API-First Design, яка виводить API на передній план дизайну продуктів та систем.

На відміну від більш зрілих методологій проектування, початковий рівень API-First Design надає можливості для дослідження цього підходу та його застосовності до різних етапів життєвого циклу розробки програмного забезпечення (SDLC). Існуючі інструменти генерації коду генерують код на основі визначеної моделі або набору даних; результат, як правило, орієнтований на бізнес та бекенд. Мало які інструменти допомагають у розробці коду на основі схеми API або корисного навантаження. Ще менше з них спрямовані на інтеграцію API з компонентами фронтенду та інтерфейсу користувача. Мотивоване API-First Design та виявленням можливостей для сприяння його впровадженню, дослідження, описане в цьому розділі, є традиційним проектом у стилі розробки програмного забезпечення, який прагне заповнити прогалину в доступних інструментах, орієнтованих на API-First, та забезпечити життєздатний прототип утиліти інтеграції API, яка полегшує інтеграцію API та дизайн інтерфейсу користувача.

Решта підрозділів організовані таким чином: підрозділ 2 містить огляд проблем, пов'язаних з API-First Design, з точки зору користувацького інтерфейсу; підрозділ 3 представляє відповідну роботу;

Сучасні програмні рішення для підприємств є ресурсоємними та високо розподіленими. Мікросервісна архітектура (MSA), еволюційна адаптація сервісно-орієнтованої архітектури (SOA), є фактичним стандартом проектування для вирішення проблем, що виникають у системах корпоративного програмного забезпечення. Повсюдне поширення SOA та MSA призвело до чітко визначених та добре задокументованих практик і методологій проектування та впровадження компонентів, що складають сучасну розподілену програмну систему

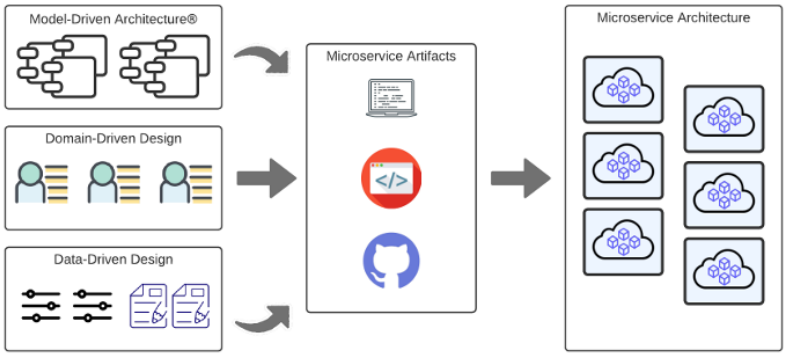

Рисунок 1.4 - Процес проектування мікросервісів

Як показано на рисунку 1.4, поширені підходи до проектування SOA включають модельно-орієнтовану архітектуру (MDA) та проектування, кероване доменами (DDD) [11]. MDA, розширення модельно-орієнтованого проектування (MDD), представленого приблизно у 2001 році, заохочує абстрагування складнощів системи за допомогою моделей, представлених уніфікованою мовою моделювання (UML). У 2003 році Ерік Еванс представив проектування, кероване доменами, методологію, яка передбачає, що реалізації програмного забезпечення відображають реальні бізнес-контексти та їхніх користувачів [11, 43]. MDA®, DDD та подібні методології проектування програмного забезпечення

зосереджені переважно на оптимізації проектування та розробки компонентів, що інкапсулюють основну бізнес-логіку та загалом спрямовані на проектування самого мікросервісу.

Також представлено на рисунку 1.4, Data-Driven Design описує методологію, за допомогою якої дані, зібрані про користувача, систему або процес, впливають на рішення щодо проектування програмного забезпечення. Data-Driven Design застосовується до різних програмних систем, включаючи бездротові технології, модулі відмовостійкості та пристрої, пов'язані з Інтернетом речей (IoT). Крім того, Data-Driven Design застосовний та адаптивний до мікросервісного та фронтенд-проектування.

Точний API є важливим артефактом добре розробленого мікросервісу [4]. Тим не менш, залишаються проблеми у визначенні та публікації чітко визначених API [6, 8]. Як обговорювалося в [1], складність у визначенні API для сучасних корпоративних систем полягає в обсязі та масштабі підтримуваних пристроїв. Крім того, API часто розглядається на пізніх етапах циклу розробки, що знижує його важливість. На противагу цьому, API-First Design є новою методологією для проектування мікросервісів [1]. Її принципи зосереджені на API як основі системи та припускають, що API розглядається на ранніх етапах циклу розробки. API-First Design також вказує на те, що всі бізнес-можливості розкриваються через чіткі, точні та чітко визначені API [19, 1].

У контексті асистованого мікросервісного та розподіленого проектування систем, доступні засоби автоматизованої генерації коду та методології проектування в першу чергу зосереджені на даних, бізнес-логіці та компонентах серверних сервісів, що складають систему. Також існують інструменти, які допомагають в автоматизованому створенні документації API. Мотивацією цього розділу є те, що мало інструментів враховують важливість та вплив API з точки зору компонентів системи, орієнтованих на користувача. Існує також мінімальна кількість інструментів, які допомагають генерувати фронтенд та користувацький код з моделі програмного забезпечення або корисного навантаження API. Цей проект прагне заповнити цю прогалину та надати

життєздатний прототип утиліти інтеграції API, яка поєднує API-First Design, інтеграцію API та дизайн інтерфейсу користувача.

Зі змінами в галузі, ймовірно, спричиненими Amazon [24], важливість чітко визначених API спонукала компанії розглядати API як товар. API-First Design набирає обертів як бажана методологія проектування MSA, що підтримує цей фокус API [1]. Цю підтримку демонструє доступна сіра література (публікації, опубліковані поза академічною сферою), включаючи.

Генерація програмного забезпечення на основі моделей – це добре досліджена тема, яка зосереджена на створенні предметно-орієнтованого коду мікросервісів та серверної частини з діаграм UML. Наприклад, Суніта та ін. демонструють життєздатність перетворення поведінкових моделей UML у XML та виконуваний код. Пізніше автори досліджують життєздатність використання метамоделей, що пов'язують UML з формальними виразами мови обмежень об'єктів (OCL) в межах UML.

Ріверо та ін. визнають, що методології модельно-орієнтованої розробки та гнучкої розробки повинні належним чином зосереджуватися на важливості якісного дизайну API. Автори підтверджують свої занепокоєння, впроваджуючи структуру метамоделі, за допомогою якої на дизайн API накладаються обмеження найкращих практик, а також розробляються та тестуються API [40].

Як показано вище, робота, пов'язана з API-First Design та створенням програмного забезпечення, в основному зосереджена на моделюванні та дизайні серверних систем таблиця 1.1 .

Дослідження інструментів та застосунків, орієнтованих на користувача, які зосереджені на автоматизації інтерфейсу користувача з підходом API-First, практично відсутні.

Таблиця 1.1:

Вимоги рівня 1

Id (Ідентифікатор)	**Requirement Description (Опис вимоги)**
R-01	Додаток має використовувати сучасні мови та технології веб-розробки.

R-02	Додаток має дозволяти користувачеві надавати корисне навантаження API (payload) як вхідні дані.
R-03	Додаток має дозволяти користувачеві надавати схему корисного навантаження API як вхідні дані.
R-04	Додаток має надавати користувачеві вигляд інтерфейсу (UI), що базується на API.
R-05	Додаток має містити набір примітивних типів даних для асоціації з корисним навантаженням.
R-06	Додаток має дозволяти користувачеві асоціювати елемент корисного навантаження з компонентом.
R-07	Додаток має виявляти та сповіщати користувача про некоректні вхідні дані корисного навантаження.
R-08	Додаток має виявляти та сповіщати користувача про некоректні вхідні дані схеми корисного навантаження.
R-09	Додаток має дозволяти користувачеві впорядковувати дані корисного навантаження API для відображення в інтерфейсі.
R-10	Додаток має дозволяти користувачеві позначати фронтенд-компоненти, пов'язані з даними API, як видимі або приховані.
R-11	Додаток має генерувати точний фронтенд-код.

На початку дослідження життєздатності прототипу API Integration ми звернулися за порадою до галузевих розробників програмного забезпечення. Ми визначили та сформулювали вимоги до застосунку API Integrator, який ми створили за допомогою традиційних практик виявлення вимог до програмного забезпечення. У наступних розділах детально описано питання виявлення та зазначено визначені вимоги, що використовувалися для формування розробки прототипу застосунку.

У наступних абзацах містяться питання для співбесіди, спрямовані на виявлення вимог та резюме від чотирьох зацікавлених сторін проекту, визначених для зручності. Серед експертів з предметної області (SME), які беруть участь в співбесіді, є два розробники програмного забезпечення повного циклу та два інженери з автоматизації тестування, всі з ступенем бакалавра або вище з комп'ютерних наук, і всі вони на момент написання статті працювали в Figure Technologies.

Запитання 1: Яка обов'язкова функція інструменту для генерації коду?

Функції, визначені як обов'язкові для інструментів генерації коду, включають підтримку мов, якими генерується код. В ідеалі, користувачі домену та експерти знайомі з цільовою мовою та не потребують вивчення

альтернативної мови для використання згенерованого коду. Малий та середній бізнес також висловили потребу в чітко визначених типах (примітивах та об'єктах) та типах даних. Крім того, потенційні користувачі визначили необхідність перевірки та тестування згенерованого коду. Як мінімум, скомпільований код є прийнятним; проте ідеальним є включення згенерованих тестів. Експерти з предметної області висловили необхідність версії вхідних даних API та результуючого коду. Нарешті, експерти з предметної області запропонували використовувати код з функцією перетягування.

Параметри через інтерфейс користувача є важливими для інструменту генерації коду інтерфейсу користувача.

Запитання 2: Які проблеми або ризики пов'язані з автоматичною генерацією коду?

Двома основними ризиками генерації коду є безпека та захист згенерованого коду, а також його якість. Користувачі висловили стурбованість тим, що погано сформований, недійсний або зловмисний вхід може призвести до неправильно згенерованого, помилкового та небезпечного коду. Опитані також зазначили, що інструменти генерації коду можуть не мати повного набору функцій та опцій налаштування, залишаючи кінцевих користувачів без необхідної функціональності коду, створеного вручну, що може призвести до коду з непередбачуваною поведінкою. Нарешті, одна з опитаних осіб висловила стурбованість тим, що інструменти генерації коду мають криву навчання, яка може переважувати переваги коду, згенерованого розробником.

Запитання 3: Як виглядає успіх інструментів для генерації коду?

Майже всі опитані представники малого та середнього бізнесу заявили, що успіх інструменту генерації коду залежить від простоти його використання та налаштування. Користувачі також відзначили важливість керування версіями та якості коду, а також його схожість або покращення якості коду, створеного розробниками.

Запитання 4: Які найбільші проблеми пов'язані з інтеграцією API у фронтенд-продукти?

Респонденти обговорили проблеми інтеграції API у фронтенд-код, включаючи виявлення критичних змін, що потребують версіонування, залежності фронтенду та API від зовнішніх компонентів, а також загальні проблеми розробки програмного забезпечення, пов'язані з тестуванням, якістю та розгортанням. Розробники, з якими проводилося опитування, також зазначили, що орієнтованість API на дані складно інтуїтивно відобразити кінцевому користувачеві, тому вони вітають ідею WYSIWYG-утиліти, яка допомагає в дизайні та інтеграції.

Запитання 5: Як виглядає процес інтеграції нового корисного навантаження API з дисплеєм або функцією фронтенду?

Відповідь на це питання варіювалася від креативної організації та відображення даних до обов'язкової функції, за якою фронтенд відповідає визначеному контракту API. Крім того, малі та середні підприємства висловили потребу забезпечити достовірність даних та належне відображення помилок, коли надходять неочікувані вхідні дані від API.

Запитання 6: Як корисне навантаження API може впливати на рішення щодо дизайну інтерфейсу користувача та розміщення даних?

Малий та середній бізнес підійшов до цього питання з кількох точок зору. По-перше, з точки зору розробки та впровадження, респонденти зазначили, що контент, керований корисним навантаженням API, передбачає модульний дизайн та відображення через динамічний характер даних API. З точки зору UX, респонденти зазначили, що орієнтований на користувача контент у застосунку повинен визначати дизайн API.

Запитання 7: Як інструмент інтеграції API та генерації коду покращить ваш досвід розробки?

Усі опитані вважають, що інструмент інтеграції API та генерації коду збільшить швидкість розробки, усунувши ручне та повторюване впровадження коду. Один потенційний користувач вважав, що інструмент інтеграції допоможе усунути надлишковість коду.

Запитання 8: Як виглядає успіх інструменту інтеграції API фронтенду?

Успішні інструменти інтеграції фронтенд-API, на думку опитаних представників малого та середнього бізнесу, повинні генерувати код без помилок та мати можливість виявляти та позначати зміни у вхідному API. Успішні інструменти також повинні скорочувати час, необхідний для початкової інтеграції API. Одне з представників малого та середнього бізнесу зазначило, що успішна утиліта зменшить обсяг прийняття рішень, пов'язаних з представленням даних. Нарешті, одне з представників малого та середнього бізнесу заявило, що функціональність API, що надається набором інструментів, свідчить про успіх, коли вона пропонує двосторонню генерацію та інтеграцію API.

Запитання 9: Хто є основними користувачами інструменту інтеграції API?

Представники малого та середнього бізнесу, які брали участь в інтерв'ю, вважали, що інструмент інтеграції API найбільше підходить для розробників програмного забезпечення та тестувальників. Експерти вважали, що менеджери продуктів також можуть бути потенційними користувачами програми. *Запитання 10: Які функції потрібні для розширення інструменту інтеграції API для використання нетехнічним персоналом?* Функції, необхідні для розширення інструменту інтеграції API для нетехнічних користувачів, включають графічний інтерфейс користувача, документацію та чітке відображення результуючого інтерфейсу користувача. Крім того, вибір зіставлення інтерфейсу користувача з даними API має бути лаконічним та зрозумілим для нетехнічного користувача.

Керуючись відповідями на контрольні питання та дотримуючись традиційного підходу до розробки вимог, ми визначили вимоги до застосунку API Integrator. Вимоги рівня 1 є обов'язковими для впровадження перевірки концепції та визначені в таблиці 2.1. Додаткові вимоги, ідеальні для повнофункціонального застосунку API Integrator та життєздатні кандидати для майбутньої роботи над цим проектом, виходять за рамки цієї роботи.

Дослідження API-First Design та його застосовності до генерації коду, зосередженого на артефактах інтерфейсу користувача та фронтенду, пропонує унікальну перспективу, яка зазвичай не розглядається в проектуванні мікросервісних систем. Натомість, основна увага частіше приділяється реалізації

та межам мікросервісу або виключно інтерфейсу користувача з точки зору взаємодії людини з комп'ютером (HCI) та користувацького досвіду (UX). Застосування, що обговорюється тут, доповнює діяльність API-First Design, чітко зосереджуючись на API-Driven frontend design та інтеграції API. Вимоги, виявлені у фахівців галузі у відповідь на питання, включені до заявки на підтвердження концепції.

З огляду на виявлені вимоги, цільова аудиторія цього підтвердження концепції обмежена технічною аудиторією (наприклад, розробниками, інженерами з тестування та технічно підкованими менеджерами продуктів).

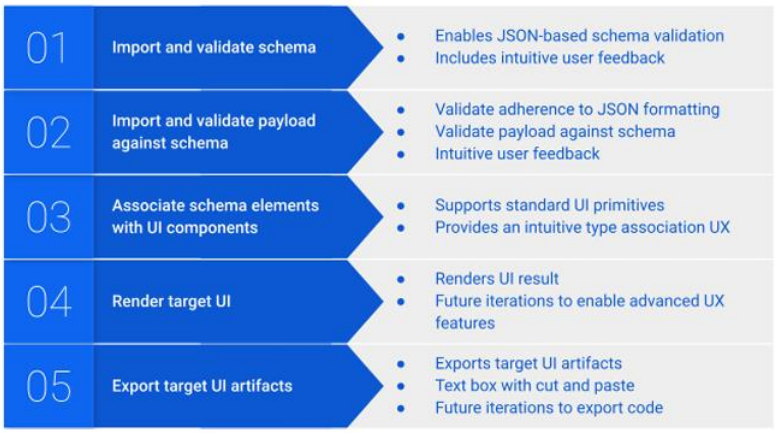

Рисунок 1.5 - Основний робочий процес інтегратора API-FD

Технології, що використовуються для цього прототипу, є стандартними веб-технологіями, включаючи JavaScript, JSON та React; доступ до програми здійснюється через сучасний браузер, такий як Google Chrome або Firefox.

Основний робочий процес прототипу API-FD Integrator показано на рисунку 1.5. Робочий процес включає схему на основі JSON та інтеграцію корисного навантаження API в рамках застосунку для дизайну інтерфейсу користувача, а також функціональність для генерації відповідних артефактів коду фронтенду.

Важливою функцією, реалізованою цією програмою, є можливість для користувача перевіряти корисне навантаження API на відповідність відомому набору правил корисного навантаження. Корисне навантаження JSON може бути

оголошене та перевірене на відповідність пов'язаній схемі JSON. З цією метою, цей проект використовує валідатор схеми AJV, щоб надати користувачам можливість розробляти інтерфейс користувача на основі перевірених корисних навантажень

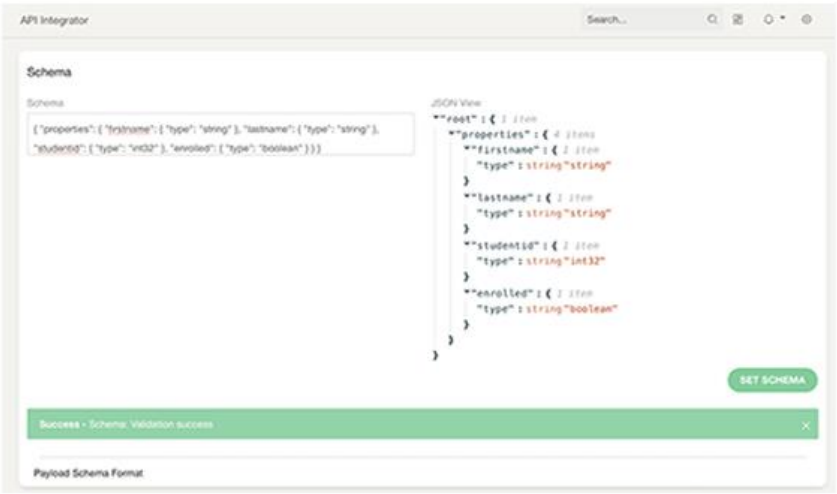

Рисунок 1.6 - API Integrator: Управління корисним навантаженням

Крім того, як показано на рисунку 1.6, застосунок містить інтерфейс, що дозволяє користувачам пов'язувати вхідні значення корисного навантаження з підтримуваними компонентами інтерфейсу користувача, і складається з цільового рендерингу інтерфейсу користувача на основі конфігурації компонента.

Функціональність інтеграції API, реалізована в рамках проекту, що обговорюється в цьому дослідженні, демонструє життєздатність підходу API-First для інтеграції API та дизайну інтерфейсу користувача. Як показано на рисунках 1.6 та 1.7, інтерфейс користувача є чистим та інтуїтивно зрозумілим, представленим як односторінковий додаток із сучасним макетом, кольорами та стандартними веб-технологіями на основі шаблону від Creative Tim.

Зародки API-First Design надають можливість дослідити комплексні можливості, що пропонуються цією новою методологією дизайну. Прототип, розроблений для цього проекту, забезпечує основу для майбутньої роботи, що досліджує застосовність API-First Design до загального SDLC. Цей проект

досліджує можливості впровадження API-first підходу до дизайну програмного забезпечення шляхом інтеграції дизайну інтерфейсу користувача та UX з розробкою API.

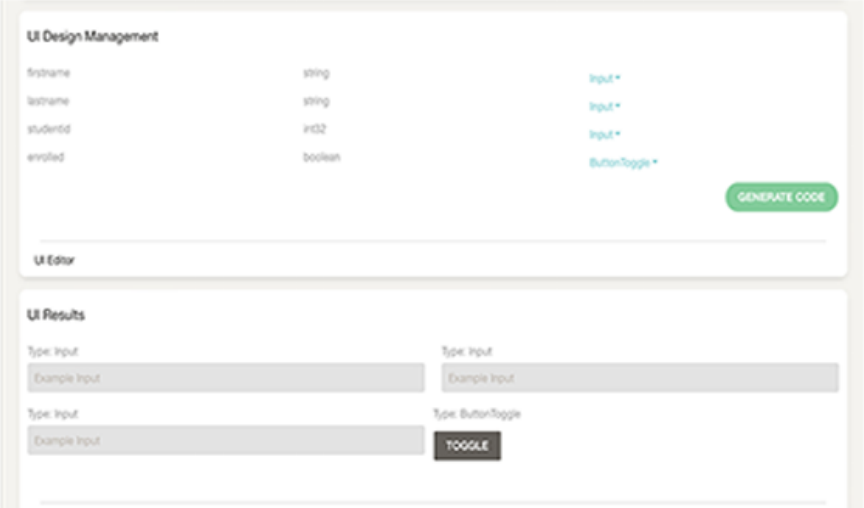

Рисунок 1.7 - API Integrator: дизайн інтерфейсу користувача

Як обговорювалося та проілюстровано на рисунку 1.5, прототип охоплює повну реалізацію основного робочого процесу для запропонованого застосунку. Майбутні ітерації цього проекту можуть включати додаткові вимоги, виявлені нашими галузевими малими та середніми підприємствами, а також наступні покращення функцій та зручності використання.

- Динамічне споживання корисного навантаження, отриманого з URL-адреси або опублікованого API.
- Основи проектування HCI, включаючи групування, балансування та покращення доступності.
- Розширені функції дизайну інтерфейсу користувача, включаючи перетягування, упорядкування та покращені функції CSS та стилізації.

Мікросервісна архітектура (MSA) є кращим підходом до проектування для вирішення проблем сучасних розподілених систем з інтенсивним використанням даних. Хоча чітко визначений інтерфейс є наслідком MSA, увага до API часто приділяється наприкінці циклу розробки. API-First Design

(проектування на першому місці) – це нова методологія для проектування програмного забезпечення та MSA [1], яка передбачає, що чіткий та чітко визначений API є основою проектування системи. Методологія вимагає додаткових досліджень, пов'язаних з її застосовністю в усій SDLC (проектуванні на першому місці). Хоча проектування та впровадження API є повсюдними в програмній інженерії, дослідження генерації коду, зосередженого на результатах користувацького інтерфейсу та фронтенду з акцентом на API-First, забезпечує унікальну перспективу, яка зазвичай не розглядається в проектуванні та розробці систем. Застосування інтеграції API, що обговорюється тут, забезпечує відправну точку для цього дослідження та підтримує покращення життєздатності API-First Design.

Формальне дослідження користувачів, яке аналізує зручність використання та застосовність застосунку API-First Integration, може визначити напрямок майбутніх ітерацій цього проєкту, додатково продемонструвати життєздатність API-First Design та вплинути на впровадження методології API-First Design.

Зрештою, інструменти та стандарти, доступні у спільноті розробників програмного забезпечення, можуть виявитися цінними в майбутніх дослідженнях та розробці підходу API-First Design. Серед них - специфікація OpenAPI [34] та мова моделювання RESTful API (RAML). Використання стандартів та інструментів, що підтримують розробку та тестування API, таких як Postman та Swagger, також може підвищити життєздатність та застосовність цього дослідження в галузі.

1.4 Висновки по розділу

У першому розділі було розглянуто теоретичні основи інженерії програмних систем із використанням API-first підходу. Проаналізовано сутність і ключові принципи API-first дизайну, що передбачають пріоритетне проєктування інтерфейсів взаємодії між компонентами системи. Досліджено

роль API в сучасній розробці програмного забезпечення як основного механізму інтеграції та взаємодії сервісів. Також обґрунтовано доцільність використання API-first підходу для підвищення гнучкості, масштабованості та повторного використання програмних компонентів. Отримані результати формують теоретичну основу для подальшого дослідження методів проєктування та реалізації API-орієнтованих систем.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ПРОЕКТУВАННЯ API-FIRST СИСТЕМ

2.1 Методологія API-орієнтованого проєктування (API-first design)

У цьому розділі пояснюється важливість інвестування зусиль у проектування API та представлено метод проектування з урахуванням принципу

API.

Проектування API означає прийняття обдуманих рішень щодо різних аспектів структури API та процесу його розробки. Проектування API має на меті покращити якість API, що робить їх легшими та приємнішими для розробників у використанні та зменшує кількість помилок. Проектування API безпосередньо впливає на його структуру, документацію та синтаксис, а через ці фізичні особливості опосередковано на багато інших аспектів розробки та використання API. Дизайн API можна описати як план вашого API. Це основа, яка допомагає створювати успішні API. Проектування API вимагає зусиль для вивчення та впровадження на практиці; тим не менш, це варто зробити, оскільки добре розроблені API та люди з навичками їх створення можуть допомогти розвинути функціонуючу екосистему API з внутрішніх, партнерських та зовнішніх API.

Гарний дизайн API має на меті надати API певні характеристики. Три бажані риси для API *легко читати та працювати з ним, важко неправильно використовувати* , а також *він повний та лаконічний.* Досвід розробника, також відомий як DX (скорочено), є важливою концепцією, що лежить в основі популярності API; API, які мають хороший досвід розробника, будуть використовуватися та подобатися розробникам. Гарне проектування власних API суттєво впливає на досвід розробника.

Існує чотири загальні стратегії проектування API:

- Стратегія комплексного проектування: цей метод використовується, коли додаток та серверна система існують до початку розробки API. Новий API додається між системами згодом. За допомогою цього підходу до проектування існуючі системи можна використовувати для допомоги у проектуванні та створенні API. Це вважається найшвидшим способом отримати робоче рішення, проте він не має найкращих вимог для забезпечення гарного дизайну API.

- Стратегія «зеленого поля»: коли немає існуючих програм чи систем, проектування API починається з нуля без будь-яких базових технологій. Ця стратегія робить можливим використання абсолютно нових технологій завдяки

початку з самого початку.

- Гнучка стратегія: Ітеративний гнучкий підхід до розробки API , який дозволяє швидкі зміни . Він дотримується ідеї, що розробка може розпочатися до того, як будуть готові всі специфікації. Гнучку стратегію рекомендується використовувати лише до публікації API.
- Стратегія фасадів: вона містить елементи як стратегій «болт-он», так і стратегій «зеленого поля». Ця стратегія зазвичай використовується в компаніях, де системи вже існують; проте підхід «зеленого поля» використовується в розробці API.

Розпочинаючи процес проектування API, важливо визначити, які дані слід розкрити, найкращий спосіб їх розкриття та як можна покращити API.

Різні організації можуть мати власні рекомендації щодо розробки API, які допомагають всі їхні API мають узгоджену структуру та якість у всій компанії.

Найкращі практики для проектування REST API. Існують найкращі практики проектування REST API, які варто враховувати під час написання нових API. Ці практики зосереджені на тому, щоб зробити REST API чистими та узгодженими, використовуючи при цьому структуру REST API на його користь.

- Назви ресурсів повинні бути у множині.
- Назви ресурсів повинні бути іменниками, без дієслів.
- Методи HTTP слід використовувати відповідно та використовувати для опису дій.
- Підресурси слід використовувати, коли ресурси пов'язані один з одним .
- Коди стану HTTP повинні повертати належний зворотний зв'язок.
- API повинні мати версії.

Проектування, орієнтоване на API, – це один із методів проектування та розробки API. Згідно з цим методом, розробка API починається з етапу проектування. На практиці це означає, що функціональність API спочатку планується та описується у стандартизованому форматі, а код для її реалізації створюється відповідно до цього плану. Життєвий цикл API завжди починається

з бізнес-потреби, яку API може задовольнити. Згідно з підходом «проектування, орієнтоване на API», наступним кроком є створення визначення API у стандартизованому машинному та людиночитальному форматі, який відповідає цій потребі. Цей метод розробки API є відносно новим, але він швидко стає все більш поширеним. «Проектування, орієнтоване на API, – це низка найкращих практик [...], які надають пріоритет кращому досвіду розробника».

API-орієнтований дизайн також називають API-орієнтованим дизайном, API-орієнтованим дизайном або схемотехнічним дизайном (Rosenstock 2018).

API-first проектування можна визначити як підхід до розробки лише API, або як підхід до розробки також додатків. Різниця між двома підходами до API-first проектування полягає в тому, що останній підкреслює, що API слід реалізувати до створення додатка. Це бажана ситуація для використання API-first проектування, оскільки метод має найбільше переваг, якщо новий додаток створюється на основі API. Інший підхід зосереджується лише на розробці API та не включає розробку додатків у визначення API-first проектування. Підхід API-first можна використовувати, навіть якщо існує існуючий додаток або функціональність, оскільки створення визначення API перед впровадженням все ще є корисним способом розробки API.

Проектування з використанням API підпадає під стратегію «зеленого поля», де процес проектування починається без базових технологій. Ця стратегія базується на симуляції API. Розробники можуть почати роботу над новим застосунком, який використовує API, розробляючи його на основі макета цього API. Це означає, що API можна використовувати до того, як з'явиться готовий API.

Основні ідеї API-first дизайну полягають у тому, що API створюються для їхніх користувачів, які є розробниками, і що API слід розглядати як незалежні сутності. API-first наголошує на тому, що дизайн API вимагає багато уваги, оскільки створення хорошого досвіду розробника є пріоритетом. На початку розробки створення хорошого дизайну починається з ігнорування існуючих систем, якщо такі є, та зосередження на тому, що потрібно від API.

Існують три принципи «API first»:

- "Ваш API - це перший користувацький інтерфейс вашої програми."
- «Спочатку ваш API, а потім реалізація».
- "Ваш API описаний (і, можливо, навіть сам по собі описовий)."

API-орієнтований дизайн – це ефективний спосіб створити API, з якими приємно працювати. Цей процес спонукає розробника правильно розробляти та тестувати власний API, що робить його простішим та зрозумілішим для розробників, і, швидше за все, він буде максимально придатним для мети, для якої він потрібен. Створення зручних для розробників API здійснюється не лише заради досвіду розробників, але й має великі переваги для компанії, яка застосовує цей підхід.

Не існує стандартизованого процесу API-first, оскільки його можна налаштувати відповідно до власних процедур компанії, оскільки єдиним важливим спільним фактором є створення специфікації API перед впровадженням. Однак відсутність детальних посібників значно ускладнює розуміння того, як процес проектування API-first працює на практиці, та початок його використання у власній роботі. У цьому розділі представлено процес, у шести кроках API-first, який доповнено інформацією про ці кроки з інших джерел, орієнтоване на API, починається з планування. Етап планування повинен давати відповіді на питання « чому » і «хто». Чому потрібен цей API та хто є його зацікавленими сторонами та споживачами ?.

Створення користувацьких історій – це хороший метод, який можна використовувати на початку розробки будь-яких API. Користувацькі історії допомагають людям , які створюють API, з'ясувати бізнес-вимоги API та отримати чітке уявлення про те, що розробляється. Вони також визначають засоби для повідомлення плану іншим зацікавленим сторонам через них.

Крок 2: Розробка та перевірка

Цей крок охоплює проектування API відповідно до плану з кроку 1. На проектування API слід витратити кілька годин. Рекомендується, щоб вже на цьому етапі був інструмент, який дозволяє тестувати запити API перед будь-

якими впровадженнями.

Крок 3: Зафіксуйте специфікацію API

На цьому етапі специфікація API створюється відповідно до проєкту та має бути зафіксована, щоб її можна було впровадити та показати як остаточний API для всіх учасників розробки, включаючи зацікавлених сторін та розробників. Рекомендується, щоб специфікація залишалася незмінною принаймні кілька місяців.

Крок 4: Тестування

На цьому етапі тестується API. Його функціональність, узгодженість та досвід розробника слід враховувати під час проведення тестів.

Крок 5: Впровадження

Впроваджуйте API, щоб зробити його доступним для зацікавлених сторін.

Крок 6: Керування та залучення

На цьому етапі API розгортається, і клієнти повинні мати можливість надати відгук про нього.

Багато експертів з цього питання рекомендують метод API-first, оскільки його використання має різні переваги. Переваги методу впливають на різні аспекти розробки та бізнесу.

API багаторазового використання, що орієнтований на API, має на меті охопити кожну необхідну функціональність у застосунку за допомогою виклику API. Дотримання цього принципу призводить до двох позитивних результатів: можливості використовувати ті самі виклики API у веб-, мобільній та планшетній розробці для одного застосунку, а також повторно використовувати їх в інших. API, що можна використовувати повторно, можуть зменшити навантаження на розробників.

Пояснюється переваги повторного використання API у різних додатках у своїй статті « Чому фраза «API-first» має бути в центрі кожного цифрового досвіду» в онлайн-виданні Digital Pulse, описуючи гіпотетичну ситуацію, коли банк хоче створити додаток, який дозволяє людям надсилати гроші своїм родичам в іншій країні.

API, які забезпечують роботу цього [додатку], численні. Одним із них може бути API адресної книги, який пов’язує користувача з людьми, з якими він здійснює транзакції. Іншим може бути API обмінного курсу для конвертації однієї валюти в іншу. API може знадобитися для автентифікації клієнта, наприклад, шляхом перевірки відповідності його PIN-коду обліковому запису. І, нарешті, ще один API може переказати фактичні гроші.

Після опису того, які API потрібні для цього банківського застосунку, як банк може використовувати два з цих старих API в новому застосунку, який люди могли б використовувати для зняття грошей з банкомата в іноземній країні.

Документація API

Проектування з урахуванням API починається з написання визначення API у стандартизованому форматі. Важливою частиною API-first є те, що створення визначення API також створює для нього документацію. Визначення API може бути саме по собі документацією; однак існує також багато інструментів для автоматичного створення більш зрозумілої документації відповідно до визначень API.

Узгодженість API

API-first допомагає створювати узгоджені API по всій компанії. Узгодженість API означає, що їхня структура та документація відповідають однаковим правилам. Усі API компанії можна створити, попередньо створивши визначення API та дотримуючись однакових стандартів і практик. Повторне використання, планування, документування та написання визначень API впливають на узгодженість. Узгодженість важлива, оскільки використання API легше вивчити, якщо вони схожі один на одного.

Створення кращого досвіду для розробників

розробника є одним із ключових аспектів, на які впливає API-орієнтований дизайн. Це сукупність багатьох переваг API-орієнтованого дизайну, які разом створюють хороший DX. Коротше кажучи, досвід розробника в контексті API означає, наскільки легко розробляти програми на основі API.

Оскільки API розробляються та тестуються з великою ретельністю

протягом усього процесу API-first-design, це впливає на досвід розробників API. Розробка застосунку на основі API з хорошою DX-підтримкою може заощадити багато часу під час розробки, і такий API, швидше за все, стане широко використовуваним, отримає хороші відгуки від розробників і спонукатиме їх використовувати його протягом тривалішого часу.

Зменшення залежностей від відкладення між командами

розробників, відповідальним за розробку різних частин проєкту, не потрібно чекати, поки один один завершить, почне або продовжить роботу. Підхід API-first дозволяє командам фронтенду, бекенду та тестування працювати одночасно після створення визначення API та макету API, як показано на рисунку 2.1. Фронтенд-розробники можуть створювати свої застосунки на основі макету API, поки бекенд-команда(и) створює(ють) свою реалізацію. Це також стосується роботи з різними клієнтами, такими як настільні та мобільні реалізації, які використовують один і той самий API.

Зробіть розробку та доставку швидшими

Процес розробки пришвидшується завдяки раніше згаданій паралельній розробці та можливості повторного використання. З початком використання методу API-first розробка може бути повільнішою, ніж розробка без ретельного проектування. Потрібен час, щоб навчитися проектувати API, створювати визначення API та використовувати метод API-first, але в довгостроковій перспективі це економить час.

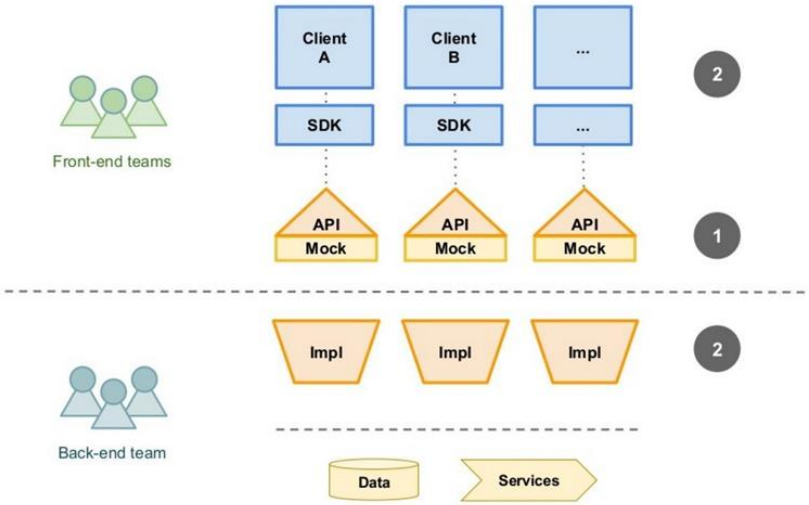

Рисунок 2.1 - Різні команди розробників, які одночасно працюють з імітованими API

Співпраця

Підхід, що орієнтований на API, дозволяє отримувати відгуки та коментарі щодо API, що розробляється, на дуже ранній стадії. Визначення API може і повинно бути надано зацікавленим сторонам з самого початку, щоб зробити дизайн якомога кращим. Це дає змогу отримати цінний відгук ще до написання будь-якого коду, а зміни до API легше вносити та витрачати менше ресурсів.

Оскільки визначення API є зрозумілим для людини та може бути представлене дуже простим, візуальним способом за допомогою спеціального інструменту, розуміння визначення API не вимагає технічних навичок, таких як знання програмування. Це робить визначення API доступним для всіх зацікавлених сторін.

Заохочує до роздумів про минулі застарілі системи

Мета API-first-проектування полягає в тому, щоб розпочати проектування без урахування застарілих систем на початку. Це призведе до того, що API будуть розроблені для фактичної потреби, яку вони повинні задовольнити, а не для того, що здається можливим з урахуванням застарілих систем. Застарілі системи повинні стати фактором у рішеннях щодо проектування лише після того,

як буде зроблено початковий проєкт, і буде вирішено, чи можна скомпрометувати стару систему чи новий проєкт.

API-First відповідає економіці API

Як перевага сама по собі , а також причина інших переваг, перелічених вище, API-first – це гарний спосіб розробки API, які стають частиною економіки API. Щоб досягти успіху в економіці API, компанії потрібні добре розроблені , швидко впроваджені та задокументовані API, і саме цього прагне API-first дизайн.

2.2 Методи визначення артефактів та побудови API-таксономії

У цьому дослідженні наша мета полягає в тому, щоб надати контент та артефакти, що розширюють знання та життєздатність API-First Design як методу програмної інженерії для проектування та архітектури розподілених програмних систем на основі мікросервісів. Розуміючи, що дотримання чітко визначеного методу покращує реплікацію досліджень та достовірність, та дотримуючись рекомендацій, цей розділ надає прозорий опис методів, що використовуються для створення таксономії, що підтримує артефакти, призначені для покращення розуміння та життєздатності API-FD та його застосування до проектування та архітектури мікросервісів.

API-FD, новаторська концепція, що орієнтована на галузь, спирається на врахування точок зору, досвіду та знань наших колег по галузі. Засноване в, використання традиційної та «сірої» літератури як внеску в дослідницькі теми, орієнтовані на галузь, враховує думку галузі та зменшує академічну упередженість. Таким чином, це дослідження спирається на галузевий досвід та академічні дослідження, зібрані в наших попередніх роботах [1, 3], як внесок у побудову таксономії та її допоміжних артефактів. З цією метою ми визначаємо групу учасників дискусії, з якими ми працюємо над удосконаленням та наданням нашого остаточного набору артефактів.

Хоча таксономія є важливою для розширення знань про API-FD, початковий рівень розвитку цієї теми також передбачає потребу в додаткових артефактах, які встановлюють базову основу знань для API-FD. Таким чином, окрім опису методу самої таксономії, у цьому розділі описано методи, що використовуються для створення важливих допоміжних артефактів. Зображено на рисунку 2.2, перш ніж розробляти саму таксономію, ми прагнемо надати словник концепцій, пов'язаних з API-FD, дослідження користувачів у формі діаграми варіантів використання та запропоноване формальне визначення методу проектування для підтримки створення таксономії.

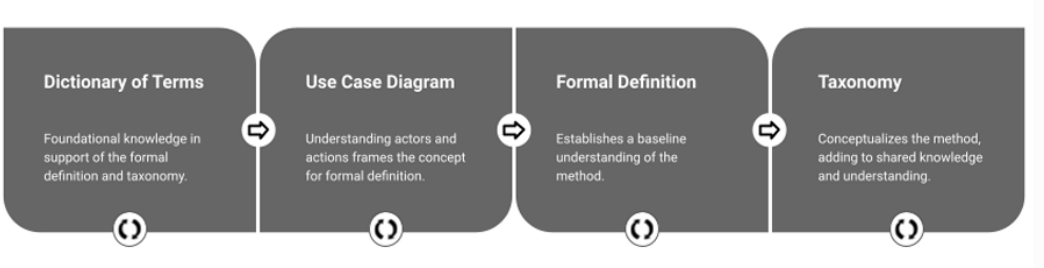

Рисунок 2.2 - Таксономія та супутні артефакти

У нашому дослідженні ми визначили необхідність розробки цих артефактів та розглядати кожен з них як будівельний блок для створення наступного артефакту. Іншими словами, знання, отримані при визначенні кожного артефакту, вбудовуються у створення наступних артефактів. Також, як показано на рисунку 2.2, ми ітеративно розробляємо ці артефакти, оскільки кожен з них закладає основу для майбутніх результатів. Нова інформація, отримана з кожного артефакту, також може бути використана для попередніх артефактів у ланцюжку. Щоб підвищити актуальність нашої роботи та як важливий крок у DSR, кожен артефакт проходить ітеративний огляд, оскільки ми оцінюємо та перевіряємо його з нашими колегами по галузі. Відкриття, зроблені з кожним артефактом, можуть вимагати внесення змін до інших артефактів. Процес зворотного зв'язку для формалізації допоміжних артефактів визначено в алгоритмі. Як показано, базова лінія для кожного результату або

артефакту створюється та переглядається групою досвідчених інженерів-програмістів та технологів. Зворотній зв'язок враховується, до артефакту вносяться зміни, збираються відгуки, і процес триває до тих пір, поки всі відгуки панелі не будуть враховані, і після завершення алгоритму не буде зроблено висновок, що кожен артефакт вважається гідним комунікації та публікації.

Попередні дослідження виявили можливості для розробки спільної мови, стандартного визначення та таксономії, які допоможуть спільнотам розробників програмного забезпечення та дослідників описувати, розуміти та впроваджувати API-First Design як життєздатну модель для проектування розподілених корпоративних програмних систем на основі мікросервісів [1, 3]. Як обговорювалося це дослідження має на меті розширити базу знань в академічних та галузевих спільнотах шляхом застосування DSR та розробки відповідних і цінних артефактів, що підтримують API-FD. Дослідження також прагне максимізувати релевантність артефактів, що підтримують API-FD, для програмної інженерії та досліджень у промисловості та академічних колах. З цією метою, крім того, це дослідження застосовує DSR та його ітеративні процеси для створення активів та власних заходів зворотного зв'язку та оцінки артефактів; це дослідження використовує білу та сіру літературу, узагальнену та агреговану в [1] та [3], як вхідні дані для цього проектування артефактів. Створення кожного з артефактів, представлених у цьому розділі, показано на рисунку 2.2 відповідно до алгоритму. Важливо наголосити, що для підвищення актуальності для галузі та зменшення академічної упередженості всі створені та обговорювані тут артефакти слід оцінювати як частину їхнього життєвого циклу розробки.

Як описано та узагальнено кожен артефакт, створений для підтримки API-FD, підлягає ретельній оцінці. Як обговорювалося наш підхід до тестування та оцінки є емпіричним, оскільки ми розробили набір анкет, призначених для збору якісних відгуків від групи наших колег по галузі. Містить відповідну інформацію про кожного з наших учасників панелі, включаючи призначений ідентифікатор для зручності використання далі в цьому розділі, поточну посаду

та роки досвіду (YOE) на посаді в галузі технологій. Також, як показано в таблиці, середній час роботи групи в цій галузі на момент написання становив 13,22 роки.

2.3 Аналіз вимог користувачів та моделювання варіантів використання (Use Case)

Окрім узагальнених публікацій [1] та [3], а також ключових слів і фраз [3], важливо структурувати наше дослідження з розумінням аудиторії, цільових споживачів і користувачів API-FD. [3] надає базове розуміння аудиторії та учасників за допомогою завершеного аналізу PICO (популяція, втручання, порівняння, результати), рекомендованого для досліджень програмної інженерії на основі доказів. Як описано в [3], популяція, пов'язана з API-First Design, включає досвідчених розробників програмного забезпечення, дизайнерів та архітекторів, які займають посади, здатні впливати на технічну реалізацію та зміни, а також осіб, здатних впливати на варіанти використання для створення бізнес-моделі, орієнтованої на API. Крім того, вимогою для розробки нашої таксономії, яка буде обговорена пізніше в цьому розділі, є розуміння цільових груп користувачів для нашої теми, що є важливим елементом.

Ітерація 1

Завдяки аналізу PICO, який сформував нашу базу знань, ми створили діаграму варіантів використання на основі UML, рисунок 2.3. що зображує ймовірних учасників, пов'язаних з API-First Design. На рисунку показано, що API-FD орієнтований на технічних та програмно-підкованих учасників, включаючи архітекторів програмного забезпечення, розробників та фахівців із забезпечення якості; сторони API-FD також включають бізнес-орієнтованих учасників, зокрема керівників, відповідальних за зростання, маркетинг та продажі.

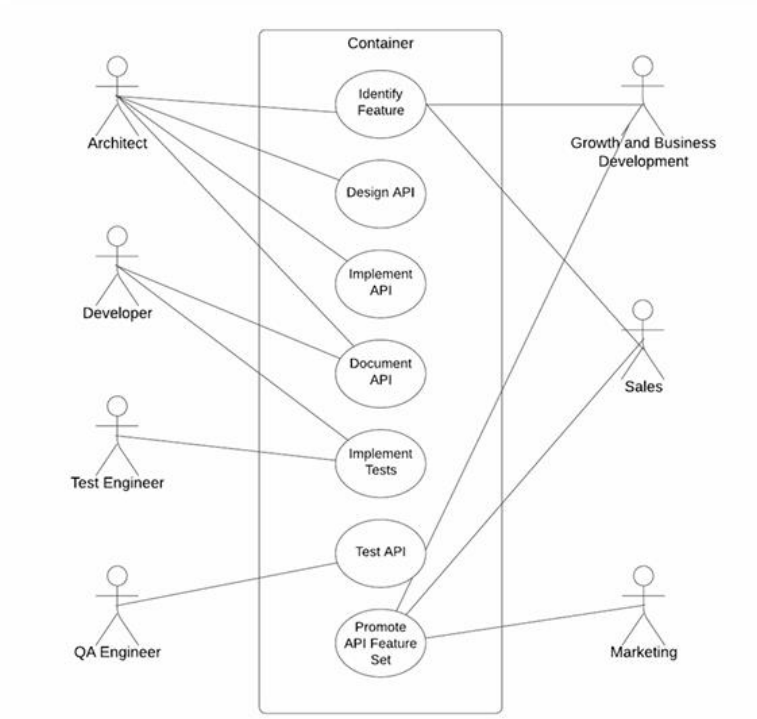

Рисунок 2.3 - Діаграма варіантів використання API-FD, ітерація 1

В опитуванні щодо відгуків щодо запропонованої нами діаграми варіантів використання для API-First Design було зібрано різні висновки, які можна коротко описати наступним чином:

- Покращення взаємозв'язку між акторами. Учасники наголосили на необхідності безпосередньої участі розробників у діях «Впровадження» та «Проектування API». Вони також обговорили необхідність того, щоб розробники розділяли відповідальність за тестування API з членами команди тестування та інженерії якості. Визначивши, що архітектори програмного забезпечення зосереджуються на проектуванні вищого рівня, а не на деталях реалізації, учасники запропонували виключити участь архітектора з «Впровадження API».
- Додаткові актори та дії. Учасники запропонували додати нових акторів: «Менеджера продукту», «Дизайнера» та «Споживачів API» (кінцевих користувачів API). Вони також запропонували додати нові дії, які завершують життєвий цикл API, включаючи огляд дизайну.
- Спрощення. Деякі учасники запропонували дослідити чіткіші та

організованіші схеми, уточнити ролі та групувати їх, наприклад, за акторами. Інші запропонували об’єднати надлишкові ролі та акторів, включаючи інженерів з тестування та контролю якості, а також відділу продажів та розвитку бізнесу.

- Життєвий цикл API. Деякі учасники закликали до більш чіткого представлення життєвого циклу API з потоком дій, який показує, як різні етапи пов'язані між собою та враховують такі аспекти, як безпека, конфіденційність та моніторинг продуктивності.

Ітерація 2

Комплексні рекомендації щодо змін призвели нас до другої ітерації дослідження користувачів та діаграми варіантів використання. Щоб уточнити та спростити діаграму, ми дотримувалися рекомендацій учасників та згрупували типи користувачів у три основні категорії. Як показано на рисунку 2.4, категорії користувачів, що застосовуються до API-FD, включають тих, хто виконує роль технічного впровадження або розробника програмного забезпечення, тих, хто відповідає за проектування продукту або виконує роль дизайнера продукту, тих, хто відповідає за розвиток та зростання бізнесу (бізнес-розробник), та нашого кінцевого користувача або споживача продукту. Друга ітерація діаграми варіантів використання також визначає ролі та обов'язки у двох окремих функціях: розвиток бізнесу та технічна реалізація. Кожна функція охоплює пов'язані з нею дії, такі як визначення, просування, проектування та реалізація API. Спрямовані ребра також демонструють ітеративний характер проектування API та зв'язки між визначеними діями. Наприклад, коли бізнес-розробник просуває API та його функції, він також може ідентифікувати додаткові API; коли інженер-програміст проектує та документує API, реалізація може зазнавати впливу та оновлюватися.

Як обговорювалося і подібно до оцінювання Ітерації 1, ми провели друге опитування через Google Forms, щоб зібрати відгуки щодо запропонованого дослідження користувачів та діаграми варіантів використання від наших галузевих партнерів, запит на яке показано на рисунку 2.4. Відгуки кожного учасника панелі включено та зведено в наступному списку.

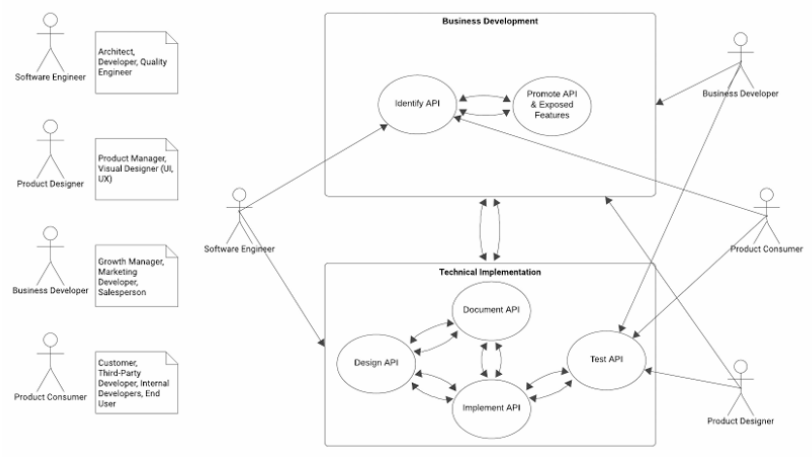

Рисунок 2.4 - Діаграма варіантів використання API-FD, ітерація 2

- Багато учасників відзначили покращення порівняно з попередніми версіями, зокрема консолідацію ролей для відображення спільної відповідальності та організаційних відмінностей.
- Учасники також зазначили, що діаграма ефективно передає ітеративний, нелінійний характер дизайну API, коли дії впливають одна на одну.
- Навіть з огляду на позитивні відгуки щодо успішної демонстрації ітеративного характеру діяльності, учасники запропонували створити циклічний погляд на діяльність з розвитку.
- Розгляньте можливість додавання вузла в категорії «Розвиток бізнесу» для представлення цілісного використання.
- Косметичні пропозиції включали заміну подвійних двонаправлених стрілок одним різнонаправленим ребром, видалення ліній, що перекриваються, та додавання кольору для візуальної привабливості та чіткості.

Фінальна ітерація

Фінальна ітерація аналізу користувачів та діаграми варіантів використання представлені на рисунку 2.5. Ця фінальна та запропонована версія артефакту на підтримку API-FD містить відповідні відгуки від наших учасників, зібрані на основі перших двох ітерацій діаграми. Найбільш помітними є категорії користувачів, що охоплюють різноманітні ролі (наприклад, інженер-програміст

охоплює високорівневих архітекторів, розробників та інженерів з якості), комплексний набір дій, пов'язаних з API-FD, організація дій, що демонструє ітеративний та циклічний характер проектування програмного забезпечення, спрямовані межі між діями та кольори, що допомагають швидко зрозуміти саму діаграму.

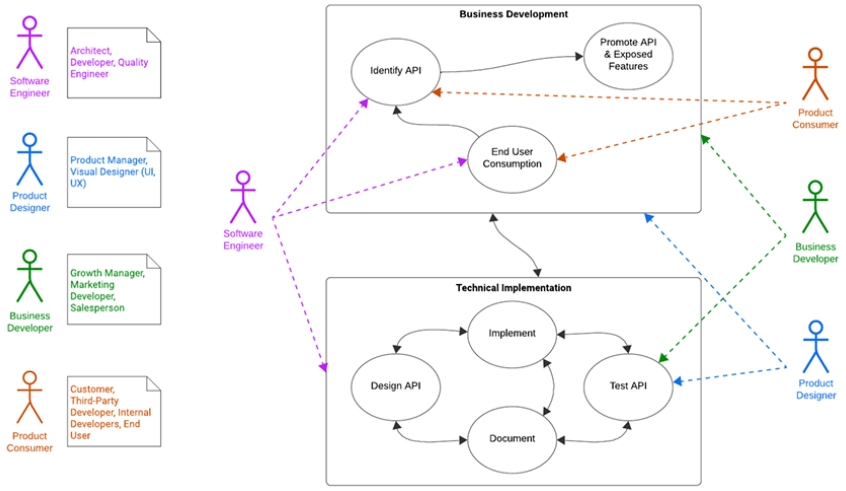

Рисунок 2.5 - Діаграма варіантів використання API-FD, остаточна версія

Як описано та як було зазначено під час нашого дослідження користувачів та діаграми варіантів використання, ми співпрацювали з нашою групою експертів, щоб визначити покращення та доповнення до запропонованого словника термінів. Як і в дослідженні користувачів, ми продовжували ітерації та збирати відгуки, доки всі учасники групи не були задоволені словником термінів.

Нижче наведено короткий виклад пропозицій щодо покращення наших визначень, що стосуються API-First Design та інших.

- Визначення API. Наші галузеві колеги запропонували включити визначення самого API. Їхні пропозиції також включають врахування переваг чітко визначеного API та культурно визначених стандартів API, таких як акцент на дизайні API, що підходить для кількох клієнтів.
- Уточнення культури API та фокусу на API. Багато учасників дискусії висловили необхідність уточнення різниці між цими термінами. Деякі вважають,

що культура API охоплює фокус на API, і що API є фундаментальним елементом обох термінів, а API розглядається як сам продукт або як центральний елемент монетизації. Один з учасників запропонував змістити наше визначення культури API до вимірюваної та дієвої організаційної поведінки.

- Уточнити поняття «місткість» та «товар». Учасники дискусії запропонували розділити ці терміни та ґрунтовніше визначити обидва, пропонуючи уточнити, що цей товар передбачає потенціал монетизації, особливо у зовнішніх партнерствах. Пов’язана з цим пропозиція визначити API як продукт.
- Розширення словника. Багато наших учасників запропонували розширити визначення, включивши пов'язані концепції дизайну та архітектури, такі як сервісно-орієнтована архітектура (SOA), доменне-орієнтоване проектування (DDD), Обмежені контексти та багатошарова архітектура. Вони також запропонували додати терміни, пов'язані з API та його життєвим циклом, шляхом визначення Управління API, Зацікавлених сторін API, Версіонування API, Документація API, Монетизація API та Безпека API.
- Удосконалити визначення мікросервісної архітектури. Один з наших експертів з розробки програмного забезпечення запропонував, щоб наше визначення MSA включало контраст між монолітним та мікросервісним підходами до проектування систем, а також щоб ми розширили переваги меж мікросервісів. Ще одна пропозиція полягає у спрощенні розмови. Де це можливо, дозволити «сервіс» як взаємозамінний термін для «мікросервісу» та уточнити типи API у визначенні.

Визначені терміни та концепції базуються на двох ітераціях оцінки, проведеної нашими галузевими колегами. Зміни від першої запропонованої версії таблиці до цієї остаточної, третьої ітерації включають уточнення таких термінів, як культура API та фокус API, додавання термінів, що покращують контекст, включаючи API та обмежений контекст, а також додаткові терміни з програмної інженерії, як пов'язані з API-FD, так і не схожі, такі як MSA та монолітна архітектура. За підтримки DSR ми визначили розумну точку зупинки

для подальших ітерацій на основі зібраних комплексних відгуків та зменшення кількості відгуків, що сприяють розвитку знань та артефактів.

2.4 Висновки по розділу

У другому розділі було досліджено методи та інструменти проєктування API-first систем. Розглянуто методологію API-орієнтованого проєктування, що дозволяє стандартизувати процес створення інтерфейсів програмних систем. Проаналізовано підходи до визначення артефактів проєктування та побудови API-таксономії для структурованої організації сервісів. Також досліджено методи аналізу вимог користувачів і моделювання варіантів використання (Use Case), що забезпечують узгодженість між бізнес-вимогами та технічною реалізацією. У результаті визначено ефективні підходи до створення якісних і узгоджених API-систем.

РОЗДІЛ 3. РОЗРОБКА, ІНТЕГРАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ API-FIRST РІШЕНЬ

3.1 Інструменти та стандарти підтримки API-first підходу

Однією з головних цілей цієї роботи є представлення, випробування та оцінка відповідних інструментів і стандартів, які допомагають ефективно створювати API з використанням підходу API-first. Наступні стандарти та інструменти, представлені в цьому розділі, використовуються в демонстрації API-first проектування в розділі . Вони обрані відповідно до обсягу роботи та є інструментами.

API-орієнтований дизайн підкреслює, що розробка API вимагає часу та зусиль, і зрештою кінцевий результат вартий витрачених ресурсів. Однак процес API-орієнтованого дизайну можна зробити ефективнішим за допомогою правильних інструментів, що заощаджує час без шкоди для якості.

Існують інструменти та стандарти, призначені для всіх етапів розробки API: проектування, створення, тестування та розгортання API. Використання інструментів допоможе процесу проектування та зробить використання методу «API first» більш доступним, а також зменшить ймовірність помилок під час цього процесу.

Використовуючи правильні інструменти, можна розробити та протестувати функціональність нового API без будь-якого програмування та залежностей від певних мов програмування, платформ чи програм. Інструменти автоматизують процес і спрощують його ітерації, що забезпечує найкращі результати.

OpenAPI (OAS) – це стандарт для визначення RESTful API таким чином, щоб вони були зрозумілими як для людей, так і для машин. Файли, створені в цьому форматі, можуть бути прочитані та використані багатьма інструментами, які допомагають проектувати, створювати та керувати API, а також надають документацію для розробників. OpenAPI має дві версії: OpenAPI 2.0 та останню OpenAPI 3.0, випущену в 2017 році. Специфікація належить ініціативі OpenAPI . Специфікацію OpenAPI можна записати у форматі YAML або JSON (Базова структура | Swagger Nd).

У цьому дослідженні для розробки демонстраційного API використовується OpenAPI 2.0, оскільки на момент проведення цього дослідження Azure API Management не підтримує версію 3.0. Microsoft зараз додає підтримку OpenAPI 3.0 до Azure.

Swagger - це набір інструментів розробника API для кожного етапу розробки API, від проектування до розгортання. Swagger є частиною SmartBear Software, яка спеціалізується на інструментах розробки та пропонує інструменти з відкритим кодом, безкоштовні та комерційні інструменти. Swagger є

оригінальним творцем специфікацій OpenAPI , і всі продукти Swagger створені для підтримки OpenAPI . (Про нас | Swagger Nd)

У цьому дослідженні було використано два продукти Swagger: *Swagger Editor* та *Swagger Codegen* .

Swagger Editor - це інструмент для редагування специфікацій OpenAPI з відкритим кодом. За його допомогою можна писати специфікації Open API для API, отримуючи миттєвий зворотний зв'язок та візуальний опис API. Swagger Editor підтримує написання визначень OpenAPI як у форматі YAML, так і JSON.

Редактор Swagger доступний для всіх середовищ розробки та може використовуватися як локально, так і онлайн. Це інструмент редактора, який динамічно відображає візуальну документацію збоку для поточної специфікації, як показано на рисунку 3.1. Він також видає повідомлення про помилку, якщо є синтаксична помилка у специфікації OpenAPI , яка пишеться, як показано на рисунку 3.2._Він також пропонує автозаповнення для швидшого написання та дозволяє швидко імпортувати та зберігати описи API, а також створювати серверні заглушки та клієнтські файли різними мовами програмування.)

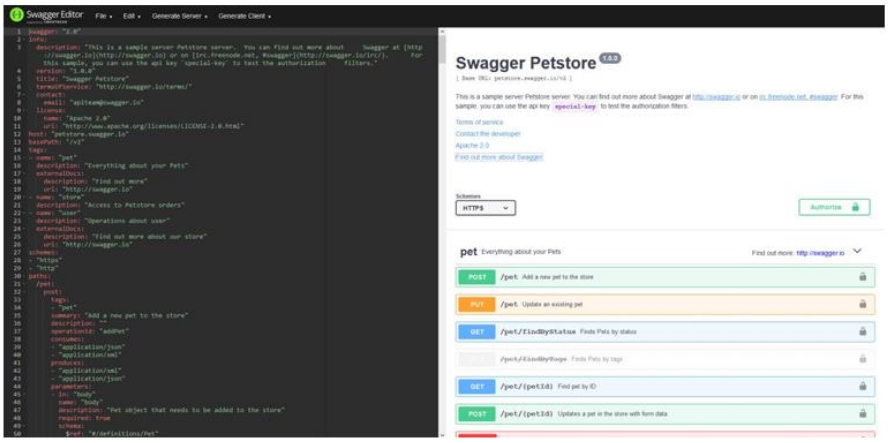

Рисунок 3.1 - Веб-інтерфейс Swagger Editor із прикладом API

Swagger Codegen - це інструмент з відкритим кодом , який використовує специфікації OpenAPI для створення клієнтських SDK та серверних кодових заготовок. Його можна завантажити з репозиторію GitHub. Swagger Codegen може генерувати код більш ніж 40 мовами програмування. (Завантажити

Swagger Codegen.ND ; Swagger Codegen.ND)

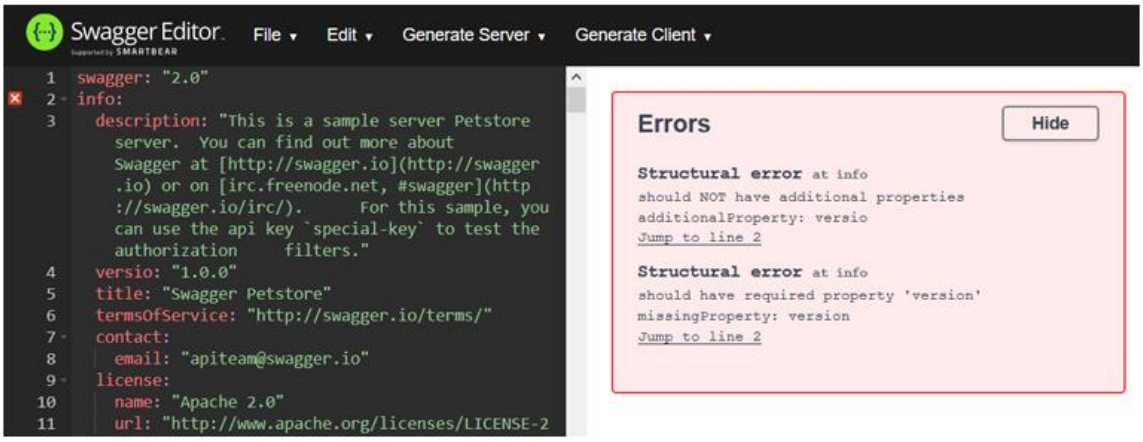

Рисунок 3.2 - Приклад повідомлення про помилку в Swagger Editor

Swagger Codegen використовується в локальному середовищі , а доступ до його операцій здійснюється через командний рядок.

Azure API Management - це хмарна платформа керування API від Microsoft. Вона є частиною Microsoft Azure, яка має багато хмарних сервісів з різних категорій розробки програмного забезпечення. Azure API Management - це місце, де можна розгортати API. (Керування API: створення шлюзів API Nd; Що таке Azure - Microsoft Cloud Services Nd). Azure API Management складається з трьох продуктів: шлюз API, портал Azure та портал розробника .

Шлюз API – це кінцева точка, яка обробляє трафік, пов’язаний з API.

Портал Azure - це адміністративна платформа, яку можна використовувати для керування всіма видами програм в одному місці. У контексті цієї роботи Azure використовується для налаштування API. На порталі можна налаштувати екземпляр служби керування API. За допомогою цієї служби можна визначати та керувати схемами API. (Microsoft Azure Portal Nd)

Портал розробника — це окрема сторінка для служби керування API з корисними функціями для розробників. Вона містить документацію для API, розгорнутих у цій службі керування API, консоль для тестування API, систему керування обліковими записами та інформацію про аналітику API.

3.2 Розробка API та клієнтського застосунку на основі API-first підходу

У цьому розділі розглядається процес проектування та впровадження нового API, а також створення невеликої клієнтської програми, яка використовує API, за допомогою методу проектування API-first. Він базується на етапах процесу API-first, з деякими змінами, пов'язаними з отриманням більш цінних результатів за допомогою управління Azure API. Цей процес є кінцевим результатом після кількох ітерацій його вдосконалення.

Предметом нового API є *відсутність учнів у школі,* яка додається до даних про окремих учнів, коли вони відсутні в школі протягом дня. API називатиметься API відсутності.

Вимоги до Absences API максимально прості, оскільки це демонстраційний API , який має бути зрозумілим, наприклад. Мета полягає в тому, щоб мати простий API, який не має більше кількох кінцевих точок API, і його вимоги вибираються з урахуванням цього.

Предметом API є додавання відсутності учнів. Авторизовані користувачі повинні мати можливість виконувати операції, пов’язані із *записами про відсутність,* які є інформацією про щоденну участь учня в школі. Ці операції включають перегляд відсутності різних учнів, додавання нових записів про відсутність та їх видалення.

Основні вимоги до Absences API з точки зору REST API полягають у тому, що має бути можливість *читати, додавати* та *видаляти* відсутність учнів. Крім того, інформація про учнів має бути редагуємою, а також має бути можливість додавати та видаляти окремих учнів. Записи про відсутність додаються на цілий день, і це єдина інформація, необхідна для визначення відвідуваності учня. Відвідування школи не вимагає жодних записів.

Користувачами Absences API є *вчителі, батьки* та *адміністратори,* тобто особи, відповідальні за ведення обліку відсутності та інформації про учнів.

Інформація, до якої має дістатися API, це *студенти* та їхня *відсутність.*

Усі записи про відсутність повинні належати студенту. Студенти повинні вказати загальну інформацію, таку як їхнє ім'я та індивідуальний ідентифікатор. Відсутність повинна містити дату та причину відсутності.

Перше, що потрібно зробити на етапі планування, це визначити, чого має досягти API та як він може відповідати бізнес-вимогам. Спочатку план має бути якомога простішим, оскільки його можна розширити пізніше.

Рекомендований підхід полягає у створенні користувацьких історій на основі вимог. Під час створення користувацьких історій для API важливо зосередитися на користувачах API, а не на

Сам API. Наприклад, «як API, я хочу мати можливість отримувати інформацію про відсутність студента» – це не правильний спосіб створення історій користувачів; вони повинні починатися з «як певний тип користувача». Історії користувачів допомагають розробникам зрозуміти, як користувачі використовуватимуть продукт, і приймати кращі дизайнерські рішення на основі історій користувачів.

Історії користувачів для цього API:

- Як вчитель, я хочу мати можливість додавати відсутність учня на певну дату, щоб відстежувати відсутність кожного учня.
- Як вчитель, я хочу мати можливість видаляти окремі пропуски учня, щоб я міг видаляти пропуски, зафіксовані за помилками.
- Як батько, я хочу мати можливість додавати відсутність для моєї дитини, щоб вчитель знав, що моя дитина не прийде до школи в певний день.
- Як адміністратор, я хочу мати можливість додавати та видаляти студентів, а також змінювати їхню інформацію, щоб інформація про поточних студентів була актуальною.

Починаючи з найпростішого варіанту, потрібен REST API для *студентів* , які можуть реєструвати *пропуски.* Вимога полягає в тому, щоб мати лише одну відсутність протягом одного дня, щоб пропуски розрізнялися за датою.

API Absences має відповідати найкращим практикам REST API. Назви ресурсів (наприклад, student) повинні бути у множині, оскільки цей ресурс може

мати кілька екземплярів одного й того ж ресурсу. Окремі ресурси всередині шляху можна ідентифікувати за унікальним значенням, яке вони мають (наприклад, *studentId* у цьому випадку). Остаточна структура шляхів REST API записана (Рисунок 3.1) .

Path
/students
/students/{studentId}
/students/{studentId}/absences
/students/{studentId}/absences/{date}

Рисунок 3.1- Шляхи REST API для API «Відпустки»

На цій структурі базується визначення OpenAPI . Написання визначення API простіше та зрозуміліше, коли є письмовий план , якого слід дотримуватися, навіть якщо остаточне визначення може змінитися під час тестової фази. Після визначення структури REST API наступним кроком є вибір CRUD-операцій для кожного шляху. Повинні бути операції для виконання всіх необхідних викликів у застосунку; проте без надлишкових операцій. Надлишковою операцією в цьому API буде, наприклад, операція *видалення* кожної відсутності одного учня, оскільки видалення кожної відсутності учня не повинно бути можливим згідно з бізнес-вимогами API. CRUD-операції, обрані для Absences API, наведено на рисунку 3.2.

Path	Operations
students	post, get
students/{studentId}	get, put, delete
students/{studentId}/absences	post, get
students/{studentId}/absences/{date}	get, put, delete

Рисунок 3.2 - Операції для кожного шляху

Обрана структура та операції REST повинні забезпечувати можливість

коміту (коміту) раніше створених користувацьких історій. Тестування користувацьких історій:

- Як вчитель, я хочу мати можливість додавати відсутність до учня, щоб відстежувати відсутність кожного учня.

Можливо з *post:/students/{ studentId }/absences*

- Як вчитель, я хочу мати можливість видаляти окремі пропуски учня, щоб я міг видаляти пропуски, зафіксовані за помилками.

Можливо з *delete:/students/{ studentId }/absences/{date}Як батько, я хочу мати можливість додавати відсутність для моєї дитини, щоб вчитель знав, що моя дитина не прийде до школи в певний день.*

Можливо з *post:/students/{ studentId }/absences*

- Як адміністратор, я хочу мати можливість додавати та видаляти студентів, а також змінювати їхню інформацію, щоб інформація про поточних студентів була актуальною.

Можливо, за допомогою *post:/students, delete:/students/{ studentId }* та *put:/students/{ studentId }*

Вже на цьому етапі процесу проектування люди, які розробляють API, можуть повідомити план проектування цього API іншим зацікавленим сторонам, щоб отримати відгуки.

Коли необхідні на цьому етапі вимоги, структура та операції CRUD, визначені, наступним кроком у процесі API-first є створення специфікації OpenAPI на їх основі. У цьому випадку це робиться в онлайн-версії Swagger Editor. Онлайн-версію було обрано, оскільки вона здатна виконати вимоги до написання визначення API та не вимагає процесу встановлення нового програмного забезпечення.

Визначення OpenAPI має структуру, яка охоплює весь REST API. У цьому прикладі визначення написано у форматі YAML, і для сумісності з Azure APIM використовується старіша версія визначення, OpenAPI 2.0. У наступній частині показано кроки для створення визначення OpenAPI для Absences API та детальну інформацію про кожну частину.

1. Метадані

Метадані, необхідні для специфікації OpenAPI, – це версія OpenAPI , яка використовується у визначенні, та інформація про API, що охоплює його назву, версію та опис.

```
swagger: '2.0'
info:
  title: Absences
  version: '1.0'
  description: Absences for students API
```

2. Базова URL-адреса

Визначте хост, шляхи та схеми. URL-адреса хоста API тепер визначена за допомогою тексту-заповнювача, оскільки вона ще не існує, і її буде оновлено до визначення після запуску служби керування Azure API з фактичною URL-адресою.

```
host: placeholder
basePath: /api/1.0
schemes:
  - http
  - https
```

3. Шляхи

Кожна окрема кінцева точка API визначена окремо в розділі під назвою *шляхи.* Для кожної кінцевої точки вказані свої HTTP-методи. HTTP-методи мають додаткову інформацію про них, яка допомагає їх описати та розрізнити. Ця інформація є важливою частиною створення документації на основі визначень OpenAPI . У цьому прикладі шлях */ students* та його HTTP-методи *post* та *get* визначені наступним чином:

```
paths:
  /students:
    post:
      summary: Add a new student
      description: Add a new student
      operationId: addStudent
      consumes:
        - application/json
      produces:
        - application/json
    get:
      summary: Lists all students
      description: Lists all students
      operationId: listStudents
      produces:
        - application/json
```

4. Параметри

HTTP-методи можуть мати параметри, тобто дані, які можна передати чотирма різними способами: через URL-шлях, рядок запиту, заголовок або тіло. Параметр операції *post:/students* передається в тілі запиту та посилається на визначення під назвою Student, яке є спільним для всього API.

```
parameters:
    - name: "body"
      in: body
      schema:
          $ref: '#/definitions/Student'

        required: true
        description: Student that will be added
```

5. Відповіді

Відповіді на виклики API додаються до визначення кожного методу HTTP. Ось відповіді для функції *post* шляху */students* :

```
responses:
      '200':
          description: OK - student added
          schema:
            $ref: "#/definitions/Student"
      '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
```

6. Моделі

Моделі – це визначені структури даних, спільні для API.

```
definitions:
  Student:
    type: object
    properties:
      studentId:
        type: integer
        format: int64
        example: '1234567890'
      firstName:
        type: string
        example: Bob
      lastName:
        type: string
        example: Smith
    required:
      - studentId
```

Редактор Swagger візуально відображає структуру постійно та динамічно оновлює її, як показано на рисунку 3.3.

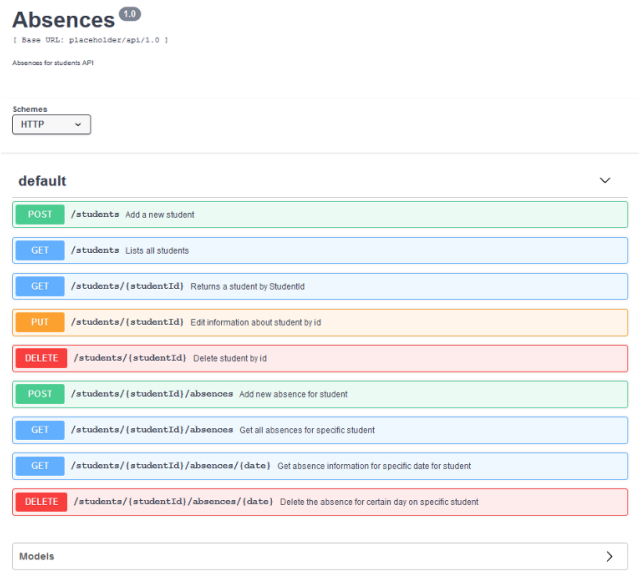

Рисунок 3.3 - Динамічна візуальна документація API «Absences» у Swagger Editor

Готове визначення API містить усі необхідні частини для визначення OpenAPI та виконано відповідно до раніше обраної структури REST, як показано на рисунку 3.2.

Завантажити специфікацію OpenAPI для керування API Azure

Коли специфікацію OpenAPI буде протестовано та готово до блокування, її можна буде завантажити через редактор Swagger. Azure API Management приймає лише формат JSON, тому файл визначення можна завантажити в цьому форматі, натиснувши «конвертувати та зберегти як JSON», як показано на рисунку 3.4.

Впровадження API в Azure

У цьому розділі розглядається впровадження API в Azure та обговорюються дії, які слід вжити в Azure для сприяння розробці з урахуванням принципів API.

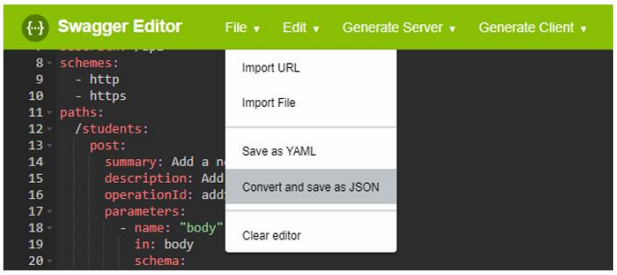

Рисунок 3.4 - Перетворення специфікації OpenAPI у формат JSON у Swagger Editor

Створення нового екземпляра служби керування API Azure

Для початку обслуговування API в Azure Portal потрібна служба API Management (скорочено APIM). Absences API буде додано до служби API Management, яка його розміщуватиме.

Новий сервіс APIM створюється шляхом переходу до розділу «Створити ресурс» на порталі Microsoft Azure та пошуку сервісу керування API на ринку Azure. Створення нового сервісу API вимагає додавання загальної інформації про новий сервіс, як описано на рисунку 3.5.

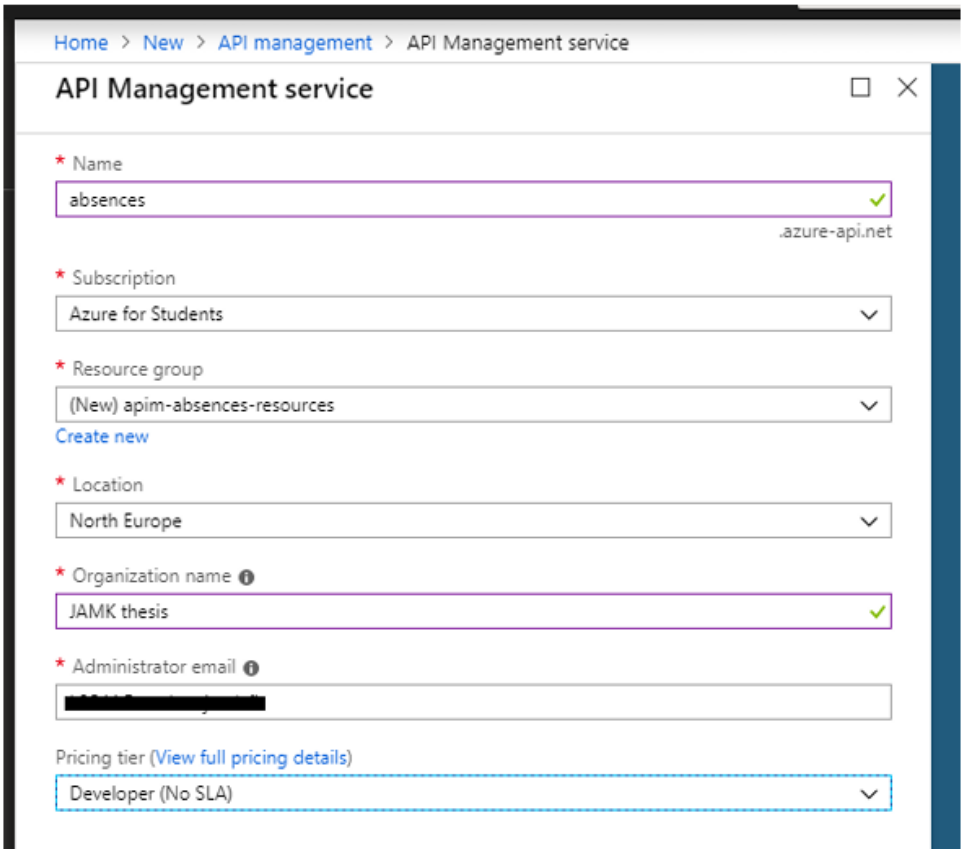

Рисунок 3.5 - Створення нової служби управління API

Імпорт специфікацій OpenAPI до Azure

Коли служба керування API створена та активована, туди можна додавати нові API. У цьому випадку специфікація OpenAPI, створена раніше, буде імпортована шляхом додавання нового API з опції специфікації OpenAPI , як показано на рисунку 3.6.

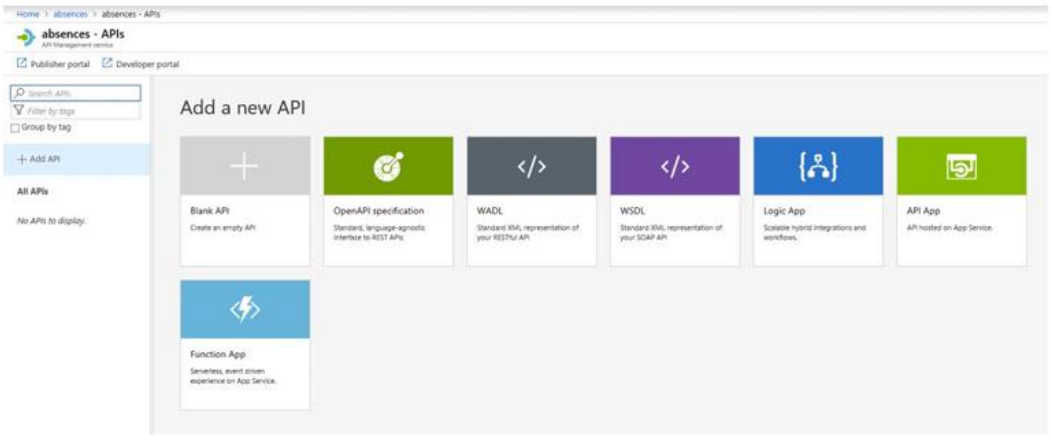

Рисунок 3.6 - Додавання нового API шляхом імпорту специфікації OpenAPI

Новому API потрібно надати таку інформацію, як назва API, як показано на рисунку 3.7. Вся інша інформація, необхідна для налаштування API, взята зі специфікації OpenAPI . Усі виклики API, шляхи та інша необхідна інформація автоматично додаються до служби керування API відповідно до специфікації.

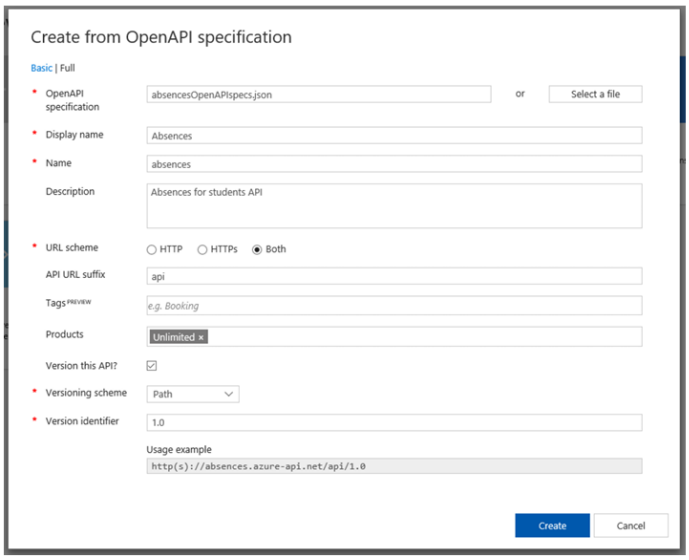

Рисунок 3.7 - Створення нового API на основі специфікації OpenAPI

Після створення API можна встановити прапорець «Потрібна підписка» , щоб використовувати ключ підписки для захисту доступу до API в керуванні Azure API, як показано на рисунку 3.8. Дійсний ключ підписки потрібен у будь-яких HTTP-запитах, що надсилаються до API, якщо цей параметр позначено.

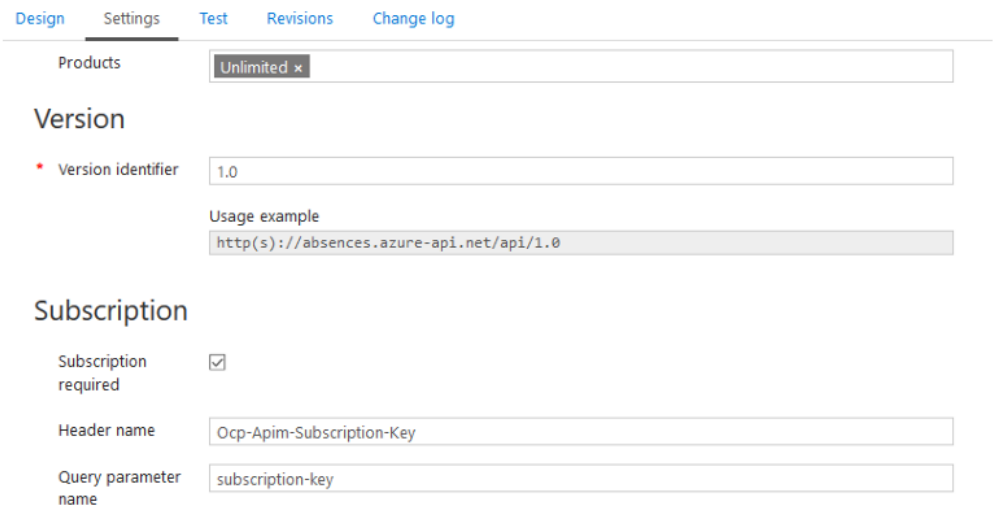

Рисунок 3.8 - Налаштування, що вимагають ключа підписки для доступу до API

Висмейування запитів на новий API

На даний момент немає бекенд-сервісів або баз даних, до яких API міг би підключитися, тому тестування нового API завжди повертає код відповіді "404 Не знайдено". У процесі API-first передбачається, що одразу після впровадження API в службі керування API не буде функціонувати бекенд. Щоб розпочати розробку з використанням API, наступним кроком є ввімкнення фіктивних відповідей керування API, які дозволяють отримувати відповіді на основі прикладів атрибутів, зазначених у специфікації OpenAPI .

Портал розробника

Тепер API можна перевірити на порталі розробника, який автоматично згенерував для нього документацію. Кожну операцію можна перевірити окремо на наявність інформації про них, подібно до редактора Swagger, як показано на рисунку 3.9.

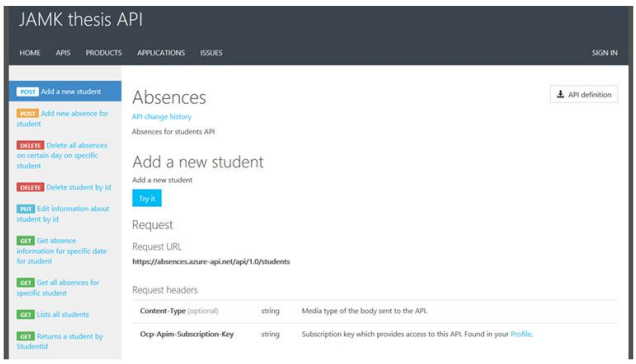

Рисунок 3.9 - Документація API «Відсутність на роботі» на порталі для розробників

Імітацію можна ввімкнути для всіх або лише для вибраних операцій, додавши нову політику зв'язку, як показано на рисунку 3.10. Можна вибрати, - який із успішних запитів буде повернуто на імітовані виклики. Для цього випадку було обрано відповідь "200 OK", тому тестування операцій завжди повертає код відповіді *200 із* прикладом даних з визначення API.

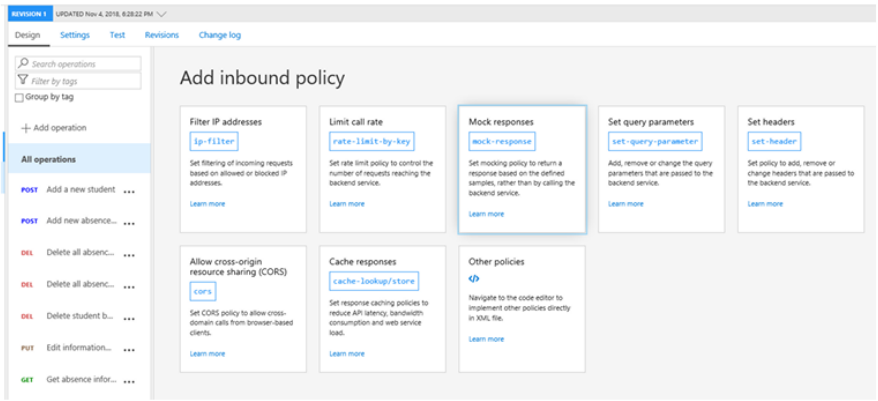

Рисунок 3.10 - Додавання фіктивної політики відповіді

Тестування фіктивних викликів API на порталі розробника

Виклики до змодельованого API можна протестувати на порталі розробника, як показано на рисунку 3.11.

Для доступу до фікованого ресурсу потрібна підписка. Можна або вручну надіслати ключ підписки в заголовку запиту, або стати передплатником через портал розробника, який має доступ до здійснення викликів API. Щоб стати

передплатником, потрібне схвалення адміністратора.

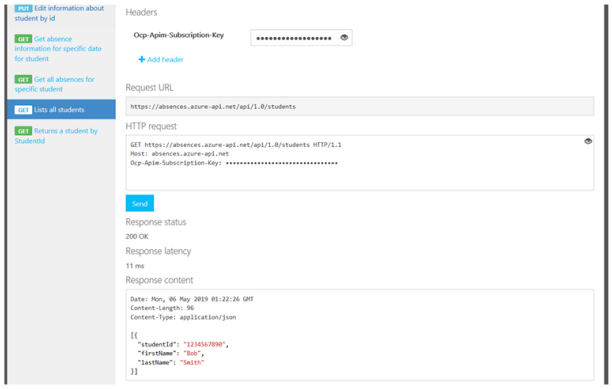

Рисунок 3.11 - Тестування виклику API для отримання списку студентів на порталі для розробників

Використання генератора коду Swagger для створення клієнта

У цьому розділі розглядається створення клієнта API з визначення OpenAPI за допомогою генератора коду Swagger. Це демо було створено в середовищі Windows 10 за допомогою Visual Studio 2017 мовою програмування C#.

Генерація клієнтського SDK за допомогою генератора коду Swagger

Swagger Codegen встановлюється локально шляхом завантаження JAR-файлу, який його містить. Цей файл є єдиним інтерфейсом для Swagger Codegen , а доступ до його операцій здійснюється через командний рядок. Для використання Swagger Codegen потрібна установка останньої версії Java.

Крок 1: Завантажте Swagger Codegen

Файл пакета Swagger Codegen .jar можна завантажити через Windows PowerShell за допомогою команди, показаної на рисунку 3.12. Тут він виконується в папці під назвою *swaggercodegen*, але файл може знаходитися будь-де.

```
PS D:\swaggercodegen> Invoke-WebRequest -OutFile swagger-codegen-cli.jar http://central.maven.org/maven2/io/swagger/swagger-codegen-cli/2.3.1/swagger-codegen-cli-2.3.1.jar
```

Рисунок 3.12 - Команда для завантаження пакета Swagger Codegen

Крок 2: Створення нового клієнтського SDK

Створення нового пакета Client SDK, який відповідає потребам цієї демонстрації, можна виконати за допомогою команди, показаної на рисунку 15. Команда посилається на раніше завантажений пакет Swagger Codegen .jar та специфікацію OpenAPI для Absences API у форматі .json , яку було завантажено з редактора Swagger. Вони обидва знаходяться в одній папці.

```
D:\swaggercodegen>java -jar swagger-codegen-cli.jar generate -l csharp -i absencesOpenAPIspecs.json -o absencesAPIsdk  -
DpackageName=AbsencesAPI.Sdk
```

Рисунок 3.13 - Створення клієнта на основі OpenAPI

Базова команда для запуску Swagger Codegen та створення нового клієнта - *java -jar swagger-codegen-cli.jar generate.* Після неї команді потрібні опції, що вказують мову програмування та специфікацію OpenAPI, що використовуються, а також шлях до папки, в якій вона створюється. Команда також використовує необов'язковий рядок для назви пакета, відмінної від його назви за замовчуванням. Ці опції:

1. *-l csharp :* Використання мови C#
2. *- я absencesOpenAPIspecs.json :* Шлях до специфікації OpenAPI
3. *-o:* Визначити шлях до папки
4. *- DpackageName = AbsencesAPI.Sdk :* Найменування пакета Absences API.Sdk

За замовчуванням команда для створення нового клієнтського SDK створює все, що доступно для генерації Swagger Codegen зі специфікацій API. Це включає всі API, моделі та тести.

Після виконання команди має бути три папки: *.swagger- codegen , docs* та *src ,* а також дев'ять файлів, таких як файл рішення Visual Studio, як показано на рисунку 3.14.

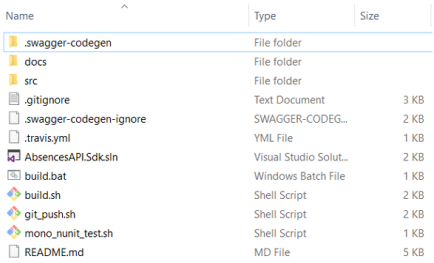

Рисунок 3.14 - Зміст, згенерований Swagger Codegen

Крок 3: Відкрийте згенерований SDK у Visual Studio

Файл рішення Visual Studio для нового клієнтського SDK можна відкрити у Visual Studio.

1. Після відкриття рішення рекомендується клацнути на ньому та натиснути «Відновити пакети NuGet», щоб уникнути проблем, спричинених відсутніми пакетами NuGet.

2. Клієнти, згенеровані Swagger Codegen, використовують бібліотеку RestSharp . Для RestSharp властивість Copy Local має бути встановлена на true з References у AbsencesAPI.Sdk , інакше збірка завершиться невдачею.

3. Створити рішення.

Деталі про те, що створив Swagger Codegen

Swagger Codegen генерує рішення, яке має структуру папок, як показано на рисунку 3.15.

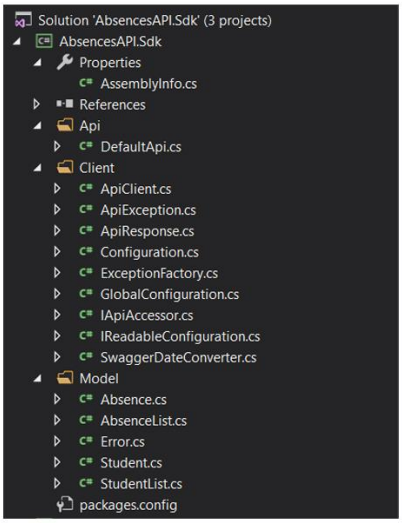

Рисунок 3.15 - Структура папок у створеному рішенні

Рішення має конфігурацію, яка знає базовий шлях API, визначений у специфікації OpenAPI, і існують різні опції для керування безпекою API, такі як надсилання токенів доступу або ключів API.

DefaultAPI.cs містить функції, які здійснюють виклики до кінцевих точок API. Ці функції мають назви, параметри та описи на основі визначення OpenAPI , як показано на рисунку 3.16.

```
/// <summary>
/// Add new absence for student Add a new absence.
/// </summary>
/// <exception cref="AbsencesAPI.Sdk.Client.ApiException">Thrown when fails to make API call</exception>
/// <param name="studentId">Student Identification number</param>
/// <param name="body">Absence</param>
/// <returns>Absence</returns>
public Absence AddAbsence (int? studentId, Absence body)
{
    ApiResponse<Absence> localVarResponse = AddAbsenceWithHttpInfo(studentId, body);
    return localVarResponse.Data;
}
```

Рисунок 3.16 - Приклад функції, згенерованої Swagger Codegen

Swagger Codegen також генерує класи, що базуються на моделях з визначення API. Вони мають усі ті ж атрибути, що показано на рисунку 3.17

```
/// <summary>
/// Initializes a new instance of the <see cref="Student" /> class.
/// </summary>
[JsonConstructorAttribute]
protected Student() { }
/// <summary>
/// Initializes a new instance of the <see cref="Student" /> class.
/// </summary>
/// <param name="StudentId">StudentId (required).</param>
/// <param name="FirstName">FirstName.</param>
/// <param name="LastName">LastName.</param>
public Student(long? StudentId = default(long?), string FirstName = default(string), string LastName = default(string))
{
    // to ensure "StudentId" is required (not null)
    if (StudentId == null)
    {
        throw new InvalidDataException("StudentId is a required property for Student and cannot be null");
    }
    else
    {
        this.StudentId = StudentId;
    }
    this.FirstName = FirstName;
    this.LastName = LastName;
}
```

Рисунок 3.17 - Клас відповідно до моделі «Student» зі специфікації OpenAPI

Крок 4: Створення консольного застосунку для здійснення викликів API
Додайте новий проект консольного застосунку до рішення AbsencesAPI.Sdk , як показано на рисунку 3.18.

1. Встановіть новий проект консольної програми як проект запуску рішення.

2. Додайте посилання на проєкт AbsencesAPI.Sdk .

Додайте рядки *за допомогою AbsencesAPI.Sdk.Api ;* та *за допомогою AbsencesAPl.Sdk.Client ;* на початок Program.cs , як показано на рисунку 3.19.

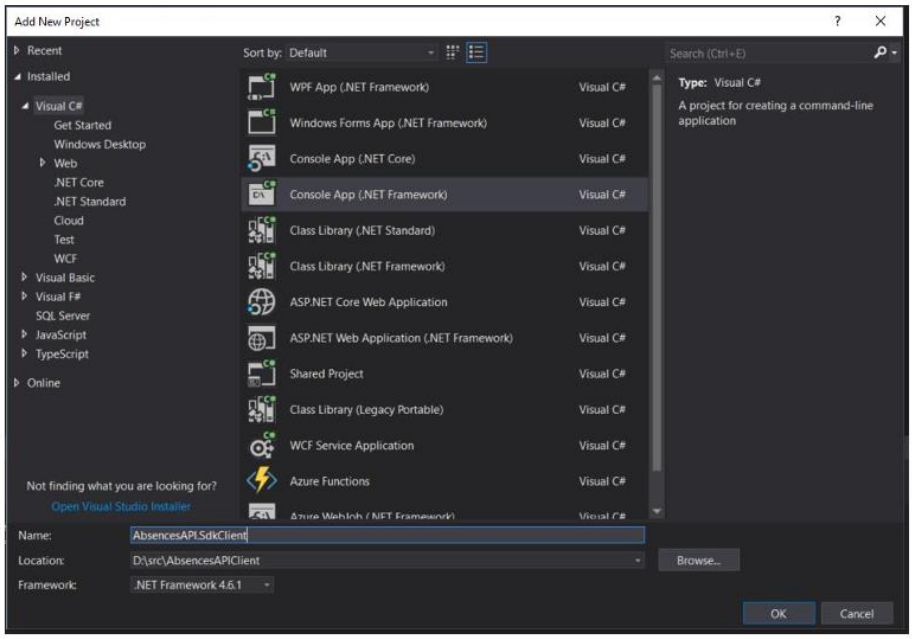

Рисунок 3.18 - Створення нової консольної програми

```
1  using System;
2  using AbsencesAPI.Sdk.Api;
3  using AbsencesAPI.Sdk.Client;
```

Рисунок 3.19 - Додавання директив using

Тепер, коли всі посилання на місці, виклики API можна додати до методу Main консольної програми. Повна програма показана на рисунку 3.20 з коментарями, що її пояснюють.

```
namespace AbsencesAPI.SdkClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Configuration sets the path automatically to predefined http://absences.azure-api.net/api/1.0
            // Subscription key is passed in header by function AddDefaultHeader
            var config = new Configuration();
            config.AddDefaultHeader("Ocp-Apim-Subscription-Key", "5c[redacted]");

            // Define new API with previous configurations
            var api = new DefaultApi(config);

            // List students with Http info to get all information
            var allStudents = api.ListStudentsWithHttpInfo();
            var httpResponse = allStudents.StatusCode; // returns 200
            var students = allStudents.Data; // returns a list of students from mock response

            Console.WriteLine("Response code: " + httpResponse + "\n");

            // Loops through a list of students that has one example student
            foreach (var student in students)
            {
                Console.WriteLine("Student info: \n" +
                    "Id: " + student.StudentId + "\n" +
                    "Name: " + student.FirstName + " " + student.LastName + "\n");
            }

        }
    }
}
```

Рисунок 3.20 - Консольний додаток, що виконує виклики API до Azure

В результаті виклик отримує фіктивну відповідь від Azure APIM, яка була визначена в специфікації OpenAPI , як показано на рисунку 3.21.

```
C:\WINDOWS\system32\cmd.exe
Response code: 200

Student info:
Id: 1234567890
Name: Bob Smith

Press any key to continue . . .
```

Рисунок 3.21 - Відповідь від імітованого API клієнтському додатку

3.3 Оцінювання ефективності тестування API та аналіз результатів

Результатом роботи є функціонуючий API-first процес у середовищі Azure з використанням двох інструментів, Swagger Editor та Codegen , а також стандарту OpenAPI у розробці. API-first проектування та інструменти були протестовані та оцінені в ітераціях шляхом розробки API та клієнтської програми на основі попередніх ітерацій. API-first процес, описаний у розділі 6, є кінцевим результатом , який відповідає вимогам відповідних інструментів та

ефективності.

Цей розділ має на меті відповісти на дослідницькі питання та розкрити кожне питання на основі дослідження.

Що таке API-орієнтований дизайн?

API-first дизайн – це метод розробки API, де розробка починається з етапу проектування. Проектування API включає визначення того, які кінцеві точки API потрібні для задоволення його вимог, та написання визначення API, яке описує API у форматі, зчитуваному людиною та машиною . Написання визначення API є важливою частиною API-first дизайну, оскільки його можна використовувати як документацію API, а API реалізується на основі цього визначення. Використання методу API-first змушує ретельно продумати структуру та призначення API, що призводить до вищої якості API.

Як інструменти та стандарти підтримують API-орієнтований дизайн?

Інструменти та стандарти, які мали підтримувати API-first-проектування, були оцінені та протестовані на їхню сумісність з методом та Azure. Інструменти, представлені в цій роботі, були достатньо цінними для процесу та підходили саме для API-first підходу.

Використання інструментів під час розробки API є ефективним та економним , а також робить весь процес відносно легким для першого разу. Оскільки інструменти пришвидшують весь процес, вони можуть компенсувати час, витрачений на проектування API, і тим самим знизити планку для початку застосування API-орієнтованого проектування. Навіть якщо ретельне проектування API має багато переваг і тому вважається виправданим витраченого часу, необхідність витрачати весь цей додатковий час на початку може здатися не ідеальним рішенням. Використання належних інструментів допомагає компенсувати цю проблему .

Створення специфікацій OpenAPI – це корисна навичка для всіх, хто працює з REST API, оскільки існує дуже багато роботи, яку можна пом'якшити за допомогою інструментів, що підтримують OpenAPI . Специфікація була основою всього процесу, оскільки вона допомагала тестувати дизайн API та

використовувалася для розгортання API в Azure та створення клієнтського SDK за допомогою Swagger Codegen. Навчитися створювати правильне визначення OpenAPI було найбільш трудомістким технічним питанням під час цього дослідження, а також важливим питанням для правильного виконання, оскільки від цього залежали інші кроки.

Редактор Swagger добре підтримував написання специфікацій OpenAPI . Найважливішою особливістю щодо навчання написанню визначень було отримання миттєвого зворотного зв'язку, який повідомляв про помилки та візуально показував визначення в міру його розвитку. Під час тестування редактора Swagger виникла одна проблема. Синтаксична помилка не була виявлена редактором Swagger і виникла під час створення нового API в Azure з визначення API. Azure APIM не приймав те саме визначення OpenAPI у форматі JSON, в якому редактор Swagger не виявив жодних помилок.

Використання Swagger Codegen для створення клієнтського SDK може заощадити багато часу, пропонуючи комплексний проект з усіма основами для здійснення викликів API. Згенерований код сам по собі також має деякі переваги; він буде узгодженим , тому з ним легко та передбачувано працювати. Поєднання згенерованого коду з імітованими викликами API є практичним рішенням. Можна одразу розпочати розробку клієнтського front-end застосунку на основі імітованого API.

у цьому дослідженні було протестовано чотири інструменти. Усі чотири інструменти, протестовані в ітераціях, допомогли краще зрозуміти використання методу API-first та з'ясувати, як специфікацію OpenAPI можна вигідно застосувати; проте два інструменти не підходили для ефективного процесу проектування API-first у поєднанні з управлінням Azure API. Ці інструменти використовувалися в першій ітерації та були виключені з остаточної документації процесу API-first. Їхні функції не додали достатньої цінності порівняно з часом, необхідним для їх налаштування та навчання їх використанню, оскільки вони дублювали функції управління Azure API. Крім того, їх не можна було б вільно використовувати в професійних цілях.

Як API-орієнтований дизайн може ефективно допомогти розробці API в середовищі Azure ?

Процес, орієнтований на API, представлений у цій роботі, демонструє ефективний спосіб застосування API-орієнтованого проектування з Azure. Він включає використання інструментів, що підтримують цей процес, та кроки, що використовують власні функції Azure.

API-орієнтований дизайн має багато переваг, які були підтверджені під час цього дослідження. Переваги використання API-орієнтованого дизайну в середовищі Azure:

- Гарна та зрозуміла документація API створюється автоматично відповідно до визначення API. Документацію можна переглянути на порталі Azure.
- Розгортання API в Azure легко здійснити, якщо є визначення API.
- API точно відповідає потребам бізнесу, оскільки його дизайну було приділено пильну увагу.
- Це зменшує ймовірність помилок, маючи інструменти, які можуть їх виявляти, та автоматизуючи частини процесу.
- Розробка та написання визначення API впливає на досвід розробника API, створюючи якісну документацію та узгоджений API.

Остаточний процес API-first з використаними інструментами та керуванням Azure API не повністю відповідав крокам API-first. Це стало результатом ітерацій , які дійшли висновку, що кроки необхідно змінити, щоб краще відповідати керуванню Azure API. Процес, який суворо дотримувався кроків API-first, можна було б виконати за допомогою іншого набору інструментів, хоча з Azure це було недоцільно . Кінцевий результат відрізняється від кроків API-first щодо тестування на ранній стадії, де рекомендувалося тестувати виклики API до інструменту перед впровадженням. Завдяки керуванню Azure API впровадження API на основі його специфікації OpenAPI відбувається швидко та легко, що робить можливим його подальше внесення змін. Раннє тестування дизайну API можна здійснити, порівнявши структуру

REST з історіями користувачів. Azure APIM також пропонує хороші функції для тестування в Azure Portal.

API-first-проектування як тема була відносно новою, і на той час інших досліджень на цю тему знайти не вдалося. Однак, матеріалу з різних джерел було достатньо для отримання достовірної інформації та формування гарного розуміння теми й основи для дослідження. Більшість цих джерел були онлайн-статтями та публікаціями в блогах представників компаній, що спеціалізуються на розробці API. Інформація про API-first-проектування з різних джерел була узгодженою, тому теоретичну основу цього дослідження можна вважати достовірною.

Метод дослідження, використаний у цій роботі, а саме дизайн-дослідження, добре відповідає меті цього дослідження. Дослідження мало на меті вирішити проблему за допомогою практичного рішення, і ітеративний характер дизайн-дослідження був необхідний для отримання корисного та вдосконаленого результату. Кожна ітерація давала більше розуміння самого процесу API-first, того, як отримати більше користі від набору інструментів та з'ясувати, чи є інструменти чи практики, які недостатньо цінні для цього процесу.

Це дослідження охоплювало один випадок у певному середовищі розробки з використанням Windows та Visual Studio через потреби доручителя роботі, що означає, що воно не є повністю застосовним до інших середовищ як таких; однак результати мають бути відтворюваними в будь-якій розробці API, виконаній у тому ж середовищі. Проходження кількох ітерацій допомогло гарантувати надійну роботу технічних частин дослідження.

Оскільки розглянутий процес API-first був реалізований як демонстрація, а не як реальний бізнес-кейс, йому бракує можливості тестування в реальному робочому середовищі та ознайомлення з думками розробників щодо його корисності. Хоча з самого початку метою було зберегти обсяг роботи достатньо невеликим, результати могли б бути різноманітнішими та більш надійними в такому середовищі. Крім того, в контексті цього дослідження можна було перевірити лише короткострокові переваги API-first проектування. Це також

очікувалося; однак, подальше дослідження довгострокових переваг було б бажаним доповненням.

Оскільки це дослідження було проведене з вступної точки зору API-first дизайну, з цієї роботи можна вивести багато тем для дослідження. Найважливішим аспектом, який не був розглянутий на практиці в цій роботі, є вплив API-first дизайну на командну роботу. З результатів цього дослідження можна зробити висновок , що можна одночасно працювати над фронтендом та бекендом після ввімкнення імітації викликів API, тому можна провести подальші дослідження його впливу. Іншим можливим предметом дослідження може бути впровадження API-first дизайну з продуктами, які вже існують. Чи варто це робити , чи взагалі можливо?

Оскільки економіка API є зростаючою тенденцією, будь-яке дослідження ефективних методів розробки API буде актуальним.

3.4 Висновки по розділу

У третьому розділі було розглянуто процес розробки, інтеграції та оцінювання ефективності API-first рішень. Проаналізовано сучасні інструменти та стандарти, що підтримують API-first підхід у розробці програмних систем. Виконано практичну реалізацію API та клієнтського застосунку на основі API-first методології, що дозволило забезпечити чітку взаємодію між компонентами системи. Також проведено оцінювання ефективності тестування API та аналіз отриманих результатів. Підсумкові дані підтверджують доцільність використання API-first підходу для підвищення якості, масштабованості та підтримуваності сучасних програмних систем.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено та реалізовано підхід до інженерії програмних систем із використанням API-first, що є сучасною методологією розробки, орієнтованою на проєктування інтерфейсів взаємодії між компонентами системи на ранніх етапах. Розроблене програмне рішення демонструє комплексний підхід до створення масштабованого, гнучкого та зручного у використанні програмного продукту. У роботі проаналізовано існуючі підходи до API-орієнтованої розробки, визначено їх переваги та недоліки, а також реалізовано функціональні можливості системи, що відповідають сучасним вимогам до якості програмного забезпечення. Процес розробки охоплював аналіз вимог, проєктування архітектури, реалізацію API та клієнтського застосунку, тестування та оцінку ефективності рішення.

На початковому етапі було проведено аналіз предметної області та сформульовано функціональні і нефункціональні вимоги до системи. Особливу увагу приділено вимогам до масштабованості, продуктивності, безпеки та зручності інтеграції, що є ключовими для сучасних розподілених систем. Моделювання бізнес-логіки дозволило структурувати процеси взаємодії між компонентами системи та визначити ключові точки інтеграції через API. Було розроблено відповідні діаграми (Use Case, Sequence, компонентні моделі), які стали основою для подальшої реалізації.

У процесі реалізації було створено серверну частину системи з використанням сучасних технологій, що забезпечують розробку RESTful API, а також клієнтський застосунок для взаємодії з користувачем. Центральним елементом архітектури стало API, яке визначає контракт між компонентами системи та забезпечує їх узгоджену взаємодію. Для опису та документування API використано відповідні стандарти, що дозволило автоматизувати процес розробки та тестування. Забезпечено ефективну взаємодію між фронтендом і бекендом, що значно зменшило ризики неузгодженості між компонентами системи.

Окрему увагу приділено питанням якості програмного забезпечення. Було проведено комплексне тестування, включаючи модульне, інтеграційне та автоматизоване тестування API. Це дозволило виявити потенційні проблеми на ранніх етапах та забезпечити стабільність роботи системи. Використання підходів CI/CD сприяло автоматизації процесів збірки, тестування та розгортання, що підвищило ефективність розробки та зменшило ймовірність помилок.

У результаті виконання роботи досягнуто поставленої мети — розроблено програмну систему із застосуванням API-first підходу, яка забезпечує ефективну інтеграцію компонентів, високу масштабованість та зручність підтримки. Основними перевагами обраного підходу є чітке визначення контрактів взаємодії, можливість паралельної розробки різних частин системи, покращення якості документації та спрощення тестування. Водночас варто зазначити, що впровадження API-first підходу потребує додаткових зусиль на етапі проєктування та високого рівня дисципліни в команді розробників.

Перспективи подальшого розвитку полягають у розширенні функціональних можливостей системи, впровадженні більш складних механізмів безпеки, використанні мікросервісної архітектури, а також інтеграції з зовнішніми сервісами та платформами. Отримані результати можуть бути використані при розробці сучасних програмних систем, що потребують високого рівня інтеграції, масштабованості та надійності.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. OpenAPI Initiative. (2025). OpenAPI Specification. https://www.openapis.org
2. Swagger Documentation. (2025). swagger.io. https://swagger.io/docs/
3. Microsoft. (2025). ASP.NET Core Web API Documentation. https://learn.microsoft.com/en-us/aspnet/core/web-api/
4. Microsoft Azure API Management. (2025). https://learn.microsoft.com/en-us/azure/api-management/
5. Google Cloud APIs Design Guide. (2024). https://cloud.google.com/apis/design
6. Richardson, L., & Amundsen, M. (2013). RESTful Web APIs. O'Reilly Media.
7. Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures.
8. Newman, S. (2021). Building Microservices (2nd ed.). O'Reilly Media.
9. Fowler, M. (2002). Patterns of Enterprise Application Architecture. https://martinfowler.com/eaaCatalog/
10. Martin, R. C. (2017). Clean Architecture. Pearson.
11. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns. Pearson.
12. Freeman, E., & Robson, E. (2021). Head First Design Patterns. O'Reilly.
13. OWASP Foundation. (2024). OWASP API Security Top 10. https://owasp.org/www-project-api-security/
14. Postman. (2025). API Development & Testing Guide. https://learning.postman.com/docs/
15. Stoplight. (2024). API Design with API-First Approach. https://stoplight.io/api-design

16. Red Hat. (2024). API-first development strategy. https://www.redhat.com/en/topics/api/what-is-api-first

17. IBM Cloud. (2025). API Connect Documentation. https://cloud.ibm.com/docs/api-connect

18. MuleSoft. (2024). API-led Connectivity Guide. https://www.mulesoft.com/resources/api/what-is-api-led-connectivity

19. Kong Inc. (2025). API Gateway Documentation. https://docs.konghq.com/

20. GraphQL Foundation. (2025). GraphQL Documentation. https://graphql.org/learn/

21. REST API Tutorial. (2024). https://restfulapi.net/

22. Docker Documentation. (2025). https://docs.docker.com/

23. Kubernetes Documentation. (2025). https://kubernetes.io/docs/

24. Microsoft. (2025). .NET Microservices Architecture Guide. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

25. Azure DevOps. (2025). CI/CD Pipelines. https://learn.microsoft.com/en-us/azure/devops/pipelines/

26. GitHub Docs. (2025). REST API & GraphQL API. https://docs.github.com/en/rest

27. GitLab. (2025). API Documentation. https://docs.gitlab.com/ee/api/

28. Selenium. (2025). Documentation. https://www.selenium.dev/documentation/

29. Cypress. (2025). Testing Framework Guide. https://docs.cypress.io/

30. OpenTelemetry. (2024). Observability Framework. https://opentelemetry.io/docs/

31. Nginx. (2025). API Gateway & Reverse Proxy. https://nginx.org/en/docs/

32. Amazon Web Services. (2025). API Gateway Documentation. https://docs.aws.amazon.com/apigateway/

33. Google Apigee. (2025). API Management Platform. https://cloud.google.com/apigee

34. Netflix Tech Blog. (2024). Microservices and API Design. https://netflixtechblog.com/

35. ThoughtWorks. (2024). API-first Architecture Insights. https://www.thoughtworks.com/insights

36. Gartner. (2024). API Strategy and Management Trends. https://www.gartner.com/en/information-technology

37. Richardson Maturity Model. (2023). REST API Maturity Model. https://martinfowler.com/articles/richardsonMaturityModel.html

38. AsyncAPI Initiative. (2025). AsyncAPI Specification. https://www.asyncapi.com/

39. JSON Schema. (2024). Documentation. https://json-schema.org/

40. W3C. (2024). Web Architecture and API Standards. https://www.w3.org/standards/webdesign/apis

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Інженерія програмних систем із використанням API-first підходу "

Обсяг пояснювальної записки: 79 аркушів

Дата закінчення дипломної роботи 10 червня 2026р.

Підпис студента _____