Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ МЕТОДІВ РОЗРОБКИ КОМП'ЮТЕРНИХ ЗАСТОСУНКІВ З ІНТЕГРОВАНОЮ API
(тема)

Виконав:
студент 2 курсу, групи ІНФм-23-1

Маренич В.В.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Руденко Д.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Мареничу Владиславу Валерійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів розробки комп'ютерних застосунків з інтегрованою API

затверджена наказом по університету від 25 листопада 2024 року № 1246Ст

2. Термін подання студентом роботи до екзаменаційної комісії 4 січня 2025 р.

3. Вихідні дані до роботи перелік методів інтеграції API, теоретичні відомості про методи обробки запитів і передачі даних у програмних додатках.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд сучасних методів інтеграції API у програмні додатки.
2. Дослідження принципів роботи REST, SOAP, GraphQL, і WebSockets.
3. Аналіз архітектурних патернів MVC і MVVM для інтеграції API.
4. Розробка програмного прототипу додатку з використанням WPF і API.
5. Аналіз продуктивності застосунків залежно від обраного формату даних (JSON, XML).

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) сучасні методи інтеграції API у програмні додатки, постановка задачі, принципи роботи REST, SOAP, GraphQL, і WebSockets, аналіз продуктивності застосунків залежно від обраного формату даних

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	25.11.2024	
2	Аналіз завдання, підбір літератури	26.11.24-28.11.24	
3	Аналіз літератури з досліджуваної проблеми	29.11.24-30.11.24	
4	Аналіз технічних засобів	01.12.24-03.12.24	
5	Розробка методу	04.12.24-05.12.24	
6	Програмна реалізація	06.12.24-11.12.24	
7	Оформлення пояснювальної записки	11.12.24-12.12.24	
8	Перевірка на плагіат	13.12.2024	
9	Рецензування	14.12.2024	
10	Підготовка презентації та доповіді	20.12.2024	
11	Занесення роботи в електронний архів	11.01.2025	
12	Попередній захист кваліфікаційної роботи	13.01.2025	

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Руденко Д.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 64 с., 2 табл., 28 рис., 28 джерел.

API ІНТЕГРАЦІЯ, REST, SOAP, GRAPHQL, WEBSOCKETS, АРХІТЕКТУРА ЗАСТОСУНКІВ, МОДЕЛЬ ЗАПИТ-ВІДПОВІДЬ, КОМУНІКАЦІЯ В РЕАЛЬНОМУ ЧАСІ, API ДОКУМЕНТАЦІЯ.

Об’єктом дослідження є процес розробки комп’ютерних застосунків з використанням інтегрованих API.

Метою дослідження є виявлення методів та засобів, що забезпечують ефективну інтеграцію API у програмні застосунки, зокрема, шляхом розробки оптимальних підходів до використання API на основі сучасних архітектурних патернів та протоколів.

В роботі використано методи програмної інженерії, аналітичного моделювання та експериментальної реалізації. Проведено аналіз популярних методів інтеграції API, зокрема REST, SOAP, GraphQL та WebSockets, досліджено їх переваги, недоліки та сфери застосування.

У результаті дослідження було розроблено програмний прототип, який демонструє ефективну інтеграцію API з використанням WPF та патерну MVVM.

API INTEGRATION, REST, SOAP, GRAPHQL, WEBSOCKETS, APPLICATION ARCHITECTURE, REQUEST-RESPONSE MODEL, REAL-TIME COMMUNICATION, API DOCUMENTATION.

The object of research is the process of developing computer applications using integrated APIs.

The aim of the research is to identify methods and tools that ensure effective integration of APIs into software applications, in particular, by developing optimal approaches to the use of APIs based on modern architectural patterns and protocols.

The methods of software engineering, analytical modeling, and experimental implementation are used in this work. An analysis of popular API integration methods, such as REST, SOAP, GraphQL, and WebSockets, was carried out, and their advantages, disadvantages, and areas of application were investigated.

As a result of the research, a software prototype was developed that demonstrates effective API integration using WPF and the MVVM pattern.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface (Інтерфейс програмного застосунку)

JSON – JavaScript Object Notation (Формат обміну даними JavaScript)

XML – eXtensible Markup Language (Розширювана мова розмітки)

HTTP – HyperText Transfer Protocol (Протокол передачі гіпертексту)

SOAP – Simple Object Access Protocol (Простий протокол доступу до об’єктів)

RSS – Really Simple Syndication (Дуже простий синдикаційний формат)

URL – Uniform Resource Locator (Уніфікований вказівник ресурсу)

XAML – eXtensible Application Markup Language (Розширювана мова розмітки застосунків)

WPF – Windows Presentation Foundation (Фреймворк для створення графічного інтерфейсу Windows)

MVVM – Model-View-ViewModel (Модель-Подання-Модель подання)

WSDL – Web Services Description Language (Мова опису вебслужб)

ВСТУП

Розробка програмного забезпечення з інтегрованими API поділяється на кілька ключових напрямків, включаючи проєктування, реалізацію та інтеграцію API.

Розробка API передбачає створення структурованого інтерфейсу, який дозволяє різним програмним компонентам ефективно взаємодіяти. Сюди входить визначення кінцевих точок, протоколів і форматів даних. API використовуються в різних сферах, забезпечуючи безперешкодну взаємодію між застосунками, наприклад, отримання даних із зовнішніх джерел, підключення до сервісів або керування пристроями.

Інтеграція API фокусується на включенні сторонніх сервісів або функцій у застосунок. Цей процес вимагає від розробників розуміння структури зовнішнього API, методів автентифікації та правильної обробки запитів і відповідей на них. Успішна інтеграція розширює можливості програмного забезпечення, надаючи користувачам розширені функції та доступ до зовнішніх даних або сервісів.

Важливість дослідження методів розробки програмного забезпечення з інтегрованими API полягає в оптимізації продуктивності, покращенні користувацького досвіду та створенні більш динамічних, підключених застосунків.

1 ОГЛЯД ДОСТУПНИХ ЗАСОБІВ

1.1 Application Programming Interface

API (Application Programming Interface) – це набір правил і протоколів, які дозволяють різним програмним застосункам спілкуватися та взаємодіяти один з одним. Він визначає методи, формати даних і протоколи, які розробники можуть використовувати для доступу та використання функціональних можливостей і даних програмної системи, сервісу або платформи.

API діє як посередник (рис. 1.1) [1–6], дозволяючи програмним компонентам безперешкодно спілкуватися та обмінюватися інформацією. Він надає чітко визначений інтерфейс, який визначає, як повинні взаємодіяти різні програмні компоненти, включаючи структуру запитів і відповідей, підтримувані формати даних (наприклад, JSON, XML), механізми автентифікації та обробку помилок.

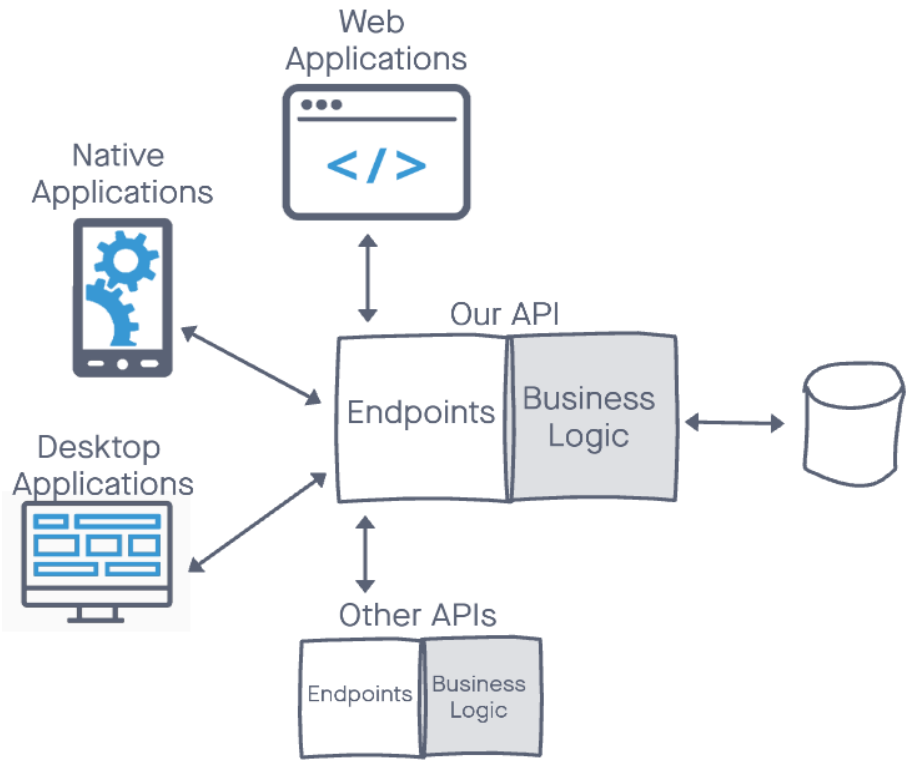


Рисунок 1.1 – Приклад взаємодії API з програмами

API можна класифікувати на різні типи, такі як веб-API, доступ до яких зазвичай здійснюється через Інтернет за допомогою стандартних протоколів, таких як HTTP, та бібліотечні API, які є наборами функцій і процедур, що їх розробники можуть використовувати у своєму власному коді [7, 8].

API відіграють важливу роль у розробці програмного забезпечення, дозволяючи розробникам використовувати існуючі функціональні можливості та інтегрувати різні програмні системи, сервіси або платформи у власні застосунки. Вони дозволяють розробникам спиратися на існуючу програмну інфраструктуру, прискорювати розробку та створювати більш надійні та багатофункціональні застосунки, використовуючи можливості інших програмних компонентів.

Найрозповсюдженішим форматом передачі даних за допомогою API є JSON.

JSON (JavaScript Object Notation) був представлений Дугласом Крокфордом на початку 2000-х років. Крокфорд, інженер-програміст і автор, розробив JSON як полегшену альтернативу XML (eXtensible Markup Language) для обміну даними.

Мотивація створення JSON полягала в тому, щоб забезпечити простіший, зручніший для читання та ефективніший формат для передачі та зберігання структурованих даних. XML, який на той час широко використовувався для обміну даними, мав складніший синтаксис і вимагав більшої пропускної здатності для передачі даних.

JSON черпав натхнення з об’єктно-літерального синтаксису JavaScript, який представляв об’єкти як пари ключ-значення. Крокфорд хотів створити незалежний від мови формат, який можна було б легко аналізувати і генерувати різними мовами програмування.

Об’єкт в JSON береться у фігурні дужки (рис. 1.2) і складається з набору пар ключ-значення. Кожен ключ – це рядок, після якого ставиться двокрапка, а відповідне значення може бути різних типів, наприклад, рядок,

число, логічне значення, null, інший об’єкт або масив. Масив в JSON – це впорядкований набір значень, укладений у квадратні дужки.

```
{
    "count": 7,
    "items": ["socks", "pants", "shirts", "hats"],
    "manufacturer": {
        "name": "Molly's Seamstress Shop",
        "id": 39233,
        "location": {
            "address": "123 Pickleton Dr.",
            "city": "Tucson",
            "state": "AZ",
            "zip": 85705
        }
    },
    "total_price": "$393.23",
    "purchase_date": "2022-05-30",
    "country": "USA"
}
```

Рисунок 1.2 – Приклад файлу JSON

У 2001 році Крокфорд опублікував початкову специфікацію JSON, яка описувала синтаксис і основні правила представлення даних у форматі JSON. JSON швидко набув популярності завдяки своїй простоті, і його використання вийшло за межі застосунків, пов’язаних з JavaScript.

Простота JSON і сумісність з широким спектром мов програмування зробили його ідеальним для веброзробки та комунікації між вебсервісами. Він став стандартом де-факто для обміну даними у веб-API, дозволяючи передавати і використовувати дані різними системами.

З моменту свого створення JSON зазнав мінімальних змін і став загальноприйнятим форматом для передачі та зберігання структурованих даних. Він підтримується практично всіма мовами програмування і відіграв вирішальну роль у зростанні вебсервісів і розробці сучасних вебзастосунків.

Другим за популярністю форматом передачі даних за допомогою API є XML.

XML (eXtensible Markup Language) – це мова розмітки, яка визначає набір правил для кодування документів у форматі, придатному як для

читання людиною, так і для читання машиною. Вона була розроблена наприкінці 1990-х років Консорціумом World Wide Web (W3C).

XML створений для моделювання, зберігання та передачі даних, що робить його придатним для широкого спектру задач. Він надає можливість визначати та записувати вміст і структуру даних за допомогою спеціальних тегів, відомих як елементи. Ці елементи беруться в кутові дужки ("<>" і "</>") і можуть бути вкладені один в одного для створення ієрархічної структури (рис. 1.3).

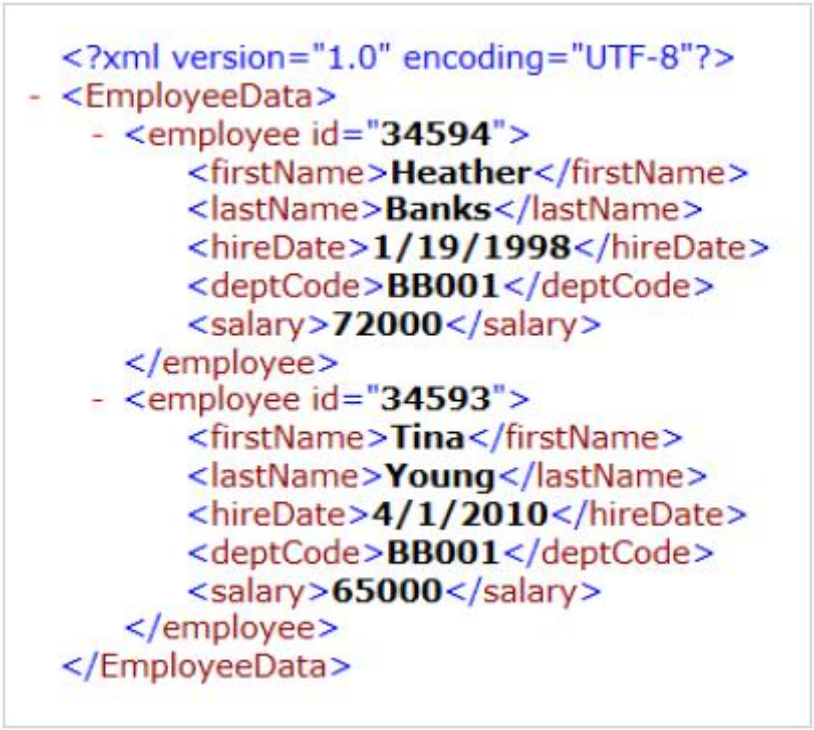


Рисунок 1.3 – Приклад файлу XML

XML-документи складаються з заголовка, який визначає версію XML, що використовується, кодування символів і будь-які інші необхідні декларації. Потім документ містить кореневий елемент, який інкапсулює інші елементи, утворюючи деревоподібну структуру.

XML забезпечує зручний і доступний спосіб представлення структурованих даних, що робить його придатним для різних цілей, включаючи зберігання даних, конфігураційні файли, обмін даними між

системами і багато іншого. Він дозволяє розробникам визначати власні теги та структурувати дані відповідно до їхніх конкретних вимог.

XML широко підтримується мовами програмування і є основою багатьох вебсервісів, таких як SOAP (Simple Object Access Protocol) і RSS (Really Simple Syndication). Однак останніми роками його використання зменшилося, оскільки JSON стає все більш популярним для обміну даними завдяки своїй простоті та ефективності.

1.2 Архітектура застосунку

Застосунок, який використовує API, зазвичай працює за архітектурою клієнт-сервер. Яка в свою чергу ділиться на Клієнта, Сервер та безпосередньо API.

Клієнтська частина програми відповідає за користувацький інтерфейс і взаємодію. Вона може бути реалізована за допомогою різних технологій, таких як веббраузер, фреймворк для десктопних застосунків (наприклад, WPF, Electron) або фреймворк для мобільних застосунків (наприклад, React Native, Flutter). Клієнт надсилає запити до сервера на отримання певних даних з API та відображає отриману інформацію користувачеві.

Серверна частина програми виступає посередником між клієнтом і API. Він отримує запити від клієнта, виконує необхідну обробку, взаємодіє з API і повертає запитувані дані клієнту. Сервер може бути реалізований за допомогою вебфреймворку (наприклад, Node.js з Express, ASP.NET) або будь-якої іншої технології, яка може обробляти HTTP-запити (рис. 1.4).

API слугує джерелом даних для застосунку. Він надає набір кінцевих точок, які дозволяють отримати доступ до різних даних, пов’язаних з грою. Сервер взаємодіє з API, надсилаючи HTTP-запити до відповідних кінцевих точок, включаючи автентифікацію, якщо це необхідно. Отримані дані потім обробляються та пересилаються клієнту.

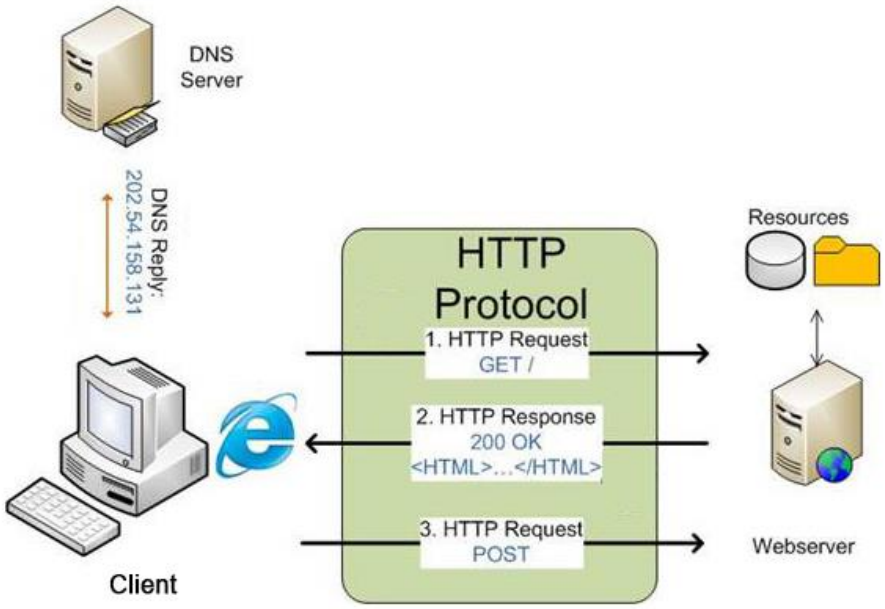


Рисунок 1.4 – Приклад роботи протоколу HTTP

Загалом, архітектура передбачає, що клієнт надсилає запити на сервер, який взаємодіє з API для отримання запитуваних даних. Сервер обробляє та готує дані для клієнта, який потім відображає їх користувачеві через інтерфейс користувача. Така архітектура забезпечує розподіл завдань, масштабованість і гнучкість при розробці застосунку.

1.3 Доступні методи інтеграції API

Інтеграція API – це процес з’єднання різних програмних систем або застосунків за допомогою API (інтерфейсів прикладного програмування) для обміну даними або функціоналом. Існує кілька методів інтеграції API залежно від архітектури системи, типу використовуваного API та вимог програми. У цьому розділі розглядаються найбільш поширені методи інтеграції API.

1.3.1 REST (Representational State Transfer)

REST – це один з найпоширеніших підходів для інтеграції API, особливо для вебзастосунків. Він базується на наборі архітектурних принципів, які використовують стандартні методи HTTP для полегшення комунікації між клієнтом і сервером.

Можна виділити декілька основних властивостей REST:

– бездержавність: кожен запит від клієнта повинен містити всю інформацію, необхідну для обробки сервером. Це означає, що сервер не зберігає ніяких сесійних даних про клієнта між запитами, що спрощує дизайн сервера і підвищує його масштабованість;

– заснований на ресурсах: у REST дані розглядаються як ресурси, кожен з яких ідентифікується унікальною URL-адресою. Клієнти виконують операції з цими ресурсами за допомогою стандартних методів HTTP, таких як GET (отримання даних), POST (створення нових ресурсів), PUT (оновлення існуючих ресурсів) і DELETE (видалення ресурсів);

– архітектура клієнт-сервер: REST розділяє функції клієнта та сервера, дозволяючи їм розвиватися незалежно. Таке розділення забезпечує більш прості оновлення та зміни на одній стороні, не впливаючи на іншу;

– кешування або процес тимчасового зберігання даних у спеціально виділеній області пам’яті (кеші), щоб забезпечити швидкий доступ до них у майбутньому: відповіді від сервера можуть кешуватися клієнтами для підвищення продуктивності, зменшуючи потребу в повторних запитах одних і тих же даних. Це особливо корисно для застосунків з високим трафіком, оскільки зменшує навантаження на сервер і прискорює час відгуку для користувачів.

RESTful API зазвичай використовують формати JSON або XML для передачі даних, причому JSON є більш популярним вибором через свою легкість. Вони ідеально підходять для створення масштабованих і

ефективних застосунків, де ресурси потребують легкого доступу та маніпулювання ними.

1.3.2 SOAP (Simple Object Access Protocol)

SOAP – це протокол, який визначає більш жорсткий та формальний метод обміну даними між системами. Він використовує XML для форматування повідомлень і часто покладається на такі протоколи, як HTTP або SMTP для передачі повідомлень.

Безпека використання SOAP полягає в забезпеченні надійної системи за допомогою WS-Security, що дозволяє наскрізне шифрування та безпечні транзакції. Це робить SOAP придатним для застосунків, де цілісність і конфіденційність даних мають першорядне значення, наприклад, у фінансовому та медичному секторах.

Крім того, завдяки вбудованим механізмам обробки помилок і повторних спроб, SOAP забезпечує надійну доставку повідомлень. Ця функція має вирішальне значення для інтеграції на рівні підприємства, де важливо гарантувати, що повідомлення будуть доставлені без втрат, навіть у разі збоїв у мережі.

SOAP має чітко визначений стандарт, який включає специфікації для кодування повідомлень, віддаленого виклику процедур і розширюваності. Ця стандартизація є корисною для складних операцій, таких як розподілені транзакції, забезпечуючи сумісність між різними системами. SOAP зазвичай використовується в корпоративних застосунках, де потрібен вищий рівень безпеки, надійності та стандартизації. Здатність SOAP виконувати операції над декількома системами робить його ідеальним для застарілих систем, які потребують інтеграції з сучасними застосунками.

1.3.3 GraphQL

GraphQL – це мова запитів, розроблена компанією Facebook, яка забезпечує більш гнучкий підхід до інтеграції API у порівнянні з традиційними REST API. Вона дозволяє клієнтам запитувати лише ті дані, які їм потрібні, а не отримувати фіксовану структуру відповіді.

Серед переваг використання можна виділити наступні:

– гнучкість, яка полягає в тому, що клієнти можуть визначити точну форму та обсяг даних, які їм потрібні, в одному запиті. Це мінімізує надмірну вибірку (отримання занадто великої кількості даних) та недостатню вибірку (отримання занадто малої кількості даних), що є особливо корисним для застосунків зі складними моделями даних, де вимоги клієнта можуть суттєво відрізнятися;

– сильна типізація: GraphQL використовує схему строгої типізації, яка описує доступні дані та операції. Ця схема слугує контрактом між клієнтом і сервером, полегшуючи розуміння того, які дані доступні, та зменшуючи ймовірність помилок у запитах;

– ефективні запити забезпечують клієнтів можливістю виконувати кілька операцій в одному запиті, об’єднуючи пошук даних у меншу кількість мережевих звернень. Це зменшує кількість запитів і підвищує продуктивність, особливо в застосунках, які потребують даних з декількох джерел. GraphQL особливо корисна в сучасних веб і мобільних застосунках, де вимоги до даних можуть бути складними та динамічними.

1.3.4 WebSockets

WebSockets забезпечують повнодуплексний канал зв’язку через одне постійне з’єднання, що дозволяє здійснювати двонаправлений обмін даними в режимі реального часу між клієнтами та серверами (рис.1.5).

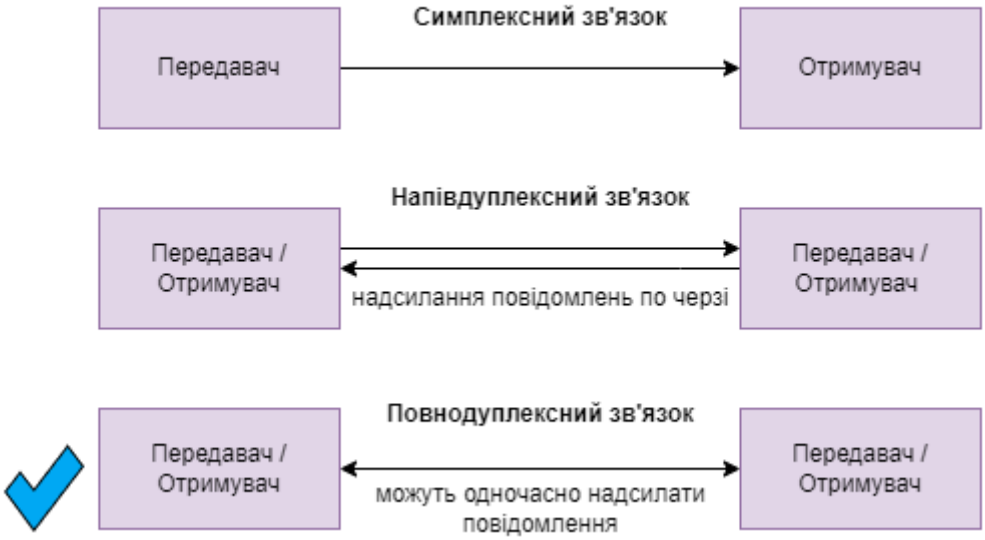


Рисунок 1.5 – Різні способи організації каналів зв’язку

На відміну від HTTP-протоколу, що діє за схемою «запит клієнта – відповідь сервера», у WebSockets обидві сторони, як сервер, так і клієнт, можуть взаємно надсилати повідомлення (табл. 1.1). Кожен учасник комунікації може одночасно приймати та передавати дані. У WebSockets обмін повідомленнями здійснюється через один спільний канал зв’язку, який залишається відкритим протягом усього часу комунікації. Будь-яка зі сторін, за потреби, може закрити цей канал.

WebSockets особливо підходять для застосунків, які вимагають миттєвого оновлення даних, таких як онлайн-ігри, чат або біржові платформи. Постійне з’єднання забезпечує миттєву передачу даних, що є критично важливим для застосунків, які потребують миттєвого зворотного зв’язку.

Підтримуючи постійне з’єднання, WebSockets значно зменшують накладні витрати, пов’язані зі створенням нових HTTP-запитів і відповідей. Така низька затримка зв’язку дуже важлива для застосунків, де час має вирішальне значення.

На відміну від традиційних API, які вимагають від клієнта опитування сервера на предмет оновлень, WebSockets дозволяють серверу передавати дані клієнту, як тільки вони стають доступними.

Таблиця 1.1. Порівняльна характеристика HTTP і WebSocket

HTTP	**WebSocket**
Однонаправлений протокол: тільки клієнт може ініціювати комунікацію — відправляти запити, сервер може тільки відповідати	Двонаправлений протокол: після встановлення з’єднання будь-яка сторона може відправляти повідомлення
З’єднання нетривале, за замовчуванням закривається після відповіді сервера	З’єднання підтримується до завершення комунікації
Під час комунікації завжди витрачається час на запит клієнта, навіть якщо важлива тільки відповідь сервера	Сервер може відправити повідомлення, не чекаючи на запит клієнта
Обов’язкове використання хедерів у повідомленнях	Після встановлення з’єднання у хедерах немає потреби, що значно зменшує розмір кожного повідомлення
Підтримує кешування автоматично	За замовчуванням не підтримує кешування

Ця архітектура, керована подіями, призводить до більш ефективного використання ресурсів і кращої продуктивності в цілому. API на основі WebSocket зазвичай використовуються в сценаріях, де взаємодія в режимі реального часу та передача даних з низькою затримкою є критично

важливими. Здатність обробляти декілька повідомлень одночасно робить їх ідеальними для сучасних інтерактивних застосунків.

1.4 Огляд мов програмування для написання застосунка, який використовує API, та вибір елементів проєкту

Існує кілька мов програмування, які зазвичай використовуються для розробки застосунків з API. Вибір мови залежить від знань розробника, платформи, вимог до продуктивності та необхідних функцій. Найпопулярніші мови для цього C++, C# та Python.

C++ є популярною мовою програмування для розробки застосунків з використанням API, оскільки вона забезпечує високу продуктивність і контроль над ресурсами. Завдяки своїм можливостям управління пам’яттю та підтримці об’єктно-орієнтованого програмування, C++ ідеально підходить для створення високоефективних і масштабованих рішень.

Існує багато бібліотек і фреймворків для роботи з API в C++, наприклад, Boost.Beast для HTTP/HTTPS і WebSocket-комунікацій, або cURL для роботи з мережевими запитами. Вони допомагають спрощувати інтеграцію з API та обробку запитів. C++ дозволяє налаштовувати і керувати з’єднаннями на низькому рівні, що надає гнучкість у взаємодії з API. Можна використовувати сокети або спеціалізовані бібліотеки для створення HTTP(S) запитів, роботи з WebSocket або RESTful API.

У сучасному C++ (починаючи з C++11) з’явилася підтримка асинхронного програмування через потоки (std::thread), обіцянки (std::promise), та асинхронні завдання (std::async), що дозволяє ефективно працювати з API без блокування основного потоку програми.

У C++ є можливість безпосередньо працювати з JSON або XML, які часто використовуються для обміну даними з API. Наприклад, бібліотеки

nlohmann/json дозволяють легко серіалізувати та десеріалізувати дані у форматі JSON.

C# – це сучасна об’єктно-орієнтована мова від Microsoft, що використовується для розробки Windows-застосунків за допомогою фреймворку .NET [9]. Вона підтримує такі функції, як узагальнення, LINQ (Language Integrated Query), async/await для асинхронного програмування та потужну обробку винятків. Ці можливості підвищують продуктивність розробника, полегшуючи написання чистого, стислого та зручного для підтримки коду при взаємодії з API. Також, C# є складовою частиною екосистеми .NET, яка надає широкий набір бібліотек, інструментів та фреймворків для створення застосунків, включаючи мережеву роботу, серіалізацію, маніпулювання даними та можливості паралельної роботи, що важливо для ефективної роботи з API.

Крім того, існують додаткові бібліотеки та фреймворки:

- RestSharp: стороння бібліотека для спрощеної роботи з API, яка надає інтуїтивно зрозумілий синтаксис для виконання запитів і обробки відповідей, що робить її популярною серед розробників C#.
- ASP.NET Core: фреймворк для створення веб-API, який надає потужні інструменти для розробки серверних застосунків з підтримкою RESTful API.

Python – популярна мова, зручна для початківців, яка має багато фреймворків для розробки кросплатформних застосунків з графічним інтерфейсом, таких як PyQt або Tkinter. Є однією з найпопулярніших мов програмування для розробки застосунків з використанням API завдяки своїй простоті, читабельності коду та великій кількості бібліотек для роботи з мережевими запитами та обробкою даних. Python широко використовується для роботи з RESTful API, інтеграції з вебсервісами, а також для створення власних API на основі фреймворків, таких як Flask або FastAPI.

Порівняння C++, C# та Python для розробки застосунків з використанням API допомагає зрозуміти, які особливості кожної з цих мов

можуть бути корисними в різних ситуаціях. У кожної з них є свої переваги та недоліки, які варто враховувати при виборі інструменту для розробки. Нижче наведено детальну порівняльну характеристику цих мов за кількома ключовими критеріями (табл. 1.2).

Таблиця 1.2. Порівняльна характеристика мов програмування для розробки застосунків з API

Критерій	**C++**	**C#**	**Python**
1	**2**	**3**	**4**
Продуктивність	Висока продуктивність завдяки компіляції в машинний код і оптимізації на низькому рівні.	Середня. Виконується на віртуальній машині (.NET), що забезпечує хорошу продуктивність, але не на рівні C++.	Низька в порівнянні з C++ і C#, оскільки Python інтерпретується, але достатня для більшості API-застосунків.
Простота розробки	Складний синтаксис та управління пам’яттю. Вимагає більше часу на розробку та дебагінг.	Простий і зрозумілий синтаксис, особливо для розробників, знайомих з мовами на основі C. Автоматичне управління пам’яттю (GC) спрощує розробку.	Дуже простий синтаксис, ідеально підходить для швидкої розробки та прототипування. Високий рівень абстракції та автоматизації.
Підтримка асинхронності	Підтримує асинхронність, але потребує ручної реалізації потоків або використання спеціалізованих бібліотек (наприклад, Boost).	Потужна підтримка асинхронності за допомогою async та await, що робить роботу з API швидкою та ефективною.	Асинхронність реалізується через asyncio та httpx, що забезпечує ефективну роботу з мережевими запитами. Проста реалізація асинхронних операцій.
Бібліотеки для роботи з API	Обмежений вибір бібліотек, таких як cURL або Boost.Beast. Потребують налаштування і можуть бути складними у використанні.	Багата екосистема бібліотек (HttpClient, RestSharp), які інтегруються з .NET. Простий та інтуїтивний інтерфейс для роботи з HTTP-запитами.	Широкий вибір бібліотек (requests, httpx, aiohttp), які забезпечують простоту та зручність роботи з API. Велика кількість документації та прикладів.
Масштабованість	Дуже висока. Підходить для створення високопродуктивних та оптимізованих систем, які мають обробляти великий обсяг даних або трафіку.	Добра масштабованість завдяки фреймворку ASP.NET, але потребує налаштувань та оптимізації на рівні сервера.	Менш масштабована в порівнянні з C++, але Python добре підходить для середніх та малих проєктів з API, особливо з використанням Flask або FastAPI.

1	2	3	4
Підтримка JSON/XML	Вимагає використання додаткових бібліотек для обробки JSON/XML (наприклад, RapidJSON). Може бути складною для новачків.	Вбудовані засоби та бібліотеки (System.Text.Json, XmlDocument), які легко інтегруються з C# застосунками.	Вбудована підтримка JSON (json), зручні бібліотеки (xml.etree.ElementTree, lxml). Простота обробки JSON-даних.
Фреймворки для створення API	Відсутні фреймворки для створення API на рівні, зручному для швидкої розробки. Часто потребує використання сторонніх інструментів або вручну писати логіку.	ASP.NET Core – потужний фреймворк для розробки API з підтримкою RESTful сервісів, маршрутизації та валідації.	Flask, FastAPI, Django REST Framework — легкі та гнучкі фреймворки для швидкої розробки API, з автоматичною генерацією документації та підтримкою OpenAPI.
Крос-платформність	C++ програми можна запускати на різних платформах, але код може потребувати модифікації для кожної з них.	.NET Core та сучасні версії C# дозволяють розробляти кросплатформені застосунки, що працюють на Windows, Linux та macOS.	Python є повністю кросплатформеним і працює на будь-якій системі, що підтримує Python-інтерпретатор (Windows, Linux, macOS).

Таким чином, можна зробити висновки, що C++ підходить для створення високопродуктивних та низькорівневих систем, де важлива оптимізація ресурсів та швидкість виконання. Однак, складність розробки та обмежена підтримка асинхронності робить його менш зручним для швидкої розробки API-застосунків. C# пропонує баланс між продуктивністю та простотою, особливо для тих, хто вже знайомий з екосистемою .NET. Завдяки потужному фреймворку ASP.NET Core, C# ідеально підходить для створення високопродуктивних API, які можуть працювати кросплатформно. Python найбільш підходить для швидкої розробки API та прототипування завдяки своїй простоті, багатому вибору бібліотек та фреймворків. Однак, для великих і продуктивних систем може вимагати додаткових оптимізацій та не забезпечує такої продуктивності, як C++ чи C#.

Переглянувши доступні технології для реалізації, було вирішено обрати C# як основну мову програмування для проєкту. Вона має потужні мовні можливості та інтегрується з .NET-екосистемою, яка пропонує широкий набір бібліотек, інструментів та фреймворків для створення ефективних та підтримуваних застосунків.

Для створення графічного інтерфейсу користувача було обрано фреймворк WPF (Windows Presentation Foundation) [10]. WPF надає потужний та гнучкий фреймворк для створення візуально привабливих користувацьких інтерфейсів. Він пропонує широкий спектр вбудованих елементів керування, стилів і шаблонів, що дозволяє створювати сучасний та адаптивний інтерфейс, який покращує користувацький досвід застосунку. Крім того, WPF підтримує прив’язку даних, що значно спрощує процес відображення та оновлення інформації, гарантуючи, що застосунок залишається синхронізованим з даними API.

WPF використовує XAML (eXtensible Application Markup Language) як декларативну мову для проєктування інтерфейсу користувача, що забезпечує чітке розділення між дизайном інтерфейсу та логікою програми. Це полегшує командну роботу з дизайнерами та покращує організацію коду.

Для організації коду в WPF було обрано шаблон проєктування MVVM (Model–View–ViewModel), який забезпечує чітке розділення відповідальностей між інтерфейсом користувача, бізнес-логікою та інтеграцією з API. Це робить код легшим для тестування, підтримки та масштабування.

1.5 Постановка задачі дослідження

Таким чином, інтеграція API є актуальним завданням для розробки сучасних програмних застосунків, оскільки вона дозволяє забезпечити доступ до зовнішніх сервісів та даних. Тому ставиться завдання розробки

методу інтеграції API з використанням сучасних підходів, таких як асинхронне програмування та шаблон проєктування MVVM.

Об’єктом дослідження є методи інтеграції API.

Метою дослідження є аналіз і порівняння різних методів інтеграції API в програмні застосунки з метою визначення найбільш ефективних підходів для створення високопродуктивних, масштабованих і зручних у використанні застосунків.

Задачі дослідження:

– огляд існуючих підходів до розробки програмних застосунків з інтегрованими API;

– дослідження основних мов програмування та фреймворків, які використовуються для інтеграції API (наприклад, C#, Python, JavaScript);

– аналіз типових проблем і викликів, що виникають під час інтеграції API;

– розробка прототипу програмного застосунка з інтеграцією API, що використовує сучасні шаблони проєктування (наприклад, MVVM).

– оцінка продуктивності та зручності підтримки розробленого рішення;

– підготовка рекомендацій щодо вибору методів інтеграції API на основі отриманих результатів дослідження.

2 АНАЛІЗ МЕТОДІВ ІНТЕГРАЦІЇ API У ПРОГРАМНІ ЗАСТОСУНКИ

2.1 Використання REST для інтеграції API

Representational State Transfer (REST) – це широко прийнятий архітектурний стиль для проектування розподілених систем, особливо для вебсервісів і мережевих застосунків. Представлений Роєм Філдінгом у 2000 році, REST надає набір рекомендацій, які використовують існуючі протоколи, в першу чергу HTTP, для забезпечення безперешкодної комунікації між клієнтами та серверами. Архітектура REST побудована навколо концепції ресурсів, які є сутностями, що можуть бути ідентифіковані за допомогою уніфікованих ідентифікаторів ресурсів (Uniform Resource Identifiers, URI). Ці ресурси можуть представляти що завгодно – від фрагмента даних до кінцевої точки сервісу.

Основні принципи REST включають у себе [11, 12]:

– бездержавний зв’язок: кожен клієнтський запит до сервера повинен містити всю інформацію, необхідну для розуміння і обробки запиту. Сервер не зберігає жодних даних про сеанс між запитами, що робить зв’язок бездержавним, що забезпечує кращу масштабованість і дозволяє серверу ефективно обробляти більшу кількість запитів;

– відокремлення клієнта від сервера, при цьому клієнт і сервер працюють окремо, клієнт відповідає за користувацький інтерфейс, а сервер обробляє дані та бізнес-логіку. Таке розділення забезпечує більшу гнучкість, оскільки зміни на клієнті або сервері можна вносити незалежно, не впливаючи на роботу іншого;

– кешування: відповіді від сервера явно позначаються як кешовані або некешовані. Якщо відповідь позначена як кешована, клієнти можуть зберігати її для подальшого використання, зменшуючи непотрібний мережевий трафік і підвищуючи продуктивність програми;

– уніфікований інтерфейс: послідовний і стандартизований підхід до комунікації між клієнтами і серверами з використанням фіксованого набору методів HTTP (GET, POST, PUT, DELETE), що спрощує взаємодію, дозволяючи клієнтам взаємодіяти з ресурсами у передбачуваний спосіб;

– багаторівнева система: архітектура дозволяє використовувати проміжні сервери, такі як балансувальники навантаження, проксі-сервери або шлюзи, які можуть виконувати функції кешування, ведення журналів або аутентифікації для підвищення безпеки, масштабованості та надійності системи.

Архітектурний стиль REST став популярним завдяки своїй простоті, гнучкості та масштабованості, що робить його найкращим вибором для багатьох сучасних веб та мобільних застосунків. Однією з головних переваг REST є його простота: він використовує стандартні методи HTTP і є відносно легким для реалізації та розуміння. Розробники можуть взаємодіяти з ресурсами, використовуючи знайомі концепції, такі як URI та коди стану HTTP. Природа RESTful API без стану сприяє масштабованості, оскільки кожен запит є незалежним, що робить його придатним для великомасштабних розподілених систем, де тисячі клієнтів можуть одночасно взаємодіяти з сервером. Ще однією перевагою є інтероперабельність, оскільки REST є адаптивним до мови, що дозволяє йому працювати з різними мовами програмування та платформами. Це дозволяє розробникам створювати системи, які можуть інтегруватися з іншими системами, незалежно від технологічного стеку. Крім того, REST пропонує гнучкість у представленні даних, підтримуючи різні формати, такі як JSON, XML або навіть звичайний текст, залежно від потреб клієнта.

Однак REST також має деякі недоліки. Принцип невизначеного стану, хоча і масштабований, може призвести до збільшення накладних витрат, оскільки кожен запит повинен містити всю необхідну інформацію, в тому числі дані для автентифікації. Це може призвести до надлишкової передачі даних, особливо в застосунках, які часто взаємодіють. Ще одним

обмеженням є відсутність підтримки реального часу, оскільки REST базується на моделі «запит – відповідь», що робить його менш придатним для застосунків, які потребують постійного потоку даних. Альтернативи, такі як WebSockets або GraphQL, часто краще підходять для спілкування в реальному часі. Крім того, REST може мати проблеми зі складними запитами або взаємозв'язками між ресурсами, оскільки йому не вистачає вбудованих механізмів для ефективного отримання пов'язаних даних, що часто вимагає декількох викликів API, які можуть збільшити затримку.

2.2 Використання SOAP для інтеграції API

SOAP (Simple Object Access Protocol) – це стандартизований протокол обміну повідомленнями, широко використовуваний у корпоративних середовищах для обміну структурованою інформацією між розподіленими системами. Він забезпечує сумісність різних платформ, мов програмування та операційних систем, що робить його чудовим вибором для складних і критично важливих застосунків. SOAP може працювати за різними транспортними протоколами, як-от HTTP, SMTP і TCP, що забезпечує гнучкість під час розгортання в різних мережевих середовищах [13].

SOAP спирається на XML (eXtensible Markup Language) для форматування повідомлень, що забезпечує сумісність із різними технологіями. XML дає змогу включати метадані, відомості про безпеку та обробку помилок безпосередньо в повідомлення, що робить SOAP придатним для застосунків, які потребують суворої відповідності, таких як охорона здоров'я, фінанси та урядові платформи. Однак залежність від XML призводить до збільшення розміру повідомлень і підвищення їхньої складності, що робить SOAP менш ідеальним для легких або мобільних застосунків.

SOAP суттєво відрізняється від REST (Representational State Transfer). У той час як SOAP використовує XML виключно для форматування

повідомлень, REST підтримує кілька форматів, таких як JSON і звичайний текст, що дозволяє розробникам вибирати найбільш ефективний формат. SOAP може зберігати стан між сеансами клієнта і сервера, що є перевагою для складних робочих процесів, тоді як REST не має стану і обробляє кожен запит незалежно. Крім того, SOAP пропонує вбудовану підтримку розширених функцій, таких як WS-Security для безпечного зв'язку, WS-ReliableMessaging для надійного обміну даними та підтримку розподілених транзакцій. Ці функції роблять SOAP ідеальним для галузей, що вимагають суворих стандартів, безпеки та надійності, в той час як REST краще підходить для легких веб та мобільних застосунків.

Однією з найбільших переваг SOAP є його надійність та безпека. Завдяки WS-Security він підтримує шифрування, цифрові підписи та автентифікацію на основі токенів, забезпечуючи захист даних і запобігаючи несанкціонованому доступу. WS-Reliable Messaging гарантує точну доставку повідомлень навіть у разі перебоїв у мережі, що робить його критично важливим для таких застосунків, як платіжні системи та обробка замовлень. SOAP також підтримує розподілені транзакції, забезпечуючи узгодженість в операціях, де кілька процесів повинні бути успішними або невдалими разом.

Структура SOAP-повідомлень складається з конверта, який інкапсулює повідомлення, заголовка для необов'язкових метаданих, таких як інформація про автентифікацію або маршрутизацію, тіла, що містить власне зміст повідомлення, і елемента несправності для повідомлення про помилки. Ці структуровані повідомлення визначаються і описуються за допомогою WSDL (Web Services Description Language), мови на основі XML, яка визначає інтерфейс SOAP-сервісу, включаючи операції, вхідні та вихідні параметри, а також кінцеві точки доступу. WSDL діє як формальний контракт між клієнтом і сервером, забезпечуючи чіткий зв'язок і дозволяючи розробникам автоматично генерувати код на стороні клієнта.

Таким чином, SOAP – це потужний протокол, розроблений для безпечної, надійної та сумісної зі стандартами комунікації в застосунках

корпоративного рівня. Хоча він може бути не таким легким і гнучким, як REST, його всеосяжний набір функцій робить його незамінним для сценаріїв, де безпека і надійність мають першорядне значення.

2.3 Використання GraphQL для інтеграції API

GraphQL – це сучасна мова запитів до даних, розроблена компанією Facebook у 2015 році для подолання обмежень традиційних парадигм API, таких як REST. На відміну від REST, який покладається на кілька кінцевих точок для різних ресурсів, GraphQL пропонує єдину, уніфіковану кінцеву точку для всіх взаємодій з даними. Це дозволяє клієнтам точно вказувати дані, які їм потрібні, в одному запиті, значно підвищуючи ефективність, зменшуючи передачу даних і спрощуючи загальний процес розробки.

GraphQL працює на основі схематичної архітектури, яка чітко визначає типи даних, їх взаємозв’язки та доступні операції. Ця схема діє як контракт між клієнтом і сервером, надаючи розробникам чіткі очікування щодо структурування даних і доступу до них. Клієнти пишуть запити в синтаксисі, схожому на JSON, який є одночасно інтуїтивно зрозумілим і потужним, що дозволяє створювати висококастомізовані запити до даних [14, 15].

Ключовою особливістю GraphQL є його здатність усувати надмірну та недостатню вибірку, поширені проблеми в RESTful API. Надмірна вибірка відбувається, коли клієнти отримують більше даних, ніж потрібно, збільшуючи використання пропускної здатності і сповільнюючи продуктивність. Недостатня вибірка, з іншого боку, змушує клієнтів робити кілька запитів, щоб зібрати всі необхідні дані. GraphQL вирішує обидві проблеми, дозволяючи клієнтам запитувати тільки ті поля, які їм потрібні, не більше і не менше. Така точність особливо корисна для застосунків зі складними вимогами до даних або обмеженими ресурсами, таких як мобільні застосунки.

Якщо порівнювати GraphQL з REST, то його переваги виходять за рамки гнучкості. REST API часто вимагають створення декількох версій кінцевих точок для внесення змін або додаткових функцій, що призводить до збільшення витрат на обслуговування. GraphQL дозволяє уникнути цього, використовуючи єдину динамічну кінцеву точку, яка адаптується до мінливих потреб, що полегшує розвиток API з часом. Крім того, GraphQL може агрегувати дані з декількох джерел в одному запиті, значно зменшуючи кількість запитів до сервера і покращуючи час відгуку. Однак ця додаткова гнучкість створює потенційні проблеми, такі як більш складна логіка на стороні сервера і необхідність ефективного управління продуктивністю запитів, щоб запобігти надмірно вкладеним або ресурсоємним запитам.

Можливості GraphQL в режимі реального часу є ще однією ключовою перевагою, особливо для застосунків, що вимагають динамічних оновлень, таких як інструменти для спільної роботи, спортивні стрічки в реальному часі або інформаційні панелі фондового ринку. Використовуючи підписки, GraphQL може передавати зміни даних у реальному часі безпосередньо клієнтам, усуваючи необхідність частого опитування і зменшуючи навантаження на сервер. Хоча ця функція покращує взаємодію з користувачем, реалізація функціональності в режимі реального часу вимагає додаткової конфігурації і може збільшити складність розробки в порівнянні з традиційною моделлю REST «запит–відповідь».

GraphQL стала популярним вибором для сучасних вебзастосунків завдяки своїй адаптивності та ефективності. Наприклад, у платформах електронної комерції GraphQL дозволяє розробникам отримувати інформацію про товар, стан запасів та відгуки клієнтів в одному запиті, покращуючи взаємодію з користувачем за рахунок зменшення часу завантаження сторінки. У застосунках для соціальних мереж GraphQL спрощує процес завантаження динамічних стрічок, профілів користувачів, сповіщень і повідомлень, забезпечуючи безперебійну та швидку роботу користувачів. Такі компанії, як GitHub, Shopify та Twitter, впровадили

GraphQL для своїх API, продемонструвавши її ефективність у великомасштабних середовищах з високим трафіком.

Незважаючи на свої численні переваги, GraphQL не позбавлена проблем. Підхід до розробки на основі схеми вимагає ретельного планування і може зайняти багато часу для команд, які не знайомі з цією технологією. Крім того, управління складністю запитів є критично важливим, оскільки глибоко вкладені або неоптимізовані запити можуть знизити продуктивність сервера. Такі інструменти, як аналізатори запитів, механізми кешування та обмежувачі швидкості часто необхідні для зменшення цих ризиків.

Отже, GraphQL – це потужний і універсальний підхід до розробки API, що пропонує неперевершену гнучкість, ефективність і можливості роботи в реальному часі. Його здатність обробляти складні взаємодії даних і вимоги, що постійно змінюються, робить його чудовим вибором для сучасних застосунків. Однак розробники повинні зважити її переваги з потенційними проблемами впровадження та підтримки. З ростом популярності GraphQL продовжує переосмислювати те, як розробляються API, встановлюючи новий стандарт для запитів до даних у веб та мобільній розробці.

2.4 Використання WebSockets для інтеграції API

WebSockets – це комунікаційний протокол, призначений для повнодуплексного двостороннього зв’язку між клієнтом і сервером через одне довготривале з’єднання. На відміну від традиційних моделей запитів–відповідей на основі HTTP, WebSockets забезпечують безперервне з’єднання, що дозволяє безперешкодно передавати дані в обох напрямках [16, 17]. Ця можливість робить WebSockets особливо придатними для застосунків у режимі реального часу, де низька затримка та миттєве оновлення даних є критично важливими.

Протокол WebSocket починається з початкового HTTP-рукостискання, під час якого клієнт надсилає запит на встановлення з’єднання WebSocket.

Після успішного рукостискання з’єднання перетворюється на WebSocket і залишається відкритим доти, доки клієнт або сервер явно не закриє його. Така постійна природа усуває необхідність у повторюваних рукостисканнях HTTP, значно зменшуючи мережеві накладні витрати і дозволяючи ефективно обмінюватися даними.

WebSockets стали наріжним каменем багатьох застосунків, що працюють у режимі реального часу. Наприклад, у системах чату WebSockets полегшують миттєву доставку повідомлень, забезпечуючи безперебійну та швидку роботу користувачів. У фінансових торгових платформах WebSockets дозволяють в режимі реального часу оновлювати ціни на акції, угоди та ринкові дані, надаючи трейдерам точну, актуальну інформацію, необхідну для прийняття важливих рішень. Багатокористувацькі онлайн ігри використовують WebSockets для взаємодії гравців у реальному часі, оновлення та синхронізації між клієнтами і серверами. Інші випадки використання включають інструменти для спільного редагування документів, прямі трансляції спортивних подій або новин, а також застосунки Інтернету речей, де пристрої безперервно передають і приймають дані.

Переваги WebSockets є переконливими, особливо для застосунків, які вимагають високої швидкості реагування та ефективності. По-перше, WebSockets усувають необхідність опитування, коли клієнт повторно надсилає HTTP-запити для перевірки наявності оновлень, дозволяючи серверу надсилати оновлення безпосередньо клієнту, коли з’являються нові дані. Це зменшує навантаження на сервер, економить пропускну здатність і мінімізує затримки, підвищуючи загальну продуктивність програми.

По-друге, зв’язок з низькою затримкою, що забезпечується WebSockets, ідеально підходить для чутливих до часу завдань, таких як ігри, живий чат або фінансова торгівля. Підтримуючи єдине постійне з’єднання, WebSockets також зменшують накладні витрати, пов’язані з багаторазовим встановленням і розривом HTTP-з’єднань, що призводить до більш плавного зв’язку і кращого використання ресурсів.

Крім того, WebSockets є універсальними і можуть передавати будь-який тип даних, включаючи текст, JSON, бінарні дані або навіть медіапотоки. Така гнучкість робить їх придатними для широкого спектру випадків використання і дозволяє розробникам створювати ефективні, масштабовані рішення для обміну даними в реальному часі.

Незважаючи на свої переваги, WebSockets створюють певні проблеми, які розробники повинні вирішувати. Однією з головних проблем є масштабованість. Управління великою кількістю одночасних з'єднань WebSocket може призвести до перевантаження серверних ресурсів, особливо в застосунках з великим обсягом трафіку. Щоб пом'якшити цю проблему, розробники часто впроваджують балансування навантаження, пули з'єднань та ефективні стратегії розподілу ресурсів.

Іншою проблемою є обробка помилок і відновлення з'єднання. З'єднання WebSocket можуть перериватися через проблеми з мережею або збої в роботі сервера, що вимагає надійної логіки для виявлення втрати з'єднання і відновлення з'єднання без переривання роботи користувача.

Безпека є ще одним важливим фактором. На відміну від HTTP, WebSockets не підтримують стандартизовані механізми, такі як HTTPS, для шифрування або автентифікації. Розробники повинні впроваджувати безпечні протоколи WebSocket (WSS) разом з надійними процесами автентифікації та авторизації, щоб захистити конфіденційні дані та запобігти несанкціонованому доступу.

Інтегруючи WebSockets з API, розробники розкривають потенціал взаємодії в реальному часі, забезпечуючи безперебійний зв'язок між клієнтами та серверами. Наприклад, в API для застосунків чату WebSockets дозволяють користувачам обмінюватися повідомленнями без затримок, в той час як API для трансляцій спортивних подій або новин можуть передавати користувачам контент в режимі реального часу, не вимагаючи від них оновлювати екран.

Таким чином, WebSockets представляють сучасний підхід до інтеграції API для застосунків у режимі реального часу, пропонуючи неперевершену продуктивність, гнучкість і зручність для користувачів. Хоча вони створюють складнощі у впровадженні та підтримці, ці проблеми перекриваються перевагами, які вони приносять динамічним, інтерактивним системам. Ретельно розробляючи та оптимізуючи рішення на основі WebSocket, розробники можуть створювати застосунки, які відповідають вимогам сучасного світу, що працює в режимі реального часу і керується даними.

2.5 Інтеграція API з обраною архітектурою застосунку

Інтеграція API в застосунок є критично важливим процесом, який вимагає ретельного розгляду архітектурних патернів, що використовуються. Вибір відповідної архітектури має важливе значення для забезпечення безперебійної взаємодії між різними компонентами, підтримки кодової бази та масштабованості в міру зростання застосунку. Двома найбільш поширеними архітектурними патернами для створення застосунків з інтегрованими API є MVC (Model–View–Controller) та MVVM (Model–View–ViewModel). Обидва патерни забезпечують структурований підхід до розробки застосунків, але вони задовольняють різні потреби та сценарії [18–20].

Архітектурний патерн MVC організовує застосунок у три взаємопов’язані компоненти. Модель відповідає за управління даними та бізнес-логікою застосунку. Вона взаємодіє з базою даних або будь-яким джерелом даних, забезпечуючи пошук, оновлення та збереження даних. Компонент View відповідає за користувацький інтерфейс і рівень презентації, забезпечуючи інтуїтивно зрозумілу взаємодію користувача з застосунком. Нарешті, контролер виступає посередником між моделлю та представленням.

Він обробляє дані, введені користувачем, маніпулює даними за допомогою моделі і відповідно оновлює представлення. Такий розподіл завдань допомагає впорядкувати процеси розробки та тестування, роблячи застосунок більш керованим.

З іншого боку, патерн MVVM підкреслює більш чітке розділення між користувацьким інтерфейсом і базовою логікою застосунку. У цій структурі модель все ще керує даними та бізнес-логікою, в той час як представлення слугує візуальним представленням цих даних. Унікальним аспектом MVVM є ViewModel, яка виступає в ролі моста між моделлю та представленням. Вона використовує механізми зв'язування даних для їх синхронізації, гарантуючи, що зміни в даних автоматично відображаються в інтерфейсі користувача. Ця архітектура особливо корисна для десктопних застосунків, створених за допомогою WPF (Windows Presentation Foundation), оскільки вона використовує потужні можливості WPF для прив'язки даних.

Інтеграція API в WPF-застосунок включає в себе кілька різних рівнів, кожен з яких служить певній меті. На рівні моделі розробники реалізують клієнтську логіку API, яка включає в себе створення HTTP-запитів, таких як GET і POST, для отримання або відправлення даних на сервер. Наприклад, можна розробити клас сервісу, який отримуватиме дані користувача з API і розбиратиме відповіді JSON на чітко визначені моделі даних, які програма зможе легко використовувати [21–23].

Рівень ViewModel слугує посередником між логікою API та інтерфейсом користувача. Він обробляє дані, отримані з API, застосовує будь-які необхідні бізнес-правила та розкриває властивості, до яких може прив'язуватися представлення. Таке розділення дозволяє створити чисту модель взаємодії, де інтерфейс користувача автоматично оновлюється при зміні даних, покращуючи користувацький досвід.

Рівень представлення – це місце, де візуальні аспекти програми реалізуються за допомогою XAML (eXtensible Application Markup Language – розширювана мова розмітки застосунків). Цей рівень зв'язується з

властивостями, виставленими ViewModel, дозволяючи користувачам динамічно взаємодіяти з застосунком. Наприклад, у застосунку WPF, який інтегрує погодний API, модель може отримувати погодні дані з API у форматі JSON. Потім ViewModel обробляє ці дані, перетворюючи їх у формат, придатний для відображення, наприклад, витягуючи поточну температуру і погодні умови. Нарешті, Представлення представило б цю інформацію у зручному для користувача інтерфейсі, наприклад, на інформаційній панелі, яка оновлюється в режимі реального часу, щоб показати найсвіжішу інформацію про погоду.

Використання патерну MVVM у поєднанні з WPF для інтеграції API має багато переваг. Однією з головних переваг є чіткий розподіл обов'язків, який гарантує, що кожен компонент програми має визначену роль і відповідальність. Такий модульний підхід полегшує обслуговування і тестування, оскільки розробники можуть зосередитися на окремих компонентах, не впливаючи на весь застосунок. Прив'язка даних спрощує процес відображення даних API в інтерфейсі користувача, що зменшує кількість шаблонного коду, необхідного для управління оновленнями вручну. Крім того, масштабованість архітектури MVVM дозволяє розробникам легко додавати нові функції або модифікувати існуючі без значного рефакторингу кодової бази.

Незважаючи на ці переваги, при використанні патерну MVVM для інтеграції API можуть виникнути певні труднощі. Початкове налаштування архітектури MVVM може вимагати більше зусиль і розуміння у порівнянні з більш простими патернами, що може призвести до більш крутої кривої навчання для нових розробників. Крім того, через прив'язку даних можуть виникати проблеми з продуктивністю, особливо при роботі з великими наборами даних або коли потрібні часті оновлення з API. Оптимізація прив'язки даних і забезпечення ефективної роботи з ними стають критично важливими для підтримки продуктивності застосунків.

Ефективно використовуючи сильні сторони WPF і патерну MVVM, розробники можуть створювати надійні, зручні для користувача застосунки, які легко інтегрують API. Такий підхід не тільки покращує взаємодію з користувачем, але й гарантує, що застосунок залишається гнучким і масштабованим, адаптуючись до майбутніх вимог і технологічного прогресу. В результаті, поєднання WPF і MVVM є потужнин рішення для розробки сучасних застосунків з інтегрованими API, що робить його кращим вибором серед розробників у цій галузі.

2.6 Проблеми та виклики інтеграції API в реальних умовах

Інтеграція API у застосунки ставить перед розробниками низку викликів і проблем, які вони повинні вирішити, щоб забезпечити безперебійну функціональність і задоволеність користувачів. Однією з головних проблем при роботі з API є управління підтримкою API, особливо щодо версій та зворотної сумісності [24]. У міру розвитку API часто зазнають змін, які можуть вплинути на існуючі інтеграції. Розробники повинні переконатися, що новіші версії API не порушують існуючу функціональність застосунків, які покладаються на старіші версії. Це вимагає ретельного планування та впровадження стратегій, які сприяють зворотній сумісності, дозволяючи застосункам функціонувати коректно, навіть коли відбуваються оновлення API. Це може включати використання попереджень про застарілість, підтримку застарілих кінцевих точок і надання чіткої документації, яка допоможе розробникам перейти на новіші версії.

Іншою важливою проблемою є управління лімітами швидкості API або квотами, які обмежують кількість запитів, які програма може зробити до API протягом певного періоду часу. Перевищення цих лімітів може призвести до відхилення запитів або дроселювання, що призводить до перебоїв у роботі сервісу. Щоб уникнути цих обмежень, розробники можуть реалізувати кілька

стратегій. Один із поширених підходів – кешування відповідей, щоб зменшити потребу в повторних викликах API. Зберігаючи дані, до яких часто звертаються, локально, застосунки можуть мінімізувати кількість запитів, що надсилаються до API. Крім того, оптимізація викликів API для об’єднання декількох запитів в один виклик може ще більше зменшити частоту взаємодії з API. Використання вебхуків або альтернативних механізмів дозволяє оновлювати дані в режимі реального часу без постійного опитування API, що також може допомогти залишатися в межах квоти. Розуміння політики постачальника API щодо лімітів швидкості має важливе значення для розробки ефективної стратегії інтеграції, яка максимізує ефективність при дотриманні цих обмежень.

Обробка помилок – ще один важливий аспект інтеграції API, який потребує ретельного розгляду. Зовнішні API-сервіси можуть зазнавати перебоїв у роботі, змінювати формати відповідей або інші проблеми, які можуть призвести до неочікуваних помилок. Розробники повинні впровадити надійні механізми обробки помилок, щоб ефективно керувати цими сценаріями. Це включає в себе стратегії повторних спроб невдалих запитів, реєстрацію помилок для подальшого аналізу та надання користувачам змістовного зворотного зв’язку, коли щось йде не так. Впровадження стратегій експоненціального відступу для повторних спроб може допомогти уникнути перевантаження API запитами під час збою. Крім того, розробка користувацьких інтерфейсів, які можуть ефективно обробляти стани завантаження або повідомлення про помилки, може покращити загальний користувацький досвід. Передбачаючи потенційні помилки і плануючи реакцію на них, розробники можуть створювати застосунки, які залишаються відмовостійкими і можуть відновлюватися після тимчасових збоїв без значних перебоїв.

Крім того, міркування безпеки є першочерговими при інтеграції API. API часто обробляють конфіденційні дані, і захист цих даних має вирішальне значення. Розробники повинні переконатися, що вони використовують

безпечні протоколи зв’язку, такі як HTTPS, для шифрування даних під час передачі. Крім того, впровадження належних механізмів автентифікації та авторизації, таких як OAuth або ключі API, має важливе значення для запобігання несанкціонованому доступу до кінцевих точок API. Регулярний перегляд політик і практик безпеки необхідний для усунення нових загроз і вразливостей [25]

Нарешті, документація відіграє життєво важливу роль в успішній інтеграції API. Добре задокументовані API надають розробникам необхідні вказівки, щоб зрозуміти, як реалізовувати функції та вирішувати потенційні проблеми. Сюди входять чіткі приклади запитів і відповідей, пояснення кодів помилок і найкращі практики ефективного використання API. Оновлення документації з урахуванням будь-яких змін в API гарантує, що розробники завжди матимуть під рукою найсвіжішу інформацію, зменшуючи плутанину і покращуючи процес розробки [26, 27].

Таким чином, інтеграція API в реальні застосунки передбачає вирішення різних проблем, включаючи підтримку API за допомогою версій і зворотної сумісності, ефективне управління квотами запитів, реалізацію надійних стратегій обробки помилок, забезпечення безпеки і ведення вичерпної документації. Проактивно виявляючи та усуваючи ці проблеми, розробники можуть підвищити надійність та продуктивність своїх застосунків, одночасно використовуючи весь потенціал інтегрованих API. Зрештою, вдумливий підхід до цих проблем призведе до покращення користувацького досвіду та більш успішного розгортання застосунків.

3 ПРАКТИЧНА РОЗРОБКА

3.1 Налаштування оточення

Спершу ніж розпочати розробку програми, необхідно налаштувати робоче середовище. Налаштування включає в себе отримання ключа доступу до API, і встановлення всіх потрібних пакетів для IDE Visual Studio 2019.

Щоб забезпечити чесне використання та стабільність системи, Riot API обмежує кількість запитів, які розробники можуть робити протягом певного періоду часу. Щоб отримати доступ до даних, розробники повинні аутентифікувати свої запити за допомогою ключів API, наданих Riot.

Для того, щоб підвищити швидкість запитів та отримати змогу звертатися до API без обмеження у 24 години, треба отримати статичний ключ до API. Його можна отримати, тільки якщо продукт буде зареєстрований на сайті Riot API.

Для реєстрації необхідно зайти на сайт API (developer.riotgames.com), створити акаунт Riot або використати створений. Далі потрібно зареєструвати продукт (рис. 3.1). Ключ, який дається при аутентифікації на сайті, буде мінятися кожні 24 години.

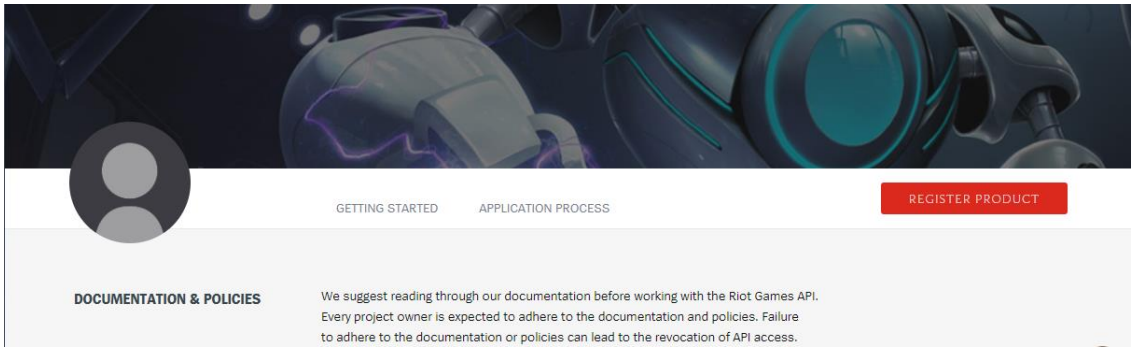


Рисунок 3.1 – Реєстрація продукту

Тепер потрібно вибрати тип продукту (рис. 3.2). Вибираємо Personal API Key.

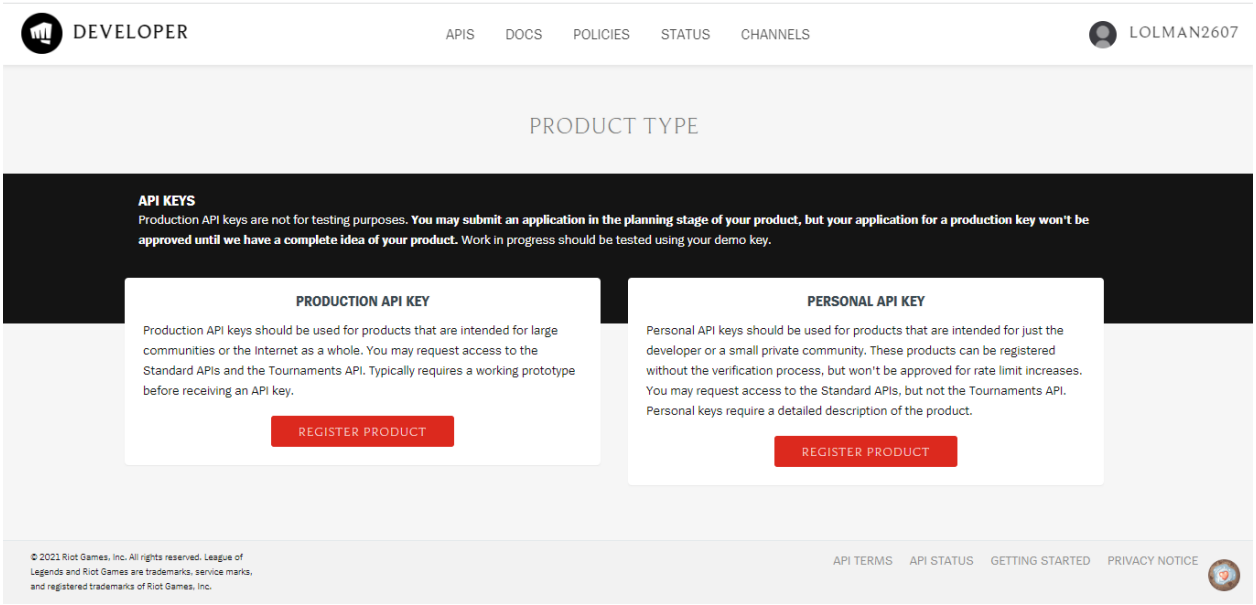


Рисунок 3.2 – Вибір типу ключа

Далі здійснюється перехід на нову сторінку та заповнення обов’язкових полів: назва проєкту, опис, групу розробки та фокусування на одну із ігор Riot (рис. 3.3).

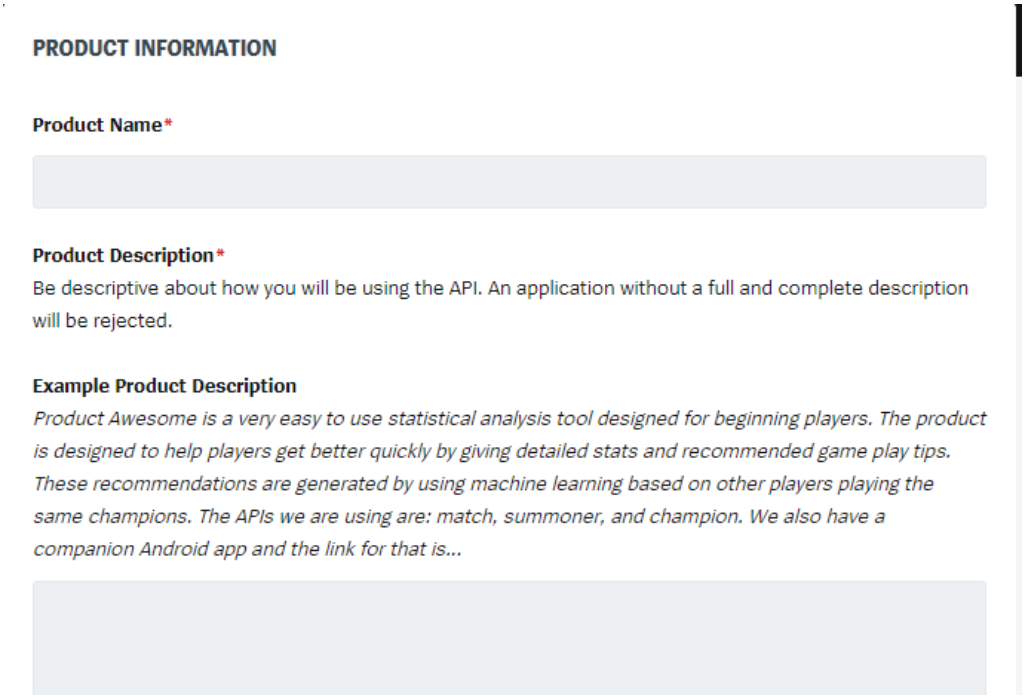


Рисунок 3.3 – Заповнення інформації по проєкту

Обробка запита на створення ключа для проєкту може зайняти певний час. Сповіщення приходить на електронну пошту, прив’язану до облікового запису Riot.

Після отримання сповіщення отримаємо ключ (рис. 3.4).

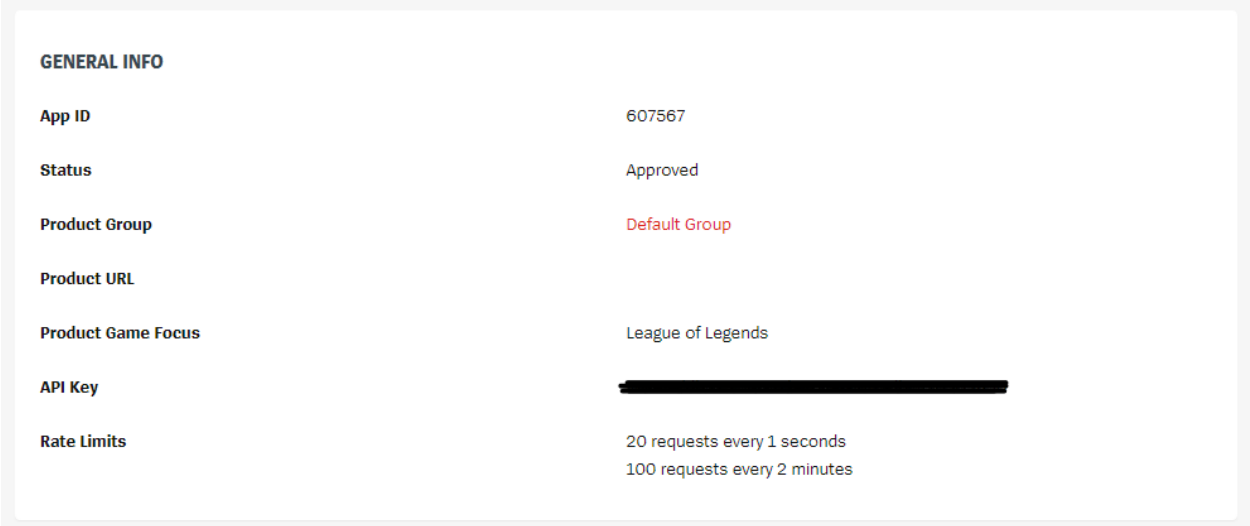


Рисунок 3.4 – Вибір типу ключа

Після отримання ключа можна переходити до налаштування IDE для написання застосунку.

При написанні програми використовується Visual Studio 2019, оскільки Visual Studio 2019 повністю налаштована під зручну розробку програм з використання .NET.

Для написання програм на WPF потрібно встановити додатковий пакет (рис. 3.5) бібліотек в інсталері (рис. 3.6) для Visual Studio 2019.

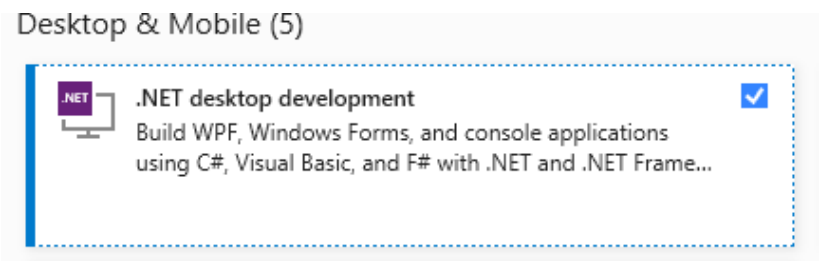


Рисунок 3.5 – Пакет для розробки комп’ютерних застосунків

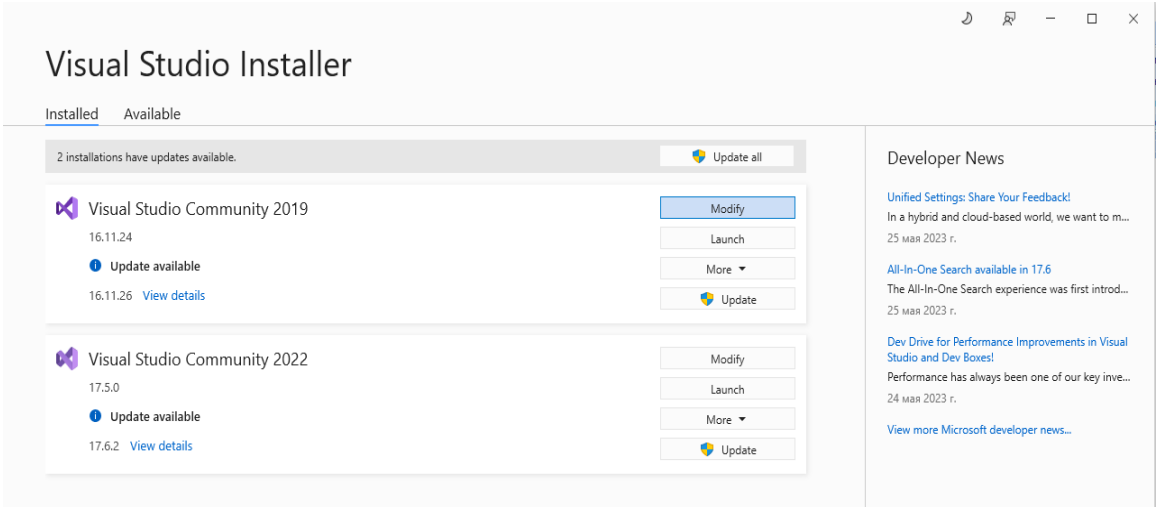


Рисунок 3.6 – Установник Visual Studio 2019

Після налаштування оточення можна переходити до розробки.

3.2 Проєктування застосунка

Для створення проєкту потрібно відкрити Visual Studio та створити WPF проєкт (рис 3.7).

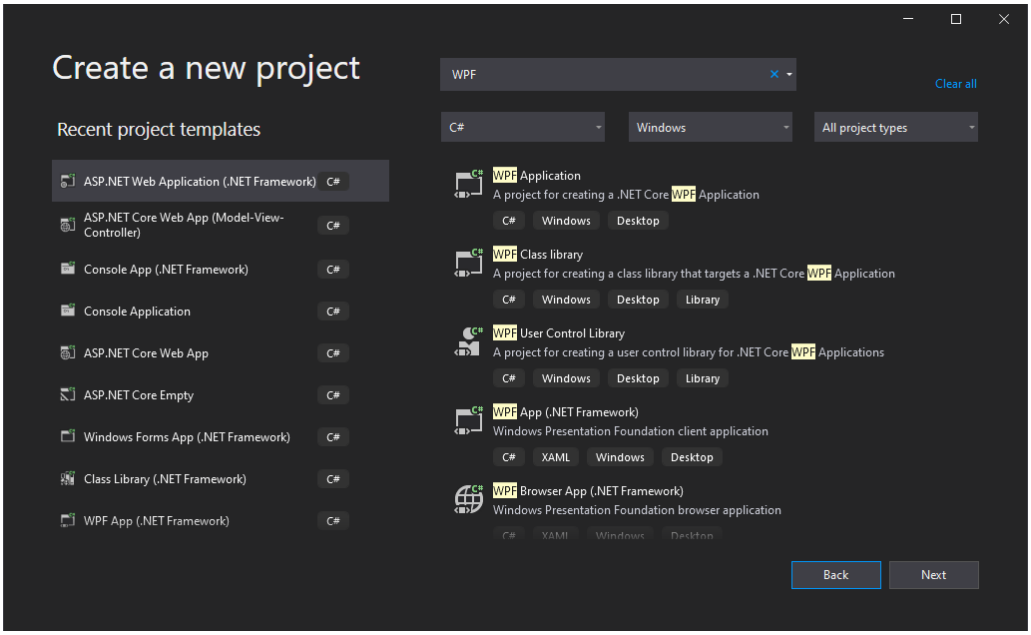


Рисунок 3.7 – Створення WPF проєкту

Після того, як створився проєкт, необхідно створити репозиторій на GitHub для зручного зберігання та відслідковування змін в проєкті.

Далі потрібно створити папку для збереження класів, які будуть взаємодіяти з Riot API (рис. 3.8). Також у цю папку додається текстовий файл з статичним ключем, що отриманий від Riot.

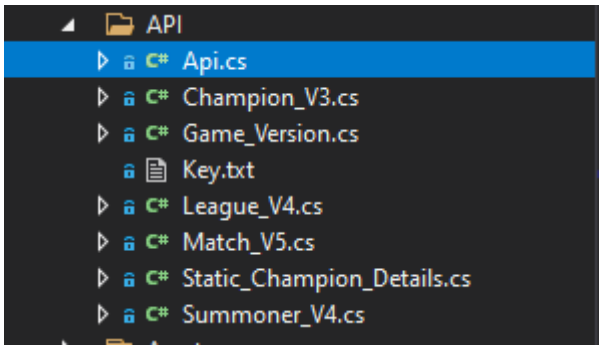


Рисунок 3.8 – Створення папки Api та файлу з ключем

Створимо базовий клас під назвою Api (рис 3.9), від якого будуть унаслідуватись усі класи для взаємодії з API, тому що запити розподілені на групи за тематикою. Щоб не робити багато однакових класів, логічно використовувати один із принципів ООП – наслідування.

В класі Api є дві властивості та чотири методи.

Властивості:

1. «Region» – властивість типу string, у котрій буде зберігатись регіон. Всі запити повинні містити в собі регіон, за яким треба буде шукати інформацію.

2. «Key» – властивість типу string, де буде зберігатись ключ до API, який вже збережений у файл.

Методи:

– конструктор: приймає рядок з регіоном. Метод призначає строку з регіоном до властивості Region та викликає метод GetKey, тим самим призначає значення властивості Key.

– GET: створює екземпляр класу HttpClient та викликає метод GetAsync об’єкта класу, та вертає властивість Result цього об’єкта.

– GetURI: приймає рядок з шляхом. Шлях є унікальним рядком для кожного запиту з API. Вертає повний рядок запиту до API.

– GetKey: приймає рядок з шляхом до файлу, де лежить ключ до API. Повертає рядок з ключем.

```
13 references
public class Api
{
    4 references
    protected string Key { get; set; }
    2 references
    private string Region { get; set; }

    6 references
    public Api(string region)
    {
        Region = region;
        Key = GetKey("../../API/Key.txt");
    }

    7 references
    protected HttpResponseMessage GET(string URL)
    {
        using (HttpClient client = new HttpClient())
        {
            var result = client.GetAsync(URL);
            result.Wait();

            return result.Result;
        }
    }
    9 references
    protected virtual string GetURI(string path)
    {
        return "https://" + Region + ".api.riotgames.com/lol/" + path + "?api_key=" + Key;
    }
    1 reference
    public string GetKey(string path)
    {
        StreamReader sr = new StreamReader(path);
        return sr.ReadToEnd();
    }
```

Рисунок 3.9 – Клас Api

Тепер можна створити головне вікно застосунку (рис. 3.10), в якому буде випадаючий список з регіонами та можливість вписати логін аккаунту, за цим логіном можна знайти інформацію по користувачу з гри League of Legends. Також там будуть кнопки для переходу на вікно з ротацією

чемпіонів, кнопка виходу з програми та кнопка переходу до вікна, де буде відображатися інформація по чемпіонах.

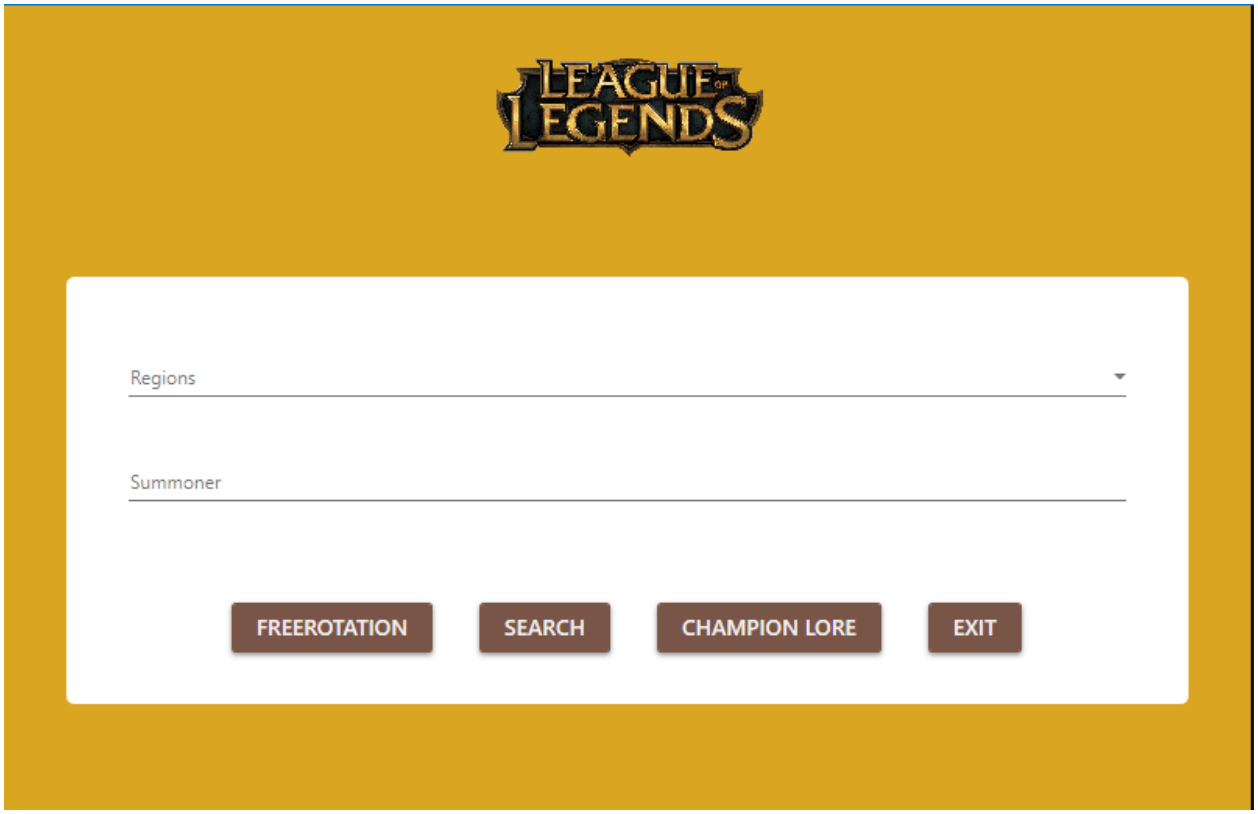


Рисунок 3.10 – Головне вікно застосунку

Після цього створюється вікно, на якому буде відображатись базова інформація за аккаунтом (рис.3.11). Тут будуть відображатися ім’я, ранг (якщо він є), перемоги та програші (якщо у гравця є ранг), іконка гравця у грі та його рівень.

Також додаємо кнопку Details, за допомогою якої можна буде відкрити сторінку, де зберігатиметься інформація з останніх 3 ігор та продубльовано інформацію з попереднього вікна та буде кнопка повернення на головне вікно.

Тепер необхідно створити класи моделі для сторінки з базовою інформацією.

Спочатку створюється модель SummonerDTO, в ній буде декілька властивостей:

– ProfileIconId: номер іконки гравця, за яким вона буде відображатися;

– Name: ім’я граця;

– SummonerLevel: рівень гравця;

– Id: номер гравця на сервері;

– Puuid: ідентифікатор гравця.

Модель PositionDTO відповідає за ранг та інформацію по виграшах, характеризується наступними властивостями:

– Tier: ранги розбиті на тири в кожному ранзі по 4 тира;

– Rank: ранги від заліза до челенджера;

– Wins: кількість виграшів у рангових іграх за цей сезон;

– Losses: кількість програшів у рангових іграх за цей сезон;

– QueueType: тип черги (в іграх потрібен, щоб знайти інформацію, яка зазначена зверху).

–

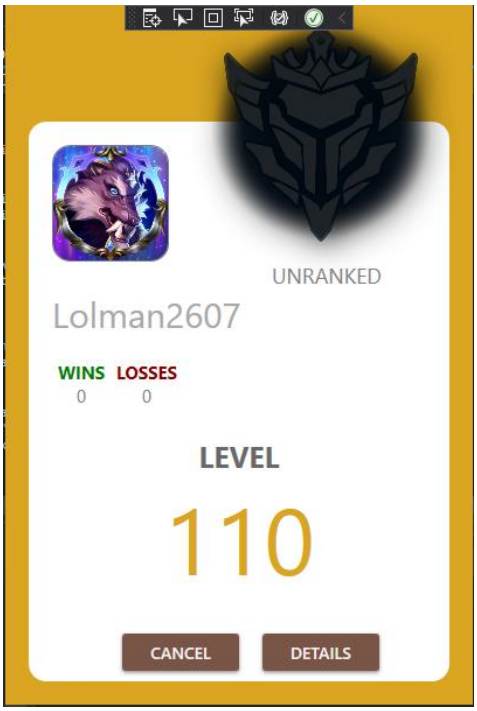


Рисунок 3.11 – Вікно з загальною інформацією по гравцю

Далі створюються класи для роботи з API, які будуть унаслідуватись від класу Api, що був написаний до цього.

Першим буде клас Summoner_V4 (рис. 3.12), який унаслідується від API. В нього буде два методи:

– конструктор: приймає регіон. Буде використовувати базовий конструктор класу Api.

– GetSummonerByName: приймає ім’я гравця. Цей метод буде отримувати JSON файл з API та конвертувати його в потрібний оберт класу, повертає об’єкт моделі SummonerDto.

```
3 references
public class Summoner_V4 : Api
{
    1 reference
    public Summoner_V4(string region) : base(region)
    {
    }

    1 reference
    public SummonerDTO GetSummonerByName(string SummonerName)
    {
        string path = "summoner/v4/summoners/by-name/" + SummonerName;
        var response = GET(GetURI(path));
        string content = response.Content.ReadAsStringAsync().Result;
        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            return JsonConvert.DeserializeObject<SummonerDTO>(content);
        }
        else
        {
            return null;
        }
    }
}
```

Рисунок 3.12 – Клас Summoner_V4

Другим буде клас League_V4 (рис. 3.13), який унаслідується від API. В нього буде два методи:

– конструктор: приймає регіон, буде використовувати базовий конструктор класу Api;

– GetPositions: приймає id гравця, буде отримувати JSON файл з API та конвертувати його в оберт класу, що нам потрібен. Повертає список об’єктів моделі PositionDto.

Для того, щоб зберігати деякі дані, які можуть стати в пригоді в подальшій розробці, створюється статичний клас Constants (рис. 3.14).

```
3 references
public class League_V4 : Api
{
    1 reference
    public League_V4(string region) : base(region)
    {

    }

    1 reference
    public List<PositionDTO> GetPositions(string summonerId)
    {
        string path = "league/v4/entries/by-summoner/" + summonerId;
        var response = GET(GetURI(path));
        string content = response.Content.ReadAsStringAsync().Result;
        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            return JsonConvert.DeserializeObject<List<PositionDTO>>(content);
        }
        else
        {
            return null;
        }
    }
}
```

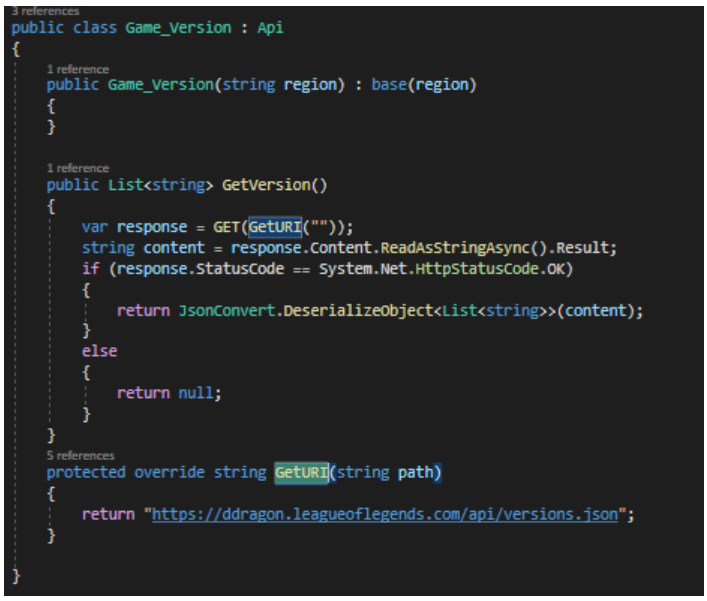
Рисунок 3.13 – Клас League_V4

```
55 references
public static class Constants
{
    7 references
    public static SummonerDTO Summoner { get; set; }
    3 references
    public static PositionDTO Position { get; set; }
    2 references
    public static ChampionInfo Champions { get; set; }
    5 references
    public static string ChampName { get; set; }
    7 references
    public static string Region { get; set; }
    31 references
    public static string Version { get; set; }

}
```

Рисунок 3.14 – Клас Constants

Ще потрібен клас, який буде доставати з API версію гри, оскільки версія гри змінюється кожні 2 тижня та потрібно отримувати свіжу інформацію. Тож необхідно створити клас Game_Version (рис 3.15), який буде наслідувати клас Api.



```
3 references
public class Game_Version : Api
{
    1 reference
    public Game_Version(string region) : base(region)
    {
    }

    1 reference
    public List<string> GetVersion()
    {
        var response = GET(GetURI(""));
        string content = response.Content.ReadAsStringAsync().Result;
        if (response.StatusCode == System.Net.HttpStatusCode.OK)
        {
            return JsonConvert.DeserializeObject<List<string>>(content);
        }
        else
        {
            return null;
        }
    }
    5 references
    protected override string GetURI(string path)
    {
        return "https://ddragon.leagueoflegends.com/api/versions.json";
    }

}
```

Рисунок 3.15 – Клас Game_Version

В класі Game_Version три методи:

– конструктор: приймає регіон, буде використовувати базовий конструктор класу Api;

– GetVersion: цей метод буде отримувати JSON файл з API та конвертувати його в список рядків з версіями гри;

– GetURI: перевизначення метода базового класу Api, щоб цей метод повертав тільки рядок з запитом на версії гри.

Далі створюються два контролери, це будуть класи, що зв’язують моделі та працюють з API.

Першим буде клас ControllerMain (рис. 3.16). В нього буде два методи:

– GetSummoner: приймає ім’я гравця. Цей метод створює екземпляр класу Summoner_V4 та викликає метод GetSummonerByName, зберігає об’єкт моделі до класу Constants. Повертає значення типу bool в залежності від того, чи отриманий об’єкт SummonerDto, чи ні;

– GetVersionOfGame: цей метод створює екземпляр класу Game_Version та викликає метод GetVersion. Зберігає рядок з версією до класу Constants та повертає значення типу bool в залежності від того, чи отриманий рядок, чи ні.

```
2 references
class ControllerMain
{
    1 reference
    public bool GetSummoner( string summonerName)
    {
        Summoner_V4 summoner_V4 = new Summoner_V4(Constants.Region);
        var summoner = summoner_V4.GetSummonerByName(summonerName);

        Constants.Summoner = summoner;

        return summoner != null;
    }

    2 references
    public bool GetVersionOfGame()
    {
        Game_Version game_Version = new Game_Version(Constants.Region);
        var version = game_Version.GetVersion();
        Constants.Version = version.First();
        return version != null;
    }
```

Рисунок 3.16 – Клас ControllerMain

Другим буде клас ControllerProfile (рис. 3.17). В нього буде два методи:

– GetPosition: приймає об’єкт SummonerDTO. Цей метод створює екземпляр класу League_V4, викликає метод GetPosition та зберігає об’єкт PositionDTO до класу Constants;

– GetContext: викликає метод GetPosition, створює екземпляр класу ViewModelProfile та передає до конструктора властивості Summoner і Position з класу Constants.

```
public class ControllerProfile
{
    1 reference
    public object GetContext()
    {
        var summoner = Constants.Summoner;
        GetPoSition(summoner);
        return new ViewModelProfile(summoner, Constants.Position);
    }

    1 reference
    private void GetPoSition(SummonerDTO summoner)
    {
        League_V4 league = new League_V4(Constants.Region);
        var position = league.GetPositions(summoner.Id).Where(p=>p.QueueType.Equals("RANKED_SOLO_5x5")).FirstOrDefault();
        Constants.Position = position != null ? position : new PositionDTO();


    }
}
```

Рисунок 3.17 – Клас ControllerProfile

Тепер можна створити клас ViewModelProfile (рис. 3.18), який буде даватиме змогу виводити отримані данні з класів ControllerProfile та ControllerMain до вікна WindowProfile за допомогою прив’язок. Прив’язки є одним із ключових нововведень у WPF.

Рисунок 3.18 – Клас ViewModelProfile

Залишилось дописати логіку до роботи вікна і перша з цілей буде досягнута.

До кнопки Search у вікні MainWindow потрібно додати подію на натискання. Ця подія буде визивати метод класу ControllerMain та відкривати вікно WindowProfile, але тільки в тому випадку, якщо ім’я та регіон будуть вказані та методи контролера повернуть True, в ішому випадку створиться вікно з написом «Not found».

Також на кнопку Exit додається подія для повного виходу з застосунку.

Після того, як вікно WindowProfile відкрите, буде спрацьовувати логіка цього вікна – запускатися конструктор та викликати метод GetContext з класу ControllerProfile, тим самим заповнивши властивість вікна DataContext об’єктом класу ViewModelProfile.

Далі визначаються події до кнопок Close та Details: Close буде переходити на головне вікно, а Details буде відкривати вікно з детальною інформацією по 3 останніх іграх цього гравця.

Тепер можна перейти до розробки вікна, в якому буде відображатися 3 останні гри та вся інформація з вікна WindowProfile. Для цього визначається клас Match_V5, що наслідується від класу Api. У ньому буде нове поле PartOfWorld, тому що ця група методів використовує у рядку запита не регіон, а частину світу. Тож при розробці це має бути враховане.

Методи класу Match_V5 (рис.3.19):

– конструктор: перебирає усі регіони та призначає за регіоном значення властивості PartOfWorld;

– GetURI: перевизначення методу базового класу на те, щоб повертав рядок з правильним посиланням;

– GetURIForMatchData: повертає рядок з видозміненим посиланням;

– GetMatchsId: приймає PUUID гравця. Цей метод буде отримувати JSON файл з API та конвертувати його в список рядків з id минулих ігор;

– GetMatchInfo: приймає id окремої гри та конвертує JSON у MatchDTO.

–

```
5 references
protected override string GetURI(string path)
{
    return "https://" + PartOfWorld + ".api.riotgames.com/lol/" + path + "&api_key=" + Key;
}

1 reference
private string GetURIForMatchData(string path)
{
    return "https://" + PartOfWorld + ".api.riotgames.com/lol/" + path + "?api_key=" + Key;
}

1 reference
public List<string> GetMatchsId(string PUUID)
{
    string path = "match/v5/matches/by-puuid/" + PUUID + "/ids?start=0&count=5";
    var response = GET(GetURI(path));
    string content = response.Content.ReadAsStringAsync().Result;
    if (response.StatusCode == System.Net.HttpStatusCode.OK)
    {
        return JsonConvert.DeserializeObject<List<string>>(content);
    }
    return null;
}
1 reference
public MatchDto GetMatchInfo(string Id)
{
    string path = "match/v5/matches/" + Id;
    var response = GET(GetURIForMatchData(path));
    string content = response.Content.ReadAsStringAsync().Result;
    if (response.StatusCode == System.Net.HttpStatusCode.OK)
    {
        return JsonConvert.DeserializeObject<MatchDto>(content);
    }
    return null;

}
```

Рисунок 3.19 – Методи класу Match_V5

Далі створюється модель MatchDTO для отримання інформацію з JSON. Оскільки JSON, який отримується в Match_V5, багаторівневий – знадобляться ще дві моделі, InfoDTO та ParticipantDto.

Модель MatchDTO буде мати лише одну властивість Info – для зберігання об’єкта моделі InfoDTO.

Модель InfoDTO буде мати три властивості:

– GameMode: режим гри;

– GameType: тип гри;

– Participants: список об’єктів класу Participant, тобто учасників, що приймали участь у грі.

Для отримання даних з матчу модель Participant має декілька властивостей:

– ChampLevel: рівень чемпіона за закінчену гру;

– ChampionName: ім’я чемпіона;

– Kills: кількість вбивств інших чемпіонів за гру;

– Deaths: кількість смертей;

– Assists: кількість допомоги в вбивстві чемпіона;

– ItemN: предмет у N клітинці;

– Summoner1Id: перша здібність гравця;

– Summoner2Id: друга здібність гравця;

– TotalMinionsKilled: кількість вбитих міньйонів;

– NeutralMinionsKilled: кількість вбитих лісних монстрів;

– Puuid: id гравця;

– TeamPosition: роль гравця;

– Win: виграла команда чи ні.

Контролер для класу Match_V5 буде мати два методи:

– GetContext, цей метод створює об’єкт класу ViewModelProfile2 та повертає його;

– GetGameInfo, отримує список ігор завдяки класу Match_V5 та повертає його.

Клас ViewModelProfile2 буде копіювати всі властивості класу ViewModelProfile2 та мати такі ж властивості, як у Participant для можливості виводити ці данні до вікна нашого застосунку.

Для відображання даних з класу ViewModelProfile2 створюється вікно WindowProfile2 (рис 3.20). Воно буде відображати інформацію з вікна WindowProfile та всі данні за 3 останні гри.

На цій сторінці буде лише одна кнопка, яка по натисканню буде закривати це вікно та відкривати вікно WindowProfile. Новий клас для роботи з API Champion_V3 буде давати дані про безплатних чемпіонів, за яких можна грати зараз, оскільки кожні 2 тижні міняється ротація.

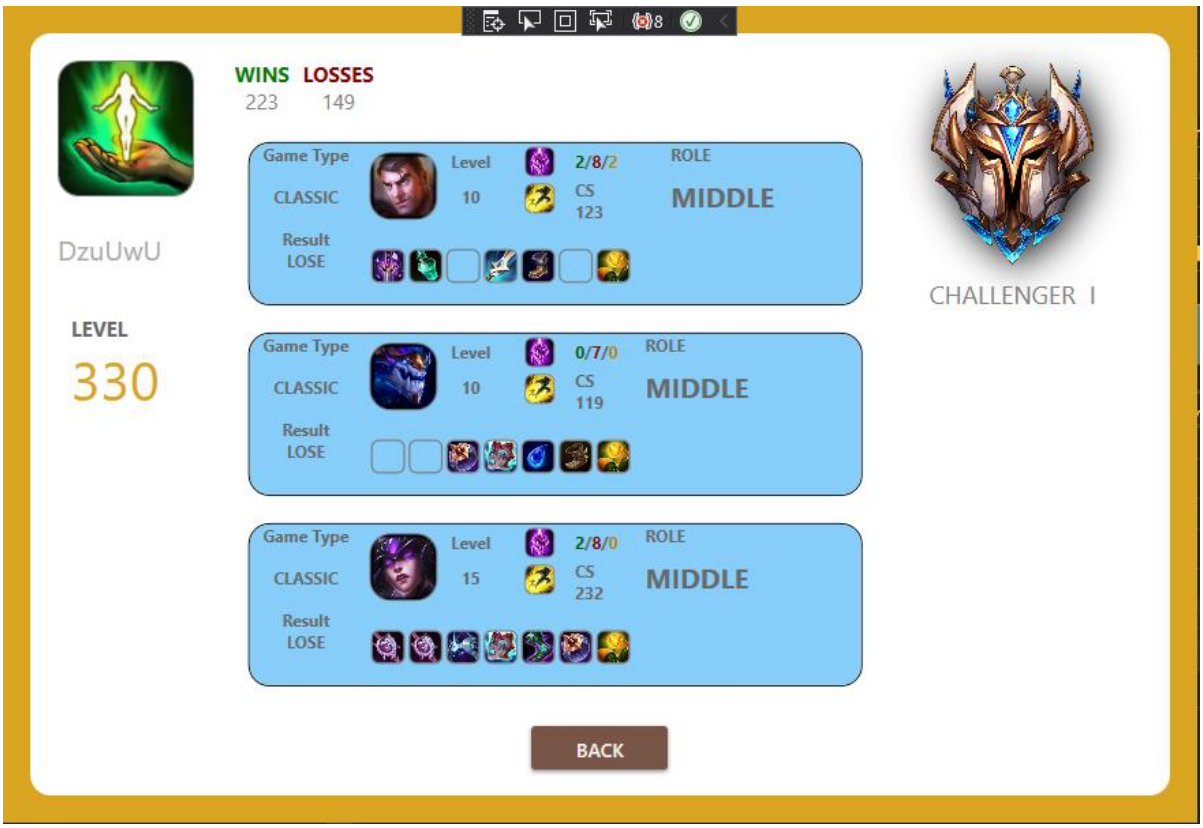


Рисунок 3.20 – Вікно WindowProfile2

В ньому буде два методи:

– конструктор: конструктор базового класу Api;

– GetChampionsRotationIds: цей метод отримує JSON файл та конвертує його в оберт класу ChampionInfo.

ChampionInfo – це модель з єдиною властивістю freeChampionIds. Це список цілих чисел. Він буде зберігати id чемпіонів, що доступні безплатно для гри.

Оскільки немає запиту, що повертав би словник з усіма чемпіонами та їх id, потрібно створити enum у файлі з класом Constants для зручного доступу, а називатиметься він Champs.

Тепер можна переходити до контролерів. По-перше, потрібно дописати один метод до ControllerMain. Цей метод буде викликати метод GetChampionsRotationIds та зберігати його до властивості в Constants.

По-друге, треба створити контролер ControllerRotation, який буде взаємодіяти з ViewModelRotation. Цей контролер буде мати два методи – GetContext і GetChampionNames. GetChampionNames перероблює список id у список імен чемпіонів у безплатній ротації для отримати картинки чемпіонів з API для вікна, де вони будуть відображатися. А GetContext буде створювати і повертати ViewModelRotation.

Залишилось зробити ViewModelRotatin та WindowRotation. ViewModelRotatin буде мати властивості типу string, що зберігають посилання на картинки чемпіонів.

Вікно WindowRotation (рис. 3.21) відображає картинки та кнопку Close, щоб перейти до головного вікна.

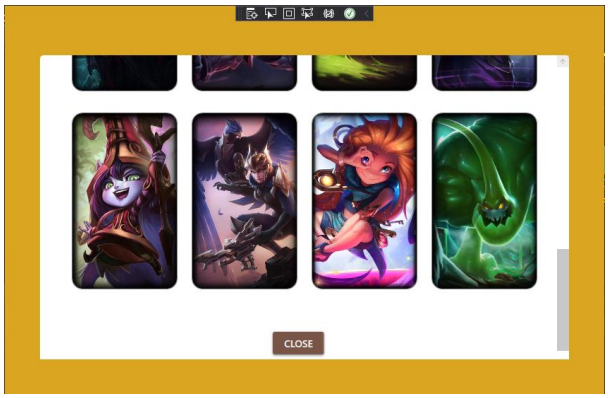


Рисунок 3.21 – Вікно WindowRotation

Отже, залишилось тільки дописати логіку сторінки MainWindow, щоб була можливість перейти на WindowRotation. Для цього потрібно написати подію для кнопки Free Rotation. Перехід буде здійснюватися, тільки якщо буде вибраний регіон.

Клас, який буде взаємодіяти з API – клас Static_Champion_Details, унаслідується з класу Api, має три методи:

– конструктор: він викликає базовий конструктор;

– GetStaticChampioInfo: отримує JSON та вибирає тільки необхідну частину, оскільки дані, що повертаються, мають поле, в якого міняється назва. Після вибору підходящої частини JSON потрібна конвертація її в конструкцію Dictionary<string,Dictionary<string,ChampData>>, тому що в отриманому файлі дуже багато вкладеностей та деякі поля міняють назву в залежності від чемпіона;

– GetURI: перевизначення методу базового класу, яке повертає рядок запиту до статичних даних.

Модель для класу Static_Champion_Details має назву ChampData, у неї чотири властивості:

– Name: для збереження імені чемпіона;

– Lore: для збереження лор чемпіона;

– Allytips: рекомендації для союзних чемпіонів;

– Enemytips: рекомендації для супротивників.

Клас контролер ControllerStaticChampData має два методи:

– GetContext: цей метод буде створювати об’єкт класу ViewModelStaticData для вікна ChampDetailsWindow, щоб можна було передавати з класу дані до вікна;

– GetStaticChampData: створить об’єкт класу Static_Champion_Details та викличе метод GetStaticChampioInfo і отримає об’єкт типу Dictionary<string,Dictionary<string,ChampData>>.

Одже, перед тим як перейти до розробки самого вікна, залишилось зробити ViewModelStaticData. Цей ViewModel буде мати чотири властивості:

– Icon: ця властивість зберігатиме рядок запиту до API, щоб отримати картинку чемпіона;

– Lore: ця властивість буде зберігати лор чемпіона;

– Allytips: ця властивість буде зберігати рекомендації союзникам;

– Enemytips: ця властивість буде зберігати рекомендації супротивникам.

Також ViewModelStaticData має конструктор, у якому присвоюються всі значення до властивостей.

Тепер створюється вікно (рис 3.22) для відображення інформації з ViewModelStaticData. Тут буде випадаючий список, де можна вибирати ім’я чемпіона, щоб подивитися інформацію, картинка з ілюстрацією чемпіона та текст з лору та рекомендацій.

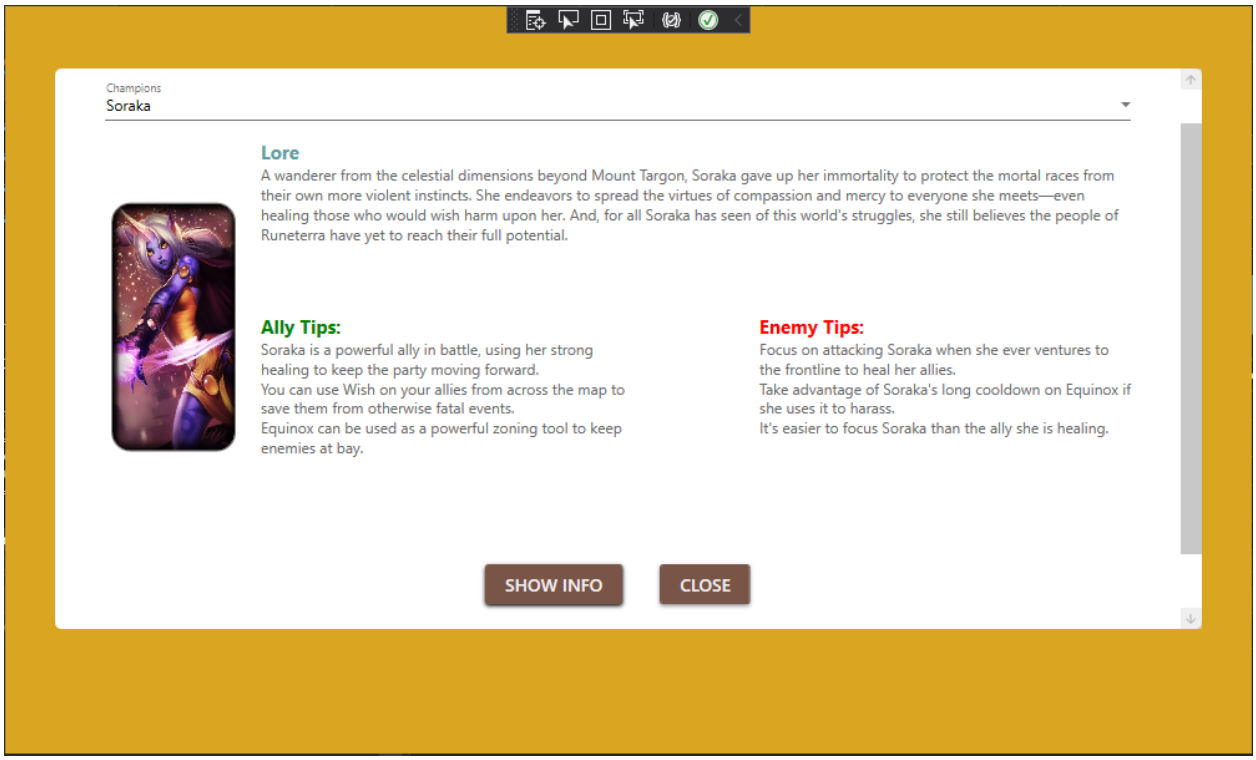


Рисунок 3.22 – Вікно ChampDetailsWindow

Останнє, що треба зробити – це дописати логіку головного вікна, щоб можна було переходити до ChampDetailsWindow та написати логіку вікна ChampDetailsWindow. Для головного вікна потрібно написати подію на

натискання для кнопки ChampionLore, де буде відкриватися вікно ChampDetailsWindow (рис 3.23).

Необхідно дописати конструктор, щоб заповнити випадаючий список нашим enum з чемпіонами, та визначити дві події для кнопки Back та ShowInfo. Back буде переходити до головного вікна при натисканні, а ShowInfo буде створювати об’єкт контролеру ControllerStaticChampData, викликати через об’єкт метод GetContext і призначати результат до DataContext вікна.

```
4 references
public partial class ChampDetailsWindow : Window
{
    ControllerStaticChampData controller;
    string championName;
    1 reference
    public ChampDetailsWindow()
    {
        InitializeComponent();
        foreach (var champ in Enum.GetNames(typeof(Champs)))
        {
            cbChampion.Items.Add(champ);
        }
        controller = new ControllerStaticChampData();
    }


    1 reference
    private void btnClose_Click(object sender, RoutedEventArgs e)
    {
        MainWindow main = new MainWindow();
        this.Close();
        main.ShowDialog();
    }

    1 reference
    private void cbChampion_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        championName = (sender as ComboBox).SelectedValue.ToString();
    }

    1 reference
    private void btnShowInfo_Click(object sender, RoutedEventArgs e)
    {
        Constants.ChampName = championName;
        this.DataContext = controller.GetContext(championName);
    }
}
```

Рисунок 3.23 – Логіка вікна ChampDetailsWindow

ВИСНОВКИ

У рамках кваліфікаційної роботи був розроблений застосунок з інтегрованою API.

Наукова новизна дослідження методів розробки комп'ютерних застосунків з інтегрованими API подягає в виявленні критичної ролі API у сучасній розробці програмного забезпечення. API забезпечують безперешкодний зв'язок між системами, пропонуючи гнучкість і масштабованість для створення складних застосунків на різних платформах і в різних доменах.

Завдяки всебічному аналізу методів інтеграції, таких як REST, SOAP, GraphQL та WebSockets, визначено їхні сильні та слабкі сторони. Простота і масштабованість REST, надійність і безпека SOAP, гнучкість запитів даних GraphQL і ефективність WebSockets у спілкуванні в режимі реального часу надають розробникам ряд інструментів для задоволення конкретних вимог застосунків.

Дослідження також підкреслило важливість архітектурних патернів, таких як MVC та MVVM, для структурування застосунків, щоб забезпечити зручність обслуговування, тестування та простоту інтеграції API. Наприклад, використання WPF з MVVM спрощує прив'язку даних і покращує розподіл завдань, що робить його надійним вибором для десктопних застосунків, які покладаються на API.

Однак дослідження також виявило проблеми в інтеграції API, включаючи управління версіями, забезпечення зворотної сумісності, управління обмеженнями швидкості та усунення помилок від сторонніх сервісів. Ці проблеми підкреслюють потребу в надійних механізмах обробки помилок, ефективному управлінні ресурсами та чіткій документації API для зменшення ризиків та підвищення надійності застосунків.

В результаті дослідження були розроблені практичні рекомендації щодо вибору відповідних API та методів інтеграції на основі вимог проєкту. Ці висновки сприяють глибшому розумінню API-орієнтованої розробки та створюють основу для майбутніх досягнень у цій галузі.

Отже, інтеграція API у комп’ютерні програми є невід’ємним аспектом сучасної програмної інженерії. Вибираючи правильні інструменти, методи та архітектурні патерни, розробники можуть створювати масштабовані, ефективні та зручні застосунки, які відповідають вимогам взаємопов’язаного цифрового світу.

Результати дослідження апробовано у вигляді тез доповіді під час II Міжнародна науково-практична конференція «SCIENTIFIC RESEARCH: MODERN CHALLENGES AND FUTURE PROSPECTS», 23-25.09.2024, Мюнхен, Німеччина [28].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Fielding, R. T., & Taylor, R. N. (2000). Architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine. (pp. 5–25).

2. Richardson, L., Amundsen, M., & Ruby, S. (2020). RESTful Web APIs: Services for a changing world. O'Reilly Media. (pp. 17–27).

3. Руденко Д.О., Бондар В.О. (2020). Огляд можливостей використання стратегій об'єктно-орієнтованого маппінгу для зіставлення сутностей при розробці web додатків. Матеріали III Міжнародної науково-практичної конференції «Priority Directions of Science and Technology Development» Київ, Україна.с.377-380.

4. Козуб Д.П., Руденко Д.О. (2023). Можливості використання API BLENDER та OPENGL при 3D моделюванні. 27-й Міжнародний молодіжний форум ≪Радіоелектроніка і молодь у XXI столітті≫. Зб. матеріалів форуму. Т. 7. Харків: ХНУРЕ. 2023. с.80–81.

5. Котенко О.Є., Руденко Д.О., Танянський О.С.(2024).ОГЛЯД СУЧАСНИХ ТЕХНОЛОГІЙ У МЕДИЦИНІ/Актуальні питання педагогіки вищої медичної освіти : зб. матеріалів Всеукраїн. наук.-практ. конф. з міжнар. участю (м. Харків, 28 трав. 2024 р.) / Харків : ХНМУ, 2024.с.132.

6. Самородов В.К., Руденко Д.О.(2024) Вплив та використання сучасних методів розробки засобів для психологічної допомоги//XIX INTERNATIONAL SCIENTIFIC AND PRACTICAL CONFERENCE «Modern Trends are the Driving Force of Scientific Progress», April 17-19, 2024 Lisbon, Portugal, pp.101-103.

7. Newman, S. (2021). Building Microservices: Designing Fine-Grained Systems (2nd ed.). O'Reilly Media. pp. 18–56.

8. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley. pp. 1-29.

9. Albahari, J., & Albahari, B. (2021). C# 10 in a Nutshell: The Definitive Reference (8th ed.). O'Reilly Media. pp. 1–76.

10. Nathan, A. (2019). Windows Presentation Foundation Unleashed (WPF) (3rd ed.). Pearson Education. pp. 9–191.

11. Fielding, R. T. (2000). Representational state transfer (REST). In Architectural styles and the design of network-based software architectures (pp. 76–85).

12. Richardson, L., Amundsen, M., & Ruby, S. (2020). RESTful Web APIs: Services for a changing world. O'Reilly Media. pp. 28–87.

13. Mitra, N. (2003). SOAP version 1.2 part 0: Primer (2nd ed.). W3C Recommendation. pp. 3–19.

14. Hartig, O., & Pérez, J. (2019). An initial analysis of Facebook's GraphQL language. In Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management. pp. 11.

15. Amareen, S., Dector, O. S., Dado, A., & Bosu, A. (2024). GraphQL Adoption and Challenges: Community-Driven Insights from StackOverflow Discussions. pp. 32.

16. Lubbers, P., Greco, B., & Salim, F. (2012). Pro HTML5 Programming: Building Rich Internet Applications with HTML5, CSS3, and WebSockets. Apress. pp. 137–169.

17. Ogundeyi, K. E., & Yinka-Banjo, C. (2019). WebSocket in real time application. Nigerian Journal of Technology, 38(4), 1010-1020.

18. Nilsen, P. (2023). API design and development: Best practices for creating robust and scalable APIs. DEV Community. URL: https://dev.to/pellenilsen/api-design-and-development-best-practices-for-creating-robust-and-scalable-apis-47mf.

19. Proxify. (2021). MVC vs MVVM Architecture. URL: https://proxify.io/articles/mvc-vs-mvvm-architecture.

20. Daoudi, A., ElBoussaidi, G., Moha, N., & Kpodjedo, S. (2019, April). An exploratory study of MVC-based architectural patterns in android apps. In

Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (pp. 1711-1720).

21. MoldStud. (2023). The role of API integration in software development. URL: https://moldstud.com/articles/p-the-role-of-api-integration-in-software-development.

22. Richardson, L., Amundsen, M., & Ruby, S. (2020). RESTful Web APIs: Services for a changing world. O'Reilly Media. pp. 237–260.

23. Fielding, R. T., & Taylor, R. N. (2000). Representational state transfer (REST). Doctoral dissertation, University of California, Irvine. pp. 107–147.

24. Mathijssen, M., Overeem, M., & Jansen, S. (2020). Identification of practices and capabilities in API management: a systematic literature review. arXiv preprint arXiv:2006.10481. 28 p.

25. . ARAVINDA A KUMAR, Divya TL (2024) Security measures implemented in RESTful API Development, 8 p.

26. MoldStud. (2024). Enhancing API scalability with microservices architecture. URL: https://moldstud.com/articles/p-enhancing-api-scalability-with-microservices-architecture.

27. Daily.dev. (2024). API versioning strategies: Best practices guide. URL: https://daily.dev/blog/api-versioning-strategies-best-practices-guide.

28. Руденко Д. О., Маренич В. В. ДОСЛІДЖЕННЯ МЕТОДІВ РОЗРОБКИ ПРОГРАМНИХ ДОДАТКІВ З ІНТЕГРОВАНИМИ API. II Міжнародна науково-практична конференція «SCIENTIFIC RESEARCH: MODERN CHALLENGES AND FUTURE PROSPECTS», 23-25.09.2024, Мюнхен, Німеччина. с.144–147.