**Міністерство освіти і науки України**

**Луцький національний технічний університет**
(повне найменування закладу вищої освіти)

**Факультет комп’ютерних та інформаційних технологій**
(повне найменування факультету)

**Кафедра комп’ютерної інженерії та кібербезпеки**
(повне найменування кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА
# ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

## БАЗА ДАНИХ ДЛЯ ОБРОБКИ ВЕЛИКОЇ КІЛЬКОСТІ ПОДІЙ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ

## DATABASE FOR PROCESSING A LARGE NUMBER OF EVENTS IN REAL TIME

спеціальність 123 Комп’ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп’ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-42
Лук’янчук Юлія Володимирівна

(підпис)

Керівник:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Кваліфікаційну роботу
допущено до захисту
« 07 » червня 2024 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2024 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та кібербезпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
____________ проф. Н.Чернящук
« 10 » 01 2024 р.

# ЗАВДАННЯ
# НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

*Лук'янчук Юлії Володимирівні*
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *База даних для обробки великої кількості подій в режимі реального часу*

Керівник роботи *к.т.н., доцент Лавренчук Світлана Василівна*

затверджені наказом закладу вищої освіти від «30» грудня 2023 року № 459/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 11.06.2024р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області та різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
*Вступ*
*Аналіз поточного стану систем обробки подій в режимі реального часу. Огляд літератури із досліджуваної проблеми*
*Теоретичні основи вибори технологій для обробки великої кількості подій*
*Розробка системи обробки подій в режимі реального часу*
*Висновки*

5. Перелік графічного (ілюстративного) матеріалу:
*Взаємодія найпопулярніших архітектур API в клієнт-серверній побудові додатків*
*Схема спроектованої бази даних*
*Схема взаємодія API з базою даних*

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| *Аналіз проблеми за темою роботи та постановка завдань дослідження* | *Лавренчук С.В., доцент* | | |
| *Теоретичне дослідження та практична реалізація* | *Лавренчук С.В., доцент* | | |
| *Практична реалізація об'єкта проектування* | *Лавренчук С.В., доцент* | | |
| *Нормоконтроль* | *Багнюк Н.В., доцент* | | |
| *Гарант ОП* | *Лавренчук С.В., доцент* | | |
| *Показник запозичень тексту* | ___*%* | | |
| *Академічна доброчесність* | *Міскевич О.І.,асистент* | | |

7. Дата видачі завдання 10.01.2024 р.

КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | *Розділ 1. Аналіз поточного стану систем обробки подій в режимі реального часу. Огляд літератури* | до 15.02.2024 р. | Виконано |
| 2. | *Розділ 2. Теоретичні основи вибори технологій для обробки великої кількості подій* | до 15.03.2024 р. | Виконано |
| 3. | *Розділ 3.Розробка системи обробки подій в режимі реального часу* | до 04.05.2024 р. | Виконано |
| 4. | *Висновки та пропозиції* | до 07.05.2025 р. | Виконано |
| 5. | *Формування списку використаних джерел* | до 10.05.2024 р. | Виконано |
| 6. | *Формування додатків* | до 15.05.2024 р. | Виконано |
| 7. | *Оформлення ілюстративного матеріалу* | до 20.05.2024 р. | Виконано |
| 8. | *Нормоконтроль* | до 01.06.2024 р. | Виконано |
| 9. | *Інструментальна перевірка на академічний плагіат* | до 04.06.2024 р | Виконано |
| 10. | *Представлення кваліфікаційної роботи бакалавра до захисту* | до 11.06.2024 р. | Виконано |

**Здобувач вищої освіти** ____________ Лук'янчук Ю.В.
(підпис) (прізвище, ініціали)

**Керівник кваліфікаційної роботи** ____________ Лавренчук С.В.
(підпис) (прізвище, ініціали)

## АНОТАЦІЯ


Лук'янчук Ю.В. База даних для обробки великої кількості подій в режимі реального часу. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2024. 57 с.

Кваліфікаційна робота складається з вступу, трьох розділів, висновків, списку використаних джерел, додатків.

Перший розділ присвячено огляду предметної області, тут розглядаються основні поняття про системи обробки подій, здійснено порівняння існуючих технологій та підходів, проаналізовано сучасні тенденції в розробці систем обробки подій та керування даними.

В другому розділі здійснено вибір та обгрунтування засобів розробки. Обрано засоби: C#, MSSQL. Розлянуто поняття технологічного стеку та критеріїв його обрання.

Третій розділ присвячено опису розробленого програмного забезпечення, а саме: спроєктованої бази дани, розробленому інтерфейсу та API для роботи з цією базою даних. Також, у цьому розділі провдено та описано тестування даного програмного забезпечення.

Ключові слова: MSSQL, C#, REST, RESTful API, API, CRUD, ACID, PostgreSQL, Oracle, MySQL, бази даних, реляційні бази даних, Dapper.

# ANNOTATION

Lukianchuk Y.V. Database for processing a large number of events in real time. Manuscript.

Qualifying work of a bachelor of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2024. 57 p.

Qualification work consists of an introduction, three sections, conclusions, a references, two appendices.

The first section is dedicated to the overview of the subject area, here the basic concepts of event processing systems are considered, a comparison of existing technologies and approaches is made, and modern trends in the development of event processing systems and data management are analyzed.

In the second section, the selection and grounding of development tools was carried out. Selected tools: C#, MSSQL. The concept of the technological stack and the criteria for its selection are clarified.

The third section is devoted to the description of the developed software, namely: the designed database, the developed interface and API for working with this database. Also, testing of this software is conducted and described in this section.

Keywords: MSSQL, C#, REST, RESTful API, API, CRUD, ACID, PostgreSQL, Oracle, MySQL, databases, relational databases, Dapper.

# ЗМІСТ

ВСТУП ........ 7

РОЗДІЛ 1 АНАЛІЗ ПОТОЧНОГО СТАНУ СИСТЕМ ОБРОБКИ ПОДІЙ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ ........ 9

1.1 Аналіз популярних систем та підходів до обробки подій ........ 9

1.2 Порівняння існуючих технологій та підходів ........ 11

1.3 Сучасні тенденції у розробці систем обробки подій ........ 17

РОЗДІЛ 2 ТЕОРЕТИЧНІ ОСНОВИ ВИБОРУ ТЕХНОЛОГІЙ ДЛЯ ОБРОБКИ ВЕЛИКОЇ КІЛЬКОСТІ ПОДІЙ ........ 20

2.1 Огляд поняття технологічного стеку та його компонентів ........ 20

2.2 Критерії вибору технологічного стеку для вирішення поточної задачі ........ 22

2.3 Вибір оптимального рішення для вирішення задачі ........ 24

РОЗДІЛ 3 РОЗРОБКА СИСТЕМИ ОБРОБКИ ПОДІЙ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ ........ 26

3.1 Проєктування базової конфігурації для системи зберігання даних ........ 26

3.2 Проєктування інтерфейсів для взаємодії з системою зберігання даних ........ 35

3.3 Тестування працездатності та швидкодії системи ........ 39

ВИСНОВКИ ........ 50

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ........ 52

ДОДАТКИ ........ 55

# ВСТУП

Сьогодні світ генерує безпрецедентний обсяг даних, що надходять з різноманітних джерел, таких як IoT-пристрої, соціальні мережі, сенсорні мережі, фінансові транзакції тощо. Ці дані, які часто називають «великими даними», характеризуються великим обсягом, високою швидкістю та різноманітністю. Такі дані в режимі реального часу стають усе важливішими для багатьох галузей, таких як телекомунікації, фінанси, охорона здоров'я, розумні міста та багато інших.

В контексті такого розмаїття інформації ефективна обробка великої кількості подій в режимі реального часу стає пріоритетною задачею для численних галузей. Багатоаспектна та динамічна природа цього завдання створює виклики для проєктування та впровадження баз даних, спеціально адаптованих для обробки подій у режимі реального часу.

Актуальність обраної теми полягає у створенні ефективної системи обробки та аналізу подій в режимі реального часу, адже зростання обсягів даних, що надходять в реальному часі з різноманітних джерел створюють у цьому велику необхідність. Вимоги до миттєвої реакції та розширення застосувань в різних галузях підкреслюють важливість розвитку баз даних, спроєктованих спеціально для оптимальної обробки великої кількості подій в режимі реального часу.

Мета кваліфікаційної роботи полягає у розробці та дослідженні бази даних, здатної ефективно обробляти великі обсяги даних в режимі реального часу.

Завдання роботи полягають у наступному:

- провести аналіз існуючих рішень для обробки даних в реальному часі;
- розробити модель бази даних, що відповідає потребам обробки великих потоків даних;
- реалізувати прототип бази даних та провести його тестування;
- оцінити ефективність та масштабованість розробленої бази даних.

Об’єктом дослідження є система обробки великої кількості подій в режимі реального часу.

Предметом дослідження є база даних, спеціально розроблена та оптимізована для ефективного збереження, обробки та аналізу великого потоку подій в режимі реального часу. У фокусі дослідження – технології та методи проєктування баз даних, які забезпечують швидкодіючу та масштабовану обробку подій, надходячи з різних джерел, таких як IoT-пристрої, сенсори, соціальні мережі та інші джерела в реальному часі.

Методами дослідження є:

- систематизації та узагальнення – вивчення теоретичних основ та найновіших досягнень у даній сфері;
- аналіз існуючих рішень – вивчення та порівняння існуючих систем баз даних та стрімінгових платформ для обробки подій в реальному часі для виявлення переваг, недоліків та тенденцій розвитку;
- емпіричне дослідження – розробка та реалізація практичних етапів проєктування та створення бази даних, включаючи вибір технологій, оптимізацію та тестування для перевірки ефективності та масштабованості;
- експерименти та випробування – використання реальних або симульованих навантажень для тестування продуктивності та надійності розробленої бази даних в режимі реального часу;
- індукції та дедукції – розробка напрямів удосконалення та оптимізації системи.

Інформаційною базою дослідження були технічні документації та специфікації, публікації, періодичні видання, монографії, навчальні посібники та інші інтернет-ресурси. У вигляді даних для тестування та валідації розробленої бази даних були використані синтетичні дані, що були створені власноруч та відкриті дані, що були взяті з відкритих джерел для додавання реалізму та репрезентації різноманітності інформації.

# РОЗДІЛ 1
# АНАЛІЗ ПОТОЧНОГО СТАНУ СИСТЕМ ОБРОБКИ ПОДІЙ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ

## 1.1 Аналіз популярних систем та підходів до обробки подій

В реальному світі події відбуваються постійно, і деякі з них відбуваються незалежно від наших дій або впливу (наприклад, природні явища, такі як схід чи захід сонця). Інші події ми викликаємо своїми діями або рішеннями. Для їх обробки, зазвичай створюють окремі системи з відповідною задачою. Одними з складових таких систем є: API та бази даних.

API (прикладний програмний інтерфейс, інтерфейс програмування застосунків) – «забезпечує абстракцію для проблеми та визначає, як клієнти повинні взаємодіяти з програмними компонентами, які реалізують рішення цієї проблеми. Самі компоненти зазвичай розповсюджуються як бібліотека програмного забезпечення, що дозволяє використовувати їх у багатьох програмах» [1].

Також, розуміти API можна як «набір протоколів, які дозволяють різним компонентам програмного забезпечення спілкуватися та передавати дані» [2]. Розробники використовують такі інтерфейси для того, щоб створити потужні, безпечні та стійкі програми, що забезпечуватимуть потрбеи користувачів.

Іншою не від’ємною частиною такої системи є база даних. Вона допомагає зберігати та актуалізовувати відповідні дані.

База даних – це «збір даних, організованих відповідно до концептуальної структури, що описує характеристики цих даних і зв’язки між їхніми відповідними об’єктами, підтримуючи одну або більше областей застосування» [3].

Згідно різноманітних характеристик бази даних поділяються на кілька типів (рисунок 1.1) [4].

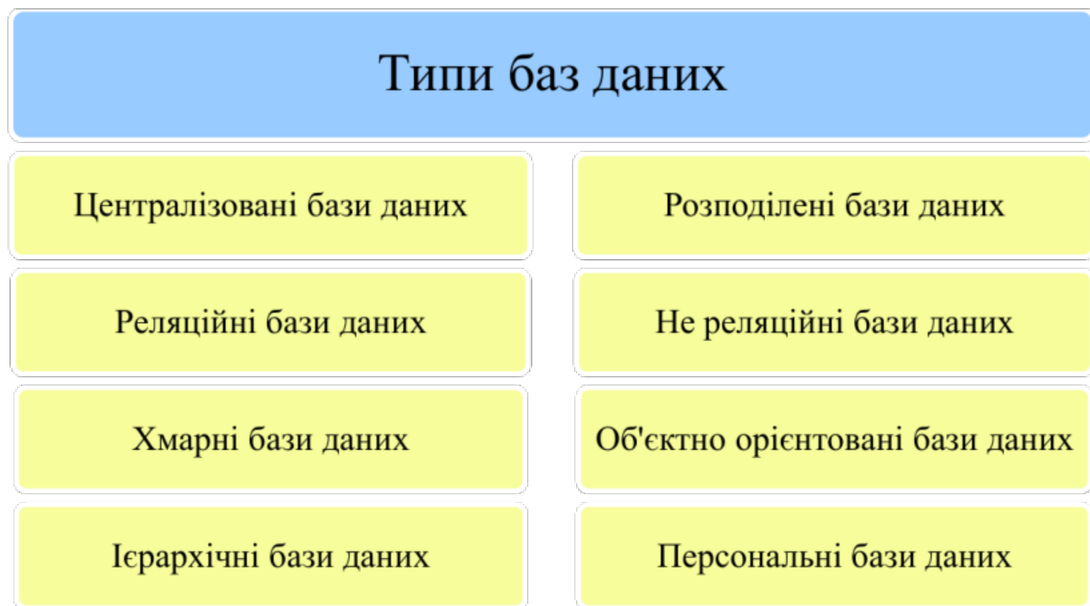



Рисунок 1.1 – Типи баз даних

Під час виконання даної роботи ми найдетальніше розглянемо та працюватимемо з реляційними базами даних.

Реляційна база даних – «сукупність пов'язаної інформації, що зберігається у двовимірних таблицях» [5]. Іншими словами, вони будуються за допомогою зовнішніх ключів – посилань на інші таблиці. Така властивість дає змогу використовувати нормалізацію та денормалізацію при створенні бази даних. Запити в таких базах даних формують за допомогою структорованої мови SQL.

SQL (мова структурованих запитів, англ. «Structured Query Language») – «це стандартна мова запитів, яка використовується для роботи з реляційними базами даних» [6]. Вона може використовуватись для:

- створення баз даних та таблиць;
- читання даних з таблиць;
- додавання даних в таблиці;
- зміна даних в таблиці;
- видалення даних в таблиці;
- створення індексів та зовнішніх ключів;
- видалення баз даних та таблиць.

Серед великого різноманіття баз даних є кілька найбільш поширених (MySQL, PostgreSQL, MongoDB, Microsoft SQL Server, SQLite, Oracle, MariaDB), згідно з щорічними опитуваннями розробників (рисунок 1.2) [7].

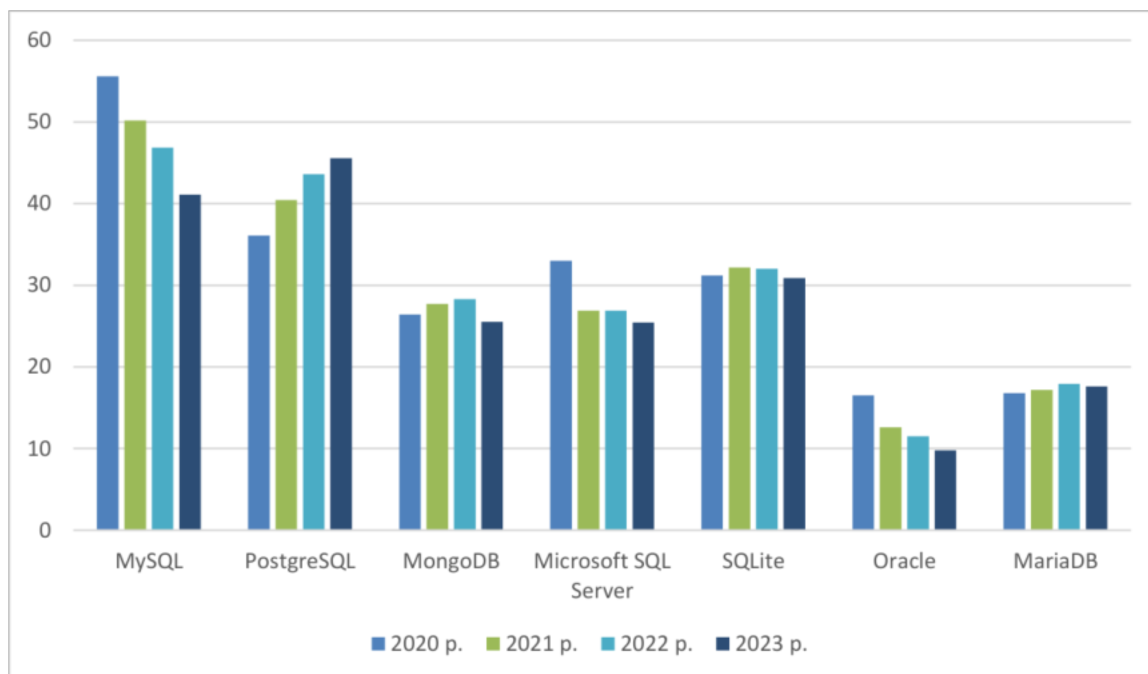



Рисунок 1.2 – Популярні бази даних серед розробників, 2020-2023 рр.

На даному рисунку дуже чітко видно, що деякі бази даних залишаються актуальними протягом багатьох років, тоді як у інших популярність поступово спадає або зростає.

## 1.2 Порівняння існуючих технологій та підходів

Серед найбільш популярних підходів до розробки різноманітних додатків та великого різноманіття архітектур, поняття клієнт-сервер з використанням API є одним з найпопулярніших. Відповідно до цього було розроблено багато типів прикладних програмних інтерфейсів для різних типів задач з використанням різноманітних протоколів.

Найбільш популярними архітектурами є: REST, SOAP, GraphQL, gRPC, WebSocket, and Webhooks [8]. Розглянемо їх детальніше, а також виокремимо переваги та недоліки.

RESTful API – «архітектура інтерфейсу прикладних програм (API), що використовує HTTP-запити для доступу до ресурсів та їхнього використання. Такими HTTP-запитами є GET, PUT, POST і DELETE. Вони дають змогу, відповідно, читати, змінювати, створювати й видаляти ресурси» [9].

Перевагами цієї архітектури є:

- простота у розумінні та використанні;
- використовує існуючі протоколи та стандарти;
- швидкодія;
- можливість обробки великої кількості запитів;
- підримування кешування;
- гнучкість та можливість використовувати різні формати та типи носіїв.

Недоліками цієї архітектури є:

- немає чіткої схеми, що може зробити її непослідовною і незрозумілою;
- не підтримує складні запити чи операції, через, що потрібна велика кількість запитів для отримання достатньої кількості даних;
- погано обробляє помилки чи винятки, адже використовує коди стану HTTP протоколу, які не завжди є чіткими.

Добре підходить для ситуацій коли:

- модель даних стабільна і проста;
- клієнт та сервер є незалежними один від одного;
- швидкість і масштабованість є важливими.

SOAP (Simple Object Access Protocol) – «протокол на основі XML, який використовується для обміну структурованою та типізованою інформацією між системами» [10].

Переваги даної архітектури полягають в наступному:

- має чітку і сувору схему, що забезпечує взаємодію та сумісність;
- підтримує складні запити, наприклад, транзакції або автентифікації;

– добре обробляє помилки та винятки, адже використовує статуси SOAP, які надають детальну інформацію.

Недоліками даної архітектури є:

– складний для використання і розуміння;
– не є масштабованим та продуктивним;
– не є гручким або розширюваним;
– додає складності існуючим протоколам.

Підійде у випадках коли:

– модель даних є складною та динамічною;
– клієнти та сервер тісно пов'язані та залежні;
– безпека та надійність є ключовим аспектом.

Порівняння двох найбільш популярних архітектур SOAP та REST наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння архітектур SOAP та REST

| | SOAP | REST |
|---|---|---|
| Протоколи | HTTP, SMTP, FTP тощо | HTTP |
| Формати повідомлень | XML | JSON, XML, HTML тощо |
| Швидкодія та об'єм повідомлень | Повідомлення більші, повільніше виконання | Повідомлення менші, швидше виконання |
| Масштабованість | Можлива | Можлива, хоча вважається більш масштабованим |
| Кешування даних | Не підтримується | Підтримується |
| Безпека | Підтримує велику кількість стандартів, як WS-Security | Здебільшого лише HTTPS, також OAuth |
| Спосіб передачі даних | Пакетна передача даних | Один запит за раз |
| Інтерфейс | WSDL (Web Services Description Language) | HTTP методи та URL-адреси |
| Тип протоколу | Stateful | Stateless |
| Простота використання | Використання XML і додаткових протоколів | Використання стандартних HTTP методів та URL-адресів |
| Підтримка транзакцій | Є | Немає |
| Стандартизація | WSDL і WS-* (Web Services Extensions) | Немає |
| Контроль помилок | SOAP статуси | HTTP статуси |
| Підтримка для мультимедіа | Краще працює з структурованими даними | Може використовувати різні формати передачі даних |
| Міжплатформенність | Використовує менш поширені формати даних | Використовує більш поширені формати даних |
| Споживання ресурсів | Акцентується на операціях та методах | Зосереджений на ресурсах |

GraphQL – «мова запитів для API та середовище виконання для виконання цих запитів із наявними даними. Надає повний і зрозумілий опис даних у вашому API, дає клієнтам можливість вимагати саме те, що їм потрібно, і нічого більше, полегшує розвиток API з часом і надає потужні інструменти розробника» [11].

Переваги:

- працює з будь якою серверною мовою, адже використовує власну мову визначення схеми;
- має одну кінцеву точку, що робить його ефективнішим;
- строго типізований, що гарантує послідовність і сумісність даних між клієнтом і сервером;
- дозволяє клієнтам отримувати лише ті дані, які їм потрібні.

Недоліки:

- складний і важкий для розуміння, вимагає вивчення нового синтаксису та логіки;
- не підтримує кешування по замовчуванню;
- погано обробляє помилки.

Найкраще використовувати у наступних ситуаціях:

- модель даних є складною та динамічною, оскільки вона може обробляти вкладені та реляційні дані;
- клієнти та сервери не залежні один від одного та самостійно визначають які дані їм потрібні;
- пропускна здатність та продуктивність важливі.

gRPC – «продукт Google, розроблений для стандартизації взаємодії між сервісами й зменшення обсягу трафіку. Це хороша заміна REST під час взаємодії між мікросервісами» [12].

Переваги:

- має чітку структуру, що забезпечує взаємодію та сумісність між сервісами;
- підтримує складні запити та операції, такі як, потокове передавання, шифрування та інші;

– швидкий і ефективний, оскільки використовує бінарний формат і функції HTTP/2 для зменшення затримки та пропускної здатності.

Недоліки:

– важкий і складний для використання та розуміння;

– вимагає створення та компіляції файлів буфера протоколу;

– використовує власні методи та елементи, що несумісні з стандартними веб-інструментами.

Найкраще підійде для:

– складних та динамічних моделей даних, оскільки може обробляти структуровані та неструктуровані дані;

– сервіси залежні один від одного;

– швидкість та ефективність важливі.

WebSocket – це «протокол, описаний у специфікації RFC 6455, і забезпечує спосіб обміну даними між браузером і сервером через постійне з'єднання. Дані можна передавати в обох напрямках у вигляді «пакетів», без розриву з'єднання та додаткових HTTP-запитів» [13].

Плюси:

– швидший і ефективніший, ніж HTTP, оскільки використовує одне з'єднання та не потребує заголовків або файлів cookie для кожного повідомлення;

– може надсилати й отримувати різні типи повідомлень через одне з'єднання;

– може надсилати дані від сервера до клієнта, не чекаючи запиту, що забезпечує зв'язок у реальному часі та на основі подій.

Мінуси:

– не підтримується деякими старими браузерами або проксі-серверами;

– апріорі небезпечно, якшо не використовуєть wss (аналог https);

– не зберігає жодної інформації про з'єднання чи повідомлення з будь якої сторони.

Підходить для:

– швидкого та інтерактивного обміну даними;

– двонаправленого та мультиплексного зв’язку, де обидві сторони можуть надсилати та отримувати кілька повідомлень різних типів одночасно;

– потрібен зв’язок в режимі реального часу та на основі подій, де сервер може надсилати дані клієнту, не чекаючи запиту.

Webhooks – механізм, який дозволяє веб-додатку автоматично повідомляти інший сервер або додаток про події, що відбуваються в першому. Замість того, щоб періодично опитувати сервер для отримання оновлень, веб-додаток може встановити webhook і попросити сервер надіслати повідомлення відразу, як тільки подія відбувається [14].

Переваги:

– простий і легкий у використанні та розумінні;

– використовує методи та формати HTTP;

– використовує існуючі стандарти та протоколи;

– масштабований та продуктивний;

– підтримує асинхронний зв’язок без опитування чи очікування.

Недоліки:

– не має узгодженої схеми, що може призвести до конфліктів та неузгодженості між серверами та клієнтами;

– не підтримує складні запити чи операції;

– надсилає лише односторонні сповіщення;

– погано обробляє помилки, не забезпечує повторних спроб.

Корисний у ситуаціях, коли:

– потрібні сповіщення, керовані подіями, наприклад, GitHub сповіщає нас про нові коміти;

– модель даних є простою і стабільною, обробляє прості типи даних;

– сервер та клієнт не залежні один від одного;

– продуктивність та масштабованість важливі.

У таблиці 1.2 наведена різниця між WebSocket та Webhooks.

Таблиця 1.2 – Різниця між WebSocket та Webhooks

| | Webhooks | WebSocket |
|---|---|---|
| Комунікація | Патерн взаємодії, де сервер повідомляє клієнта про події, які відбуваються, відправляючи HTTP-запити на заздалегідь визначений URL-адрес клієнта | Протокол, який забезпечує двосторонню інтерактивну комунікацію між клієнтом та сервером, дозволяючи обидвом сторонам відправляти та отримувати повідомлення в реальному часі через одне постійне з'єднання |
| Тип комунікації | Комунікація відбувається на основі HTTP-запитів, які сервер відправляє на URL-адресу, зазначену клієнтом | Комунікація відбувається через повноцінне двостороннє з'єднання між клієнтом та сервером, що дозволяє взаємодіяти в реальному часі без необхідності постійно відправляти HTTP-запити |
| Тип використання | Використовуються для повідомлення про асинхронні події | Використовуються для побудови реального часу, інтерактивних додатків |
| Протоколи | Використовують стандартні HTTP-протоколи для взаємодії | Використовують WebSocket-протокол, який побудований на основі TCP і забезпечує бінарне повідомлення між клієнтом та сервером |

У додатку А подано схематичне зображення взаємодії усіх перелічених архітектур в клієнт-серверній побудові додатків.

### 1.3 Сучасні тенденції у розробці систем обробки подій

Системи обробки подій дійсно стали невід'ємною частиною нашого життя. Вони є всюди навколо нас, від онлайн-банкінгу та мобільних оповіщень до систем безпеки. Оскільки, суспільство не стоїть на місці і дуже динамічно розвивається, а роль систем обробки подій у такому технологічному суспільстві зростає, то вони дуже активно та постійно розвиваються. Серед основних тендецій розвитку цих технологій можна назвати:

1. Буде сильніше розвиватись напрямок систем обробки подій в режимі реального часу, адже усі системи в яких уже використовується обробка подій потребують миттєваої реакції на них.

2. Масштабованість. У зв'язку з постійним збільшенням обсягів даних, такі системи теж повинні модифіковуватись: розподілені архітектури, контейнеризація та обчислення в хмарі вже для цього використовуються.

3. Безвідмовність. Буде підвищуватись увага до зменшення затримок у таких системах та їх швидкодії, адже ці застосунки вимагають мінімізацію реакції на винекнення події.

4. Розвиток технологій машинного навчання та штучного інтелекту. Цільком можливо, що у найближчі роки у системи обробки подій поступово будуть впроваджувати різних «асистентів», адже їм можна доручити різноманітні задачі: від важких регуляних обчислень, щоб машина тренувалась і до генерації різноманітних звітів та графіків. Не виключено, що саме штучний інтелект може забрати рутинну роботу як в роботі з великими даними, так і в роботі з великою кількістю подій.

5. Стандартизація. Для полегшення взаємодій та інтеграці між різноманітними системами обробки подій тв зручності як розробниківт, так і клієнтів.

6. Гнучкість та адаптативність. Системи обробки подій повинні бути гнучкими та адаптивними до змін у вихідних даних та вимогах додатків.

7. Віртуальна та доповнена реальності. Ці технології відкривають необмежений потенціал для користувачів та їх зручності через можливість взаємодії з реальним та не реальним світом.

8. Безсерверна архітектура. Поступово системи обробки подій будуть позбавлятись серверів, адже використання хмарних сховищ покращить як взаємодію так і швидкодію таких систем.

9. Поширення на сфери, у яких системи обробки подій ще не є популярними. У будь якій сфері життя людини використання такої системи є дуже зручним та ефективним, тому можливо, що ці системи будуть все дужче входити у найрізноманітніші сфери нашого життя.

10. Аналітика в режимі реального часу. Ці системи все частіше використовуються для аналітики даних в режимі реального часу, що дозволяє їм отримувати інформацію про події в міру їх виникнення.

11. Використання потокових даних для більшої швидкодії.

12. Покращення рівня захисту та безпеки клієнтів.

13. Відкритий та загальнодоступний вихідний код додатків.

14. Технології обробки природної мови.

Наявність такої великої кількості тенденцій показує, що системи обробки подій стають все більш гнучкими та потужнішими. Вони все частіше використовуються в ширшому колі програм і відіграють важливішу роль у сучасних програмних системах. Їх використання зростає в міру того, як системи стають більш динамічними, складними та масштабованими [15].

Важливо зазначити, що системи обробки подій стають невід'ємною складовою для багатьох галузей, зокрема фінансів, медицини, IoT, моніторингу та аналітики.

Тенденції включають в себе спрямованість на реальний час, масштабованість, низьку латентність, гнучкість, адаптивність, підтримку складних аналітичних операцій, інтеграцію з мікросервісною архітектурою, підвищену увагу до безпеки та конфіденційності, а також до стандартизації та інтероперабельності [16].

За допомогою цих систем компанії можуть ефективно виявляти та реагувати на події в реальному часі, аналізувати потокові дані, виявляти аномалії, прогнозувати та приймати обґрунтовані управлінські рішення. Із зростанням обсягу даних та складності вимог, системи обробки подій лишаються важливим інструментом для досягнення бізнесових цілей у сучасному світі.

# РОЗДІЛ 2
# ТЕОРЕТИЧНІ ОСНОВИ ВИБОРУ ТЕХНОЛОГІЙ ДЛЯ ОБРОБКИ ВЕЛИКОЇ КІЛЬКОСТІ ПОДІЙ

## 2.1 Огляд поняття технологічного стеку та його компонентів

Перш ніж звертати уваги на критерії при виборі технолгій для розробки базової частини проєкту, потрібно розібратись, що таке технологічний стек та що він в себе включає.

Технологічний стек – «це набір програмних компонентів, інструментів і технологій, які використовуються для розробки та виконання додатків» [17]. Його можна порівняти з «екосистемою», де кожен елемент виконує певну функцію і взаємодіє з іншими компонентами для досягнення цілі – створення повністю функціонального продукту.

Він включає в себе як зовнішні, так і внутрішні технології, необхідні для розробки продукту. Значення стеку може змінюватись в залежності від вимог та цілей проєкту, уподобань і досвіду команди розробників, а також загальноприйнятих технологій у компанії.

Основні компоненти типового стека технологій:

- фронтенд технології – комбінація засобів, що забезпечує роботу веб-сайту чи програми за рахунок прямого контекту та взаємодії з користувачем;
- бекенд технології – комбінація засобів, що забезпечує роботу веб-сайту або програми, залишаючись невидимою для користувача.

Детальніше розглянемо кожен з цих компонентів. Фронтенд технології зазвичай включають в себе:

- HTML: використовується для створення структури сторінки.
- CSS: використовується для оформлення та організації вигляду веб-сторінок та керування їх макетом.
- JavaScript: використовується для створення інтерактивних та динамічних веб сторінок. Часто використовується разом з різноманітними

бібліотеками та фреймворками, такими як React, Angular або Vue.js, для створення інтерфейсів користувача.

У той час як бекенд технології можна поділити на кілька великих груп, які необхідні як для зручності користувача, так і для зручності розробників. Перша велика група, на яку варто звертати увагу – це серверні мови програмування. Вони використовуються для обробки серверної логіки, обробки запитів і спілкування з базами даних. Напопулярніші варіанти включають в себе:

- Node.js – це середовище виконання JavaScript, яке часто використовується для розробки масштабованих серверних програм.
- Python – відомий своєю простотою та читабельністю, і використовується з такими фреймворками, як Django або Flask.
- Ruby – часто використовується з фреймворком Ruby on Rails.
- Java – універсальна та широко використовувана мова, яка часто поєднується з такими фреймворками, як Spring.
- C# – зазвичай використовується для створення додатків та веб-служб на базі Windows, часто з інфраструктурою ASP.NET.
- PHP – відомий своїм широким використанням у веб-розробці, особливо з фреймворками на основі PHP, такими як Laravel.

Наступною ключовою групою є бази даних. Для розробки веб-сайтів та програм зазвичай використовуються або реляційні бази даних (MySQL, PostgreSQL, Microsoft SQL Server, Oracle), або не реляційні (MongoDB, Cassandra, Redis). Перші використовують мову структурованих запитів для обробки даних, тоді як другі підходять для обробки різних типів даних і вимог до масштабованості.

Ще одними важливими групами є фреймворки та бібліотеки для створення серверних програм, таких як Express.js (для Node.js), Django (для Python), Ruby on Rails, Spring (для Java) і ASP.NET (для C#) та веб-сервери – програмне забезпечення, відповідальне за обслуговування веб-вмісту та обробку запитів HTTP, наприклад Apache, Nginx або Microsoft Internet Information Services (IIS) [18].

Також, не менш важливими складовими технологічного стеку є інструменти контролю версій (наприклад, Git), безперервна інтеграція та безперервне розгортання (CI/CD), а також платформи контейнеризації, такі як Docker та інструменти оркестрування, наприклад Kubernetes.

Не менш важилвими є різні бібліотеки та пакети, які розширюють функціональність і ефективність коду та хмарні сервіси, для розміщення, масштабування та керування додатками. Також, багато розробників під час вибору технологічного стеку і спираючись на пореби проєкту можуть використовувати рішення для кешування, брокери повідомлень і API сторонніх розробників.

## 2.2 Критерії вибору технологічного стеку для вирішення поточної задачі

Вибір технологічного стеку для реалізації будь-якої задачі є ключовим етапом у процесі розробки програмного забезпечення. При цьому, враховуючи особливості конкретної задачі, слід ретельно вивчити різні аспекти та можливості різних технологій для забезпечення оптимальності та ефективності реалізації. Необхідно проаналізувати ключові аспекти, які допоможуть у виборі оптимального технологічного стеку для успішного вирішення поставленої задачі.

Отже, до основних вимог під час вибору технологічного стеку, для врішення задачі з побудови бази даних та API для оброки великої кількості подій, належать:

1. Технологічний стек повинен забезпечувати швидку та ефективну обробку подій в реальному часі, щоб забезпечити низьку затримку і високу швидкість відповіді системи.

2. Обраний технологічний стек повинен бути здатним масштабуватися горизонтально та вертикально, щоб забезпечити підтримку великої кількості одночасних подій і збільшення обсягу обробки даних.

3. Вибрані технології повинні бути надійними та стабільними, щоб гарантувати безперебійну роботу системи в умовах високого навантаження.

4. Технології повинні підтримувати різноманітні джерела даних та формати, так щоб було можливо ефективно інтегрувати їх у систему обробки подій.

5. Важливо враховувати наявність розвинутої екосистеми та підтримки спільноти для обраного технологічного стеку, що дозволить швидко вирішувати проблеми та використовувати нові можливості.

6. Технології мають бути зручними у використанні та мають забезпечувати швидкий розвиток програмного забезпечення.

7. Технології мають підтримувати зручну та ефективну інтеграцію зі зовнішніми API для обміну даними та взаємодії з іншими системами.

8. Важливо звертати увагу на вартість використання обраних технологій, включаючи вартість ліцензій та можливі додаткові витрати на підтримку та розширення.

9. Важливо вибрати технології, які відповідають сучасним стандартам та рекомендаціям у галузі обробки подій.

10. Важливо мати доступ до потрібних інструментів розробки, документації та підтримки для ефективної розробки та підтримки системи обробки подій.

Оцінка та вибір технологій та архітектурних рішень відповідно до зазначених критеріїв визначає стратегічну основу для активного та стійкого розвитку проєкту. Вибір оптимальних технологій та архітектурних підходів відіграє ключову роль у забезпеченні постійного удосконалення, масштабованості та адаптації проєкту до змінних потреб та умов. Ретельний аналіз цих пунктів визначає надійну основу для подальшого успішного розвитку проєкту, забезпечуючи його конкурентоспроможність та стабільність в майбутньому [19].

## 2.3 Вибір оптимального рішення для вирішення задачі

На основі зазначених критеріїв та перечислених технології, для вирішення цієї задачі були обрані:

- Microsoft SQL Server – це реляційна система керування базами даних (СКБД), розроблена компанією Microsoft. Вона відома своєю надійністю, продуктивністю та розширюваністю і широко використовується для зберігання та управління даними в різних типах додатків, від невеликих веб-сайтів до великих корпоративних систем. Microsoft SQL Server підтримує широкий спектр функцій, включаючи транзакційність, кластеризацію, реплікацію, аналітику, засоби бізнес-аналітики та інтеграцію з іншими продуктами Microsoft, такими як Visual Studio та Azure. Він також має різні редакції, включаючи безкоштовну версію Express, яка надає можливості для розробки та використання баз даних без великих витрат. Продовжує залишатися однією з провідних СКБД на ринку, завдяки своїм широким можливостям, підтримці та розвитку.
- .NET 8 – надає розширені можливості для розробки високопродуктивних та масштабованих додатків. Ця версія фреймворка пропонує покращену підтримку реляційних баз даних, таких як SQL Server, та надає зручні інструменти для роботи з ними, такі як Entity Framework Core. Крім того, .NET 8 має вбудовану підтримку для розробки REST API з використанням ASP.NET Core, що дозволяє швидко створювати та розгортати веб-служби для обміну даними між клієнтами та сервером за стандартом REST.
- Dapper – це легковагова та швидка бібліотека доступу до даних в середовищі .NET, яка надає зручний та ефективний спосіб взаємодії з реляційними базами даних за допомогою мови SQL. Вона відома своєю високою швидкістю роботи, простотою використання та легкою інтеграцією з різними типами баз даних та іншими фреймворками .NET. Dapper дозволяє безпечно виконувати параметризовані запити та автоматично мапує результати до об’єктів .NET, що робить процес взаємодії з даними швидким, зручним та ефективним [20].

– REST API (Representational State Transfer Application Programming Interface) – це архітектурний стиль для створення веб-служб, які забезпечують спосіб взаємодії між різними компонентами програмного забезпечення за допомогою HTTP-протоколу. Основні принципи REST включають використання стандартних HTTP-методів (GET, POST, PUT, DELETE) для взаємодії з ресурсами, безстандартність (клієнт не зберігає стан на сервері), кешування, однорівневу систему (простота та складність), а також універсальні ресурсні ідентифікатори (URI). REST API є популярним та потужним засобом для побудови розподілених систем, оскільки дозволяє легко взаємодіяти з веб-службами та обмінюватися даними між ними з використанням стандартних веб-протоколів.

В загальному, стек технологій невеликий, проте досить надійний та зручний у використанні. В цілому, обрані технології чудово підійдуть для реалізації даного завдання, адже вони є зручними у використанні, забезпечують швидку та ефективну обробку подій в реальному часі, є надійними та стабільними, підтримують різноманітні джерела даних та формати, мають підтримку спільноти, підтримують зручну та ефективну інтеграцію зі зовнішніми API для обміну даними та взаємодії з іншими системами, здебільшого є безкоштовними та відповідають сучасним стандартам.

# РОЗДІЛ 3
# РОЗРОБКА СИСТЕМИ ОБРОБКИ ПОДІЙ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ

## 3.1 Проєктування базової конфігурації для системи зберігання даних

Перш ніж займатись проєктуванням бази даних, необхідно визначити систему керування даними. Для розробки даного проєкту зупинимось на реляційних базах даних, адже вони зручніше зберігають дані, а тож надають велику кільксть інструментів для роботи з ними.

При розробці даного проєкту вибір був серед п’яти популярних систем (згідно опитувань описаних у розділі 1): MSSQL, PostgreSQL, Oracle, MySQL, MariaDB. Для того, щоб обрати найбільш відповідну проведемо їх порівняння (таблиця 3.1) [21].

Таблиця 3.1 – Порівняння архітектур систем керування базами даних

| | Тип бази даних | Ліцензування | Масштабованість | Підтримувані типи даних |
|---|---|---|---|---|
| MySQL | SQL | GNU | Вертикальна | Структуровані, напівструктуровані |
| MariaDB | SQL | GNU | Вертикальна | Структуровані, напівструктуровані |
| MSSQL | T-SQL | Комерційна | Вертикальна | Структуровані, напівструктуровані, не структуровані |
| PostgreSQL | SQL, зв’язки між об’єктами | Open-source | Вертикальна | Структуровані, напівструктуровані, не структуровані |
| Oracle | SQL, інші моделі даних | Комерційна | Вертикальна та горизонтальна | Структуровані, напівструктуровані, не структуровані |

Здебільшого, ці бази даних досить схожі між собою за технічними властивостями. Також, варто зазначити, що в порівнні з іншими популярним на ринку ці є досить важкими для вивчення та розуміння, але досить популярними через їхню надійність.

Не зважаючи на те, що ці бази даних досить популярні серед розробників, вони всеодно мають якісь недоліки. У таблиці 3.2 наведені переваги та недоліки кожної з систем керування, а також можливі випадки застосування.

Таблиця 3.2 – Переваги та недоліки систем керування базами даних

| | Переваги | Недоліки | Можливе застосування |
|---|---|---|---|
| MySQL | Безкоштовна інсталяція, простий синтакис, можливість працювати в хмарі | Складність масштабування, часткова open-source підтримка, не повна відповідність стандартам | Малі веб-застосунки, OLAP/OLTP системи, IoT застосунки |
| MariaDB | Шифрування, великий функціонал, висока ефективність | Невелика спільнота, різниця у версіях з MySQL | |
| MSSQL | Наявність великої кількості версій, наскрізне рішення для бізнес-даних, повна документація та підтримка спільноти, хмарні бази даних | Ціна, не зрозумілі умови ліцензування, складний процес налаштування та оптимізації | Масштабні корпоративні програми, легка інтегрованість в системи корпорації Microsoft |
| PostgreSQL | Легка масштабованість, підтримка користувальницьких типів даних, легко інтегровані інструменти сторонніх розробників, підтримка спільнотою | Невідповідна документація, нестача інструментів звітності та аудиту | Аналіз даних, інструмент для автоматизації баз даних, фінансові установи |
| Oracle | Інновація для щоденної рутини, сильна технічна підтримка та хороша документація, великий об’єм | Ціна, важка для вивчення, витрачає багато ресурсів | Публічний та державний сектор, фінансові установи, масштабні корпоративні програми |

Як було зазначено у попередньому розділі, для розробки даного проєкту була обрана MSSQL, адже вона гарно інтегрована з уже обраною мовою програмування, а також підходить для обробки великих об’ємів даних.

Під час постановки завдання для розробки даного проєкту було виокремлено окремі, основні сутності: користувачі, девайси та сенсори. На основі цих об’єктів і буде побудована наша база даних. Схема бази даних представлена у додатку Б.

На поданій схемі видно, що для зберігання даних про користувачів було створено 3 таблиці, про девайси – 9, а про сенсори – 6.

Таблиці «users» та «users_roles» відповідають за збереження даних користувачів: логін, пароль та дозволи на роботу з даними про девайси та сенсори. Таблиця «users_devices» показує зв’язок між користувачами та девайсами.

Таблиця «devices» містить інформацію про девайс: назву, серійний номер, опис, тип, наявність сенсорів, кількість сенсорів та інформацію про те чи є даний девайс активним. Таблиці «device_groups» та «locations» несуть більш інформативну мету: вони об’єднують девайси у певні групи, які можуть вказувати на положення цих девайсів в приміщенні. Таблиця «device_value» зберігає дані, передані девайсом, наприклад, якщо це холодильник то температуру в середині цього холодильника, а таблиця «device_data» зберігає інформацію про поточний стан цього девайсу: його заряд (якщо таке є), дату останнього заряджання, поточну версію прогрманого забезпечення і статус чи є з девайсом зараз зв’язок. Таблиця «configurations» зберігає дані про поточну конфігурацію девайсу, а таблиця «device_updates» зберігає історію оновлення даних про цей девайс.

Таблиця «sensors» зберігає загальну інформацію про сенсор: назву, те до якого девайсу даний сенсор належить, тип сенсору, його серійний номер та те чи він активний. Таблиця «sensor_types» зберігає узагальнену інформацію про те, який тип має даний сенсор та який тип даних він може передавати. Таблиця «sensor_value» зберігає дані передані сенсором, а таблиця «sensor_data» зберігає інформацію про поточний стан цього сенсору. Таблиця «sensor_updates» зберігає історію оновлення даних про цей сенсор.

На базі цієї схеми був написаний скрипт, що створить базу даних та відповідні таблиці.

На рисунку 3.1 представлено зразок стоврення таблиці мовою SQL: перелік колонок з відповідними типами даних та інформацією про те чи вони можуть мати в записі NULL, посилання на колонку, яка буде в кожному записі

унікальною (Primary key) та посилання на колонки, що мають містити дані з інших таблиць (Foreign key).

```
CREATE TABLE [dbo].[device]
(
    [id]            uniqueidentifier NOT NULL DEFAULT NEWSEQUENTIALID(),
    [name]          NVARCHAR(255)    NULL,
    [serial_number] NVARCHAR(255)    NOT NULL,
    [description]   NVARCHAR(255)    NULL,
    [group_id]      uniqueidentifier NOT NULL,
    [enabled]       BIT              NOT NULL DEFAULT 1,
    [has_sensors]   BIT              NOT NULL DEFAULT 1,
    [sensors_count] INT              NOT NULL,
    [type]          INT              NOT NULL,
    CONSTRAINT "PK_device" PRIMARY KEY ([id]),
    CONSTRAINT "FK_device_group_id" FOREIGN KEY ([group_id]) REFERENCES [dbo].[device_groups] ([id])
);
```

Рисунок 3.1 – Зразок створення таблиці «device» мовою SQL

За такою схемою були створені усі таблиці, вказані у схемі, представленій у додатку А.

Наступним кроком виконання завдання було написання відповідних запитів у базу даних для їх створення, оновлення та видалення.

Серед обов’язкових таблиць, необхідних до заповнення можна назвати таблиці «users», «users_roles», «devices» та «sensors», адже вони зберігають дані про користувачів, їхні права, девайсти та сенсори. У той же час, таблиці «device_value», «sensor_value», «device_data», «sensor_data», «device_updates» та «sensor_updates» будуть постіійно оновлюватись, так як вони повинні зберігати дані про поточний або останній переданий пристроєм чи сенсором статус.

Для того щоб виконувати певні запити чи на рівні бази даних, чи на рівні коду будемо використовувати збережені процедури (stored procedures).

Збережені процедури (stored procedures) – це підпрограми або функції, які зберігаються в базі даних та можуть бути викликані та виконані в будь-який момент. Вони використовуються для виконання складних операцій з даними, що вимагають багатошарової логіки обробки, а також для виконання транзакцій.

Збережені процедури можуть пришвидшувати роботу з базою даних забезпечуючи переваги в плануванні операцій та безпеки [22].

Здебільшого на один вид об'єктів (таблиць) в базі даних пишеться, щонайменше 3 процедури (InsertOrUpdate, Get, DeleteById). У даному проєкті здебільшого будемо використовувати наступні види процедур:

– insert – для того, щоб додати дані в таблицю (рисунок 3.2);

```
CREATE PROCEDURE dbo.InsertDevice
    @name NVARCHAR(255),
    @serial_number NVARCHAR(255),
    @description NVARCHAR(255),
    @group_id UNIQUEIDENTIFIER,
    @enabled BIT = 1,
    @has_sensors BIT = 1,
    @sensors_count INT,
    @type INT
AS
BEGIN
    SET NOCOUNT ON;

    INSERT INTO dbo.device (name, serial_number, description, group_id, enabled, has_sensors, sensors_count, type)
    VALUES (@name, @serial_number, @description, @group_id, @enabled, @has_sensors, @sensors_count, @type);
END
GO
```

Рисунок 3.2 – Зразок процедури «dbo.InsertDevice»

– delete – для того, щоб видалити запис (або кілька записів) за певною умовою (рисунок 3.3);

```
CREATE PROCEDURE dbo.DeleteDevice
@id UNIQUEIDENTIFIER
AS
BEGIN
    SET NOCOUNT ON;

    DELETE FROM dbo.device
    WHERE id = @id;
END
GO
```

Рисунок 3.3 – Зразок процедури «dbo.DeleteDevice»

– update – для того, щоб повністю чбо частково оновити дані в таблиці або певному записі (рисунок 3.4);

```sql
CREATE PROCEDURE dbo.UpdateDevice
    @id UNIQUEIDENTIFIER,
    @name NVARCHAR(255),
    @serial_number NVARCHAR(255),
    @description NVARCHAR(255),
    @group_id UNIQUEIDENTIFIER,
    @enabled BIT,
    @has_sensors BIT,
    @sensors_count INT,
    @type INT
AS
BEGIN
    SET NOCOUNT ON;

    UPDATE dbo.device
    SET name = @name,
        serial_number = @serial_number,
        description = @description,
        group_id = @group_id,
        enabled = @enabled,
        has_sensors = @has_sensors,
        sensors_count = @sensors_count,
        type = @type
    WHERE id = @id;
END
GO
```

Рисунок 3.4 – Зразок процедури «dbo.UpdateDevice»

– getById – для того, щоб отримати один конкретний запис з таблиці по унікальному ідентифікатору, id, який є не може повторюватись (рисунок 3.5);

```sql
CREATE PROCEDURE dbo.GetDeviceById
@id UNIQUEIDENTIFIER
AS
BEGIN
    SET NOCOUNT ON;

    SELECT id, name, serial_number, description, group_id, enabled, has_sensors, sensors_count, type
    FROM dbo.device
    WHERE id = @id;
END
GO
```

Рис. 3.5 – Зразок процедури «dbo.GetDeviceById»

– get – для того, щоб отримати всі записи в таблиці, по критерію або без нього (рисунок 3.6).

```
CREATE PROCEDURE dbo.GetAllDevices
AS
BEGIN
    SET NOCOUNT ON;

    SELECT id, name, serial_number, description, group_id, enabled, has_sensors, sensors_count, type
    FROM dbo.device;
END
GO
```

Рисунок 3.6 – Зразок процедури «dbo.GetAllDevices»

Варто зазначити, що параметри в збережених процедурах пишуться одразу біля назви та позначаються символом «@». Також, на початку файлу ми бачимо команду «Create procedure» – це означає, що база даних повинна створити екземпляр такої процедури, якщо ж ми захочемо її змінити потрібно використати команду «Alter procedure», а щоб видилити – «Drop procedure».

Варто зазначити, що досить часто в таких збережених процедурах можуть виконуватись і складніші задачі, як, наприклад, додавання даних не лише в основну табличку, але і пов’язану з нею, чи велекий обрахунок даних, чи складна логіка по фільтрації запиту, чи навіть виклик кастомних функцій, написаних виключно під конкретну задачу.

Саме такий перелік збережених процедур задовільняє практично всі потреби розробників та користувачів, адже надає можливість користувачам маніпулювати даними. Така система взаємодії з даними називається CRUD.

CRUD – це скорочення від Create (Створення), Read (Читання), Update (Оновлення) і Delete (Видалення). Це основні операції, які можна виконати з даними в базі даних або будь-якому іншому джерелі даних. Вони є ключовими для багатьох систем управління базами даних (СУБД) та додатків, що працюють з даними [23]:

1. Create (створення) дозволяє створити новий запис або об’єкт у базі даних. Це може бути новий користувач, елемент списку, запис у таблиці і т.д.

2. Read (читання) дозволяє отримати інформацію або дані з бази даних без змінювання їх. Це може бути запит на вибірку всіх записів, фільтровану вибірку, отримання конкретного запису за ідентифікатором і т.д.

3. Update (оновлення) дозволяє змінити існуючі дані або записи в базі даних. Це може бути зміна значень полів у записі, оновлення інформації про користувача, оновлення даних у таблиці і т.д.

4. Delete (видалення) дозволяє видалити записи або об'єкти з бази даних. Це може бути видалення користувача, видалення елементів списку, видалення записів у таблиці і т.д.

Ці операції утворюють основу для будь-якої системи, що працює з даними, і дозволяють взаємодіяти з даними, забезпечуючи можливість створення, читання, оновлення та видалення.

Варто зазначити, що в поєднанні з основними операціями CRUD є кілька важливих концепцій і практик, які пов'язані з їх використанням:

1. Кожна операція CRUD повинна бути атомарною, тобто вона повинна виконуватися повністю або не виконуватися взагалі. Це означає, що якщо операція не може бути виконана повністю (наприклад, через помилку), то жодної зміни не повинно бути внесено.

2. Кожна операція CRUD повинна дотримуватися системи прав доступу. Наприклад, користувачі можуть мати різні права на створення, читання, оновлення та видалення даних, і ці права повинні враховуватися при виконанні операцій.

3. Деякі операції CRUD можуть вимагати групування у транзакції, особливо якщо вони пов'язані між собою. Транзакції забезпечують консистентність даних шляхом групування операцій, які повинні бути виконані атомарно.

4. Журналювання використовується для реєстрації змін в даних, здійснених через операції CRUD. Це допомагає відстежувати історію змін даних і відновлювати їх у випадку помилок або втрати.

5. Перед виконанням операцій CRUD часто виконується валідація даних. Це допомагає забезпечити, що дані, надіслані для створення, оновлення або видалення, відповідають певним критеріям або обмеженням.

Одними з найбільших переваг цього підходу є:

– простота, адже CRUD – це простий і зрозумілий спосіб взаємодії з даними. Він надає зрозумілі та стандартні операції для створення, читання, оновлення та видалення даних;

– CRUD застосовується у багатьох сферах, від розробки веб-додатків до управління базами даних, що дозволяє розробникам легко розуміти та використовувати його;

– ми можемо розширювати функціональність CRUD для відповідності специфічним потребам нашого додатку або домену;

– CRUD дозволяє розробникам швидко реалізовувати базові операції з даними, що підвищує продуктивність розробки.

Хоча навіть тут є свої недоліки:

– CRUD може бути обмеженим для складних додатків або систем, які потребують специфічного оброблення даних;

– в деяких випадках CRUD може призвести до зайвого читання або запису даних, що може негативно впливати на продуктивність або витрати ресурсів;

– загальний підхід CRUD може призвести до неправильного використання або неправильної обробки даних, що може вплинути на цілісність даних або безпеку;

– якщо CRUD-операції не виконуються в межах однієї транзакції, це може призвести до конфліктів або невідповідностей в даних, особливо в умовах паралельного виконання.

Хоча не зважаючи на недоліки, цей підхід є потужним і універсальним засобом для роботи з даними, проте для досягнення найкращих результатів його слід розглядати як базовий інструмент і доповнювати іншими методами та паттернами в залежності від потреб конкретного додатку. Найчастіше, патерном, який використовується в розробці разом з CRUD є REST.

Також, при розробці збережених процедур використовувася принцип ACID (Atomicity, Consistency, Isolation, Durability) [24]:

– Atomicity (Атомарність) – транзакція виконується повністю або не виконується взагалі. Якщо одна частина транзакції не вдається, всі зміни, зроблені в рамках цієї транзакції, відкатуються до початкового стану.

– Consistency (Консистентність) – транзакція переводить базу даних з одного допустимого стану в інший, не порушуючи жодних правил або обмежень. Після завершення транзакції всі дані мають бути валідними відповідно до визначених правил.

– Isolation (Ізоляція) – виконання транзакцій паралельно не повинно впливати на їхній результат. Проміжні стани транзакції невидимі для інших транзакцій, поки транзакція не завершиться.

– Durability (Довговічність) – після успішного завершення транзакції її результати зберігаються постійно, навіть у разі збою системи. Зміни записуються в постійне сховище.

Цей принцип роботи з базами даних дозволяє їм залишатись стабільними, гарно та впевнено працювати, а також попереджає будь-які втрати даних чи їхню не відповідність.

## 3.2 Проєктування інтерфейсів для взаємодії з системою зберігання даних

Для того, щоб клієнтські рішення могли спілкуватись з нашою базою даних, як додатки, так і пристрої, необхідно розробити для них інтерфейс для такої взаємодії. За базову архітектуру інтерфейсу візьмемо REST (RESTful) API. При розробці даного інтерфейсу для кожного об’єкту в нас буде окремий контролер, який буде взаємодіяти з даними через базові CRUD операції.

Але всередині буде складніша схема взаємодії. Коли ми робитимемо запит на один з ендпоїнтів, насправді, ми будемо отримувати звичайний json об’єкт, який ми розпарсимо у відповідний тип даних, далі ми будемо створювати з’єднання з базою даних та використовувати ORM бібліотеку для того, щоб перетворити звичайний C# код на SQL запит. Варто наголосити, що в даній

архітектурі використовується взаємодія через збережені процедури, тож сам код має трішки простіший вигляд. На рисунку 3.7 приведена схема взаємодії логіки під час виконня запиту з API.

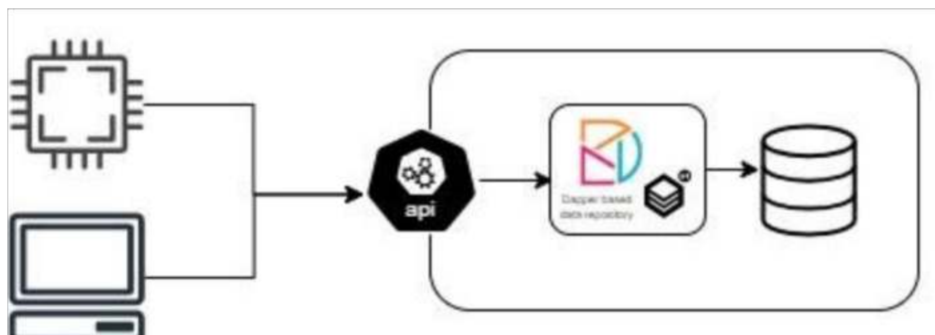



Рисунок 3.7 – Схема взаємодія API з базою даних

Отже, першим кроком для того щоб забезпечити спілкування з базою даних буде створення базового репозиторію для того, щоб можна було викликати збережені процедури прямо з C# коду. Важливо врахувати, що для кожного типу об’єкту має бути написаний окремий репозиторій. Для того щоб написати такий репозиторій ми будемо використовувати ORM бібліотеку Dapper.

Dapper є відкритою об’єктно-реляційною бібліотекою для .NET та .NET Core додатків, яка значно спрощує роботу з базами даних. Завдяки своїй високій продуктивності та легкості використання, Dapper дозволяє розробникам швидко та ефективно взаємодіяти з базами даних, виконуючи прямі SQL запити, конвертуючи результати в об’єкти, а також викликаючи збережені процедури. Це особливо корисно для тих, хто цінує простоту та швидкість розробки, але при цьому хоче зберегти контроль над SQL кодом. Однією з основних переваг Dapper є його продуктивність. У порівнянні з іншими ORM бібліотеками, такими як Entity Framework, Dapper забезпечує швидший доступ до даних, оскільки мінімізує накладні витрати на обробку запитів. Це робить його ідеальним вибором для додатків, де висока продуктивність є критично важливою. Крім того, Dapper підтримує широкий спектр функціональностей, які полегшують роботу з базою даних. Він дозволяє виконувати складні SQL запити, які можуть включати в себе приєднання таблиць, агрегацію даних та використання

підзапитів. Також Dapper дозволяє працювати зі збереженими процедурами, що забезпечує додаткову гнучкість та безпеку під час виконання критично важливих операцій [20]. На рисунку 3.8 показано зразок написання методу, для виконання збереженої процедури яка додаватиме запис про девайс в базу даних.

```
34        public async Task<int> InsertAsync(Device device)
35        {
36            await using var connection = new SqlConnection(_connectionString);
37            var parameters :{Name,SerialNumber,...} = new
38            {
39                device.Name,
40                device.SerialNumber,
41                device.Description,
42                device.GroupId,
43                device.Enabled,
44                device.HasSensors,
45                device.SensorsCount,
46                device.Type
47            };
48            return await connection.ExecuteAsync( sql: "dbo.InsertDevice", parameters,
49                commandType: System.Data.CommandType.StoredProcedure); // Task<int>
50        }
```

Рисунок 3.8 – Метод для додавання запису про девайс в базу даних

Якщо коротко, то:

– у рядку 36 створюється з’єднання з базою даних;

– у рядках 37-47 створюється об’єкт анонімного типу даних, який передаватиме параметри у збережену процедуру;

– у рядках 48-49 викликається метод виконання збережної процедури для додавання нового запису. Як параметри в цей метод передаються назва збережної процедури, список її параметрів та вказівка, що це має виконуватись як збережена процедура.

Цей метод повертає тип int, який позначає кількість змінених рядків. За даним прикладом були написані аналогічні методи для виконання CRUD операцій з усіма перелічинеми раніше типами об’єктів.

Наступним кроком є створення ендпоінтів, через які клієнт зможе передавати дані серверу, щоб той відповідно міг зберегти, оновити чи видалити

їх з місця збереження або ж просто видати певну агрегацію раніше збережених даних за певною умовою.

HTTP endpoint (ендпоїнт) – «це кінцева точка (адреса) веб-сервісу чи додатку, яку можна викликати (запитати) за допомогою HTTP-протоколу. В інших словах, HTTP endpoint – це URL або URI (Uniform Resource Identifier), яке представляє конкретний ресурс або функцію, доступну через HTTP» [25].

Для написання цього прикладного програмного інтерфейсу використовуватимемо технологію Minimal API, яка є доступною для розробки засобами мови програмування C# та платформи .NET.

Насправді, Minimal API це ті самі звичайні HTTP API, просто вони побудовані таким чином, щоб їх можна було створити з мінімальними залежностями. Вони є чудовим рішенням для мікросервісної архітектури чи просто додатків, які хочуть включати в себе мінімальний набір файлів [26].

Найбільшим викликом при виконанні даного завдання було не збільшити розмір основного файлу (Program.cs) в кільканадцять разів, адже типів об'єктів для яких були написані дата сервіси вийшло багато. Тому, не зважаючи на постійне використання Minimal API саме в головному файлі програми, було прийнято рішення, все погрупувати та розділити відповідні ендпоїнти по різних статичних класах та файлах в залежності від типу об'єкту. На рисунку 3.9 наведено приклад HTTP POST ендпоїнту для того, щоб додати нові дані.

```
22        app.MapPost("/api/devices", async (DeviceRepository repository, Device device) =>
23        {
24            try
25            {
26                var result = await repository.InsertAsync(device);
27                return result > 0 ? Results.Created() : Results.BadRequest();
28            }
29            catch (Exception ex)
30            {
31                return Results.BadRequest(ex.Message);
32            }
33        });
```

Рисунок 3.9 – HTTP POST ендпоїнт для додавання нових даних

У рядку 22 відбувається створення даного ендпоїнту: для програми зазначається, що це має бути тип запиту POST, передається зразок URL за якою має відбуватись обмін даними, а потім створюється анонімний лямбда вираз, який і виконує всю основну логіку по опрацюванню даних. У рядках 24 та 29 можна побачити звичайний обробник помилок (try catch) для того, щоб наше API не припиняло роботу при неочікуваних помилках, а просто повертало статус 400 та інформацію про те, яка саме помилка сталась. У рядку 26 відбувається виклик методу, зображеного на рисунку 3.8 для додавання новго запису про девайс в базу даних. У рядку 27 використовується тернарний оператор через перевірку змінених рядків в базі даних: якшо викликаний нами метод повернув значення більше 0, то ми повертаємо клієнту статус 201 повідомляючи його про те, що цей запис був успішно створений, у випадку, якщо даний запис не вдалось зберегти, то ми повертаємо в такому випадку статус 400. В перспективі, в цей метод потрібно додати валідацію даних, перед тим як пробувати зберегти їх в базу даних, адже тоді буде менша ймовірність того, що з невідомої причини ми не можемо зберегти їх і якщо таке станеться, то клієнту буде простіше пояснити, що саме не так з його даними просто повернувши статус код 400 та опис помилки валідації.

Аналогічно до цього методу, були написані і інші типи запитів для конкретного типу даних «девайси» та інших раніше перечислених. Важливо зазначити, що на звичайних методах GET не було прописано обробників помилок, адже вони ніяким чином дані не змінюють, тому там в них сенсу ніякого немає.

### 3.3 Тестування працездатності та швидкодії системи

Для того, щоб почати працювати з даною системою потрібно створити першу роль для користувачів та, відповідного, самого користувача. Для цього достатньо буде просто запустити проєкт та скористатись Swagger UI [27], яка за

замовчуванням додається до веб проєктів на платформі .NET. На рисунку 3.10 зображено приклад звичайного POST запиту через Swagger UI.

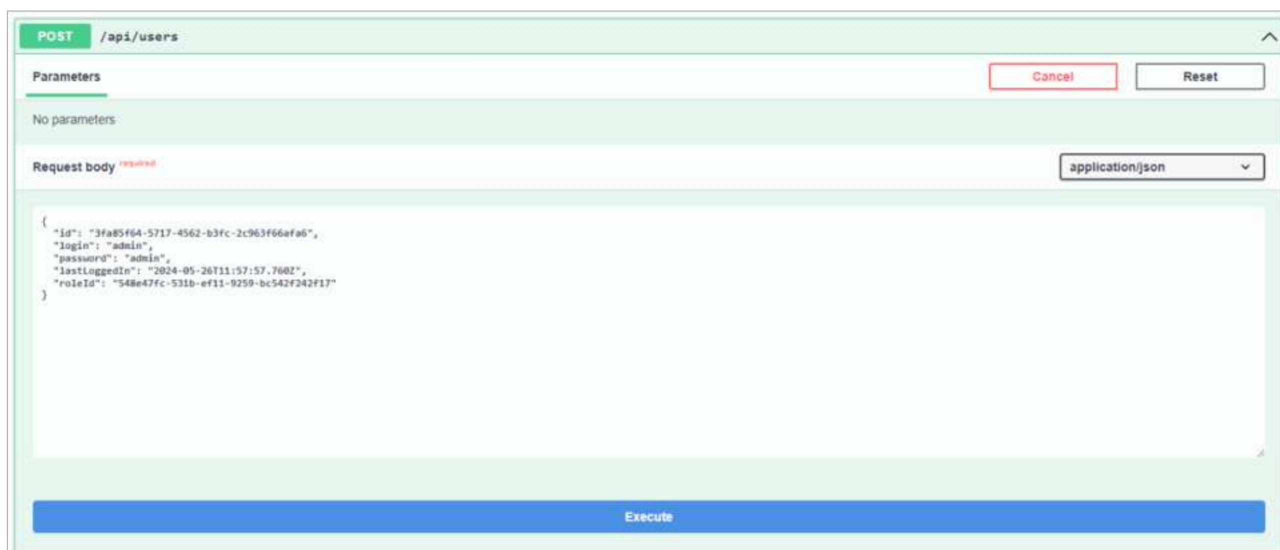



Рисунок 3.10 – HTTP POST запит для додавання користувача

Після того як ми успішно виконали запит, варто переконатись чи наш користувач був дійсно збережений в базу даних. Для цього можна використати запит GET через той самий Swagger UI. На рисунку 3.11 зображено результат виконання GET запиту через Swagger UI.

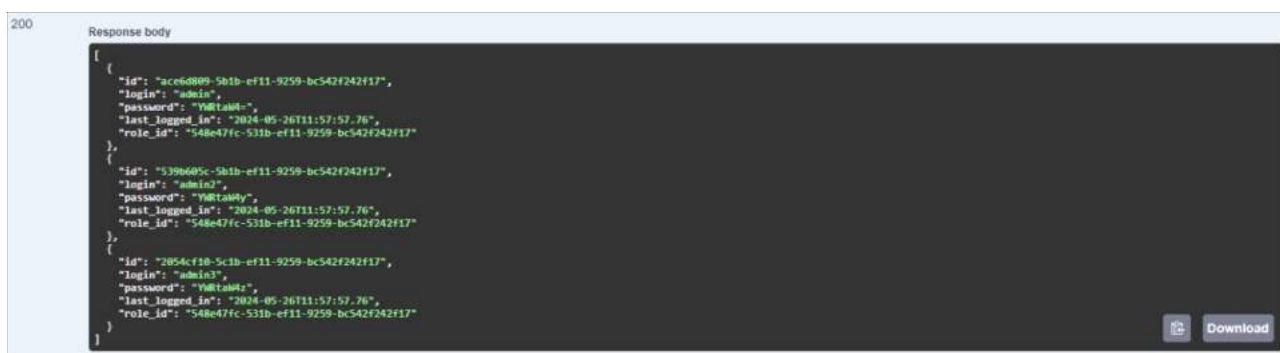



Рисунок 3.11 – Результат виконання HTTP GET запиту для отримання усіх доданих користувачів

Оскільки, наш користувач та роль були успішно додані то можна приступити до додавання девайсів та їхніх сенсорів. Для цього ми також можемо

використати Swagger UI, адже процес буде схожим до того, що відбувався під час створення користувача.

Оскільки девайс ми успішно додали, то для того, щоб додати його сенсори ми створимо невеликий додаток за допомогою якого ми і створимо нові записи в базі даних.

За задумом, це простий, не великий консольний додаток, який просто створює зв’язок з локально піднятим API та просто надсилає запити, очікуючи відповідь. Для того, щоб ми могли надіслати запит створимо невеликий клас, який міститиме у собі перелік методів, що створюють запит на той чи інший ендпоїнт на нашому API. На рисунку 3.12 зображено методи, які використовувались під час створення запитів на API по створенню нових сенсорів.

```
22      public async Task<List<Sensor>> GetAllSensorsAsync()
23      {
24          var response = await _httpClient.GetAsync( requestUri: "/api/sensors");
25          response.EnsureSuccessStatusCode();
26          return await response.Content.ReadFromJsonAsync<List<Sensor>>();
27      }
28
        1 usage
29      public async Task<bool> CreateSensorAsync(Sensor sensor)
30      {
31          var response = await _httpClient.PostAsJsonAsync( requestUri: "/api/sensors", sensor);
32          return response.IsSuccessStatusCode;
33      }
34
```

Рисунок 3.12 – Методи по створенню сенсора та отримання всіх існуючих сенсорів

У рядку 24 створюється базовий GET запит на API через вбудовану бібліотеку C# для отримання усіх існуючих сенсорів. У рядку 25 ми переконуємось, що запит був успішно виконаний та отримав будь який з 200-х статус кодів. У рядку 26 ми повертаємо список сенсорів через розпаршування json об’єкту, який ми отримали у відповідь.

У рядку 31 створюється базовий POST запит на API через вбудовану бібліотеку C# для створення новго сенсору. У рядку 32 ми повертаємо булеве значення про те чи був успішним наш запит.

Дані методи є максимально простими, що полегшує тестування, хоча є не вірним архітектурним рішенням, адже у випадку не очікуваної помилки, наш додаток просто зламається.

Після того як ми створили базовий клас для взаємодії з API ми можемо створити клас, що буде певною надбудовою та буде виконувати всю основну лоігку по створенню нових об'єктів та опрацюванню відповіді від сервера.

В такому класі, у випадку зі створення кількох сенсорів, ми просто створюємо цикл, який буде генерувати дані на основні частково прописаних змінних та відправляти їх на API. На рисунку 3.13 представлено метод, який створював та відправляв на API нові сенсори.

```
20        private static async Task CreateSensor(SensorClient client)
21        {
22            for(var i = 0; i < 6; i++)
23            {
24                var stopwatch = Stopwatch.StartNew();
25                var sensor = new Sensor()
26                {
27                    deviceId = Guid.Parse("228789e0-911b-ef11-9259-bc542f242f17"),
28                    enabled = true,
29                    name = $"sensor_{i}",
30                    serial_number = $"SN_000{i}",
31                    type_id = Guid.Parse("f3c3647a-961b-ef11-9259-bc542f242f17")
32                };
33
34                await client.CreateSensorAsync(sensor);
35                stopwatch.Stop();
36                Console.WriteLine($"Passing sensor {sensor.name} to API and getting response: {stopwatch.ElapsedMilliseconds} ms");
37            }
38        }
39
```

Рисунок 3.13 – Методи для створення об'єкта сенсора

У рядку 22-37 вкионується цикл, який містить в собі наступні операції:

– у рядку 24 ми створюємо новий об'єкт для того, щоб слідкувати за часом виконання запиту;

– у рядках 25 – 32 ми створюємо новий об'єкт сенсора;

– у рядку 34 ми відправляємо запит на API через метод представлений на рис. 3.12;

– у рядку 35 ми зупиняємо початий у рядку 24 таймер виконання запиту;

– у рядку 36 ми виводимо час виконання у консоль.

На рисунку 3.14 представлений час виконання запитів для створення сенсорів на API.

```
Started inserting sensors
Passing sensor sensor_0 to API and getting response: 172 ms
Passing sensor sensor_1 to API and getting response: 5 ms
Passing sensor sensor_2 to API and getting response: 3 ms
Passing sensor sensor_3 to API and getting response: 3 ms
Passing sensor sensor_4 to API and getting response: 2 ms
Passing sensor sensor_5 to API and getting response: 3 ms
Total time inserting all sensors to db: 198 ms
```

Рисунок 3.14 – Час виконання запитів для створення сенсорів

У всіх випадках виконання даних запитів час виконання є допустимим та зручним для будь якого клієнта, а різницю між часом виконання першого та другого запиту можна пояснити тим, що перший запит це була перша фактична спроба з'єднатись з API, тому у наступних запитах крок простої перевірки з'єднання було пропущено.

На рисунку 3.15 представлено час додавання нових записів у базу даних, через створені раніше збережені процедури та використання бібліотеки Dapper.

```
Inserting sensor: 252 ms
Inserting sensor: 7 ms
Inserting sensor: 1 ms
Inserting sensor: 1 ms
Inserting sensor: 1 ms
Inserting sensor: 1 ms
```

Рисунок 3.15 – Час виконання запитів для створення сенсорів

Як і у попередньому випадку, час виконання запитів є допустимим, а різницю між часом виконання першого та другого запиту можна пояснити через

перевірку з’єднання з базою даних під час першого запиту і, оскільки, з’єднання було успішно встановлене, у наступних запитах ця перевірка не виконувалась.

Наступною важливою перевіркою створеної раніше логіки буде паралельне надислання запитів з даними девайсів (тобто його показниками) та сенсорів цього девайсу. Для цього ми точно так само створимо метод, який в циклі буде надсилати близько 100 запитів на API разом з часом виконання. Те саме для спостереження зробимо і на стороні серверного рішення, щоб бачити приблизний час виконання при додаванні нових записів у базу даних. Під час написання програмного коду для тестування API були використані процеси розпаралелювання потоків, щоб можна було виконати більше запитів.

На рисунку 3.16 пердаставлено метод для тестування додавання нових записів з даними з девайсу.

```
24      private static void InsertDeviceData(DeviceDataClient client)
25      {
26          Parallel.For( fromInclusive: 0,  toExclusive: 100,  body: (i int) =>
27          {
28              var stopwatch = Stopwatch.StartNew();
29              var deviceData = new DeviceData()
30              {
31                  device_id = Guid.Parse("228789e0-911b-ef11-9259-bc542f242f17"),
32                  charge_level = 100 - i,
33                  connection_status = 1,
34                  last_charged = DateTime.Now.AddDays(-2),
35                  last_connected = DateTime.Now,
36                  last_updated = DateTime.Now,
37                  permanent_source = true,
38                  version = "1.0.0",
39                  name =$"#{i}"
40              };
41
42              var result bool = Task.Run( function: () => client.CreateDeviceDataAsync(deviceData)).GetAwaiter().GetResult();
43              if (result) _totalInsertedData++;
44              stopwatch.Stop();
45              Console.WriteLine(
46                  $"Passing device data ({deviceData.name}) to API and getting response: {stopwatch.ElapsedMilliseconds} ms");
47          });
48      }
```

Рисунок 3.16 – Створення нових записів з даними з девайсу

У рядках 26-47 виконується розпаралелізована версія циклу for, яка є доступною в мові програмування C#. Всередні цього циклу, ми створюємо лямбда функцію, яка і виконує всю основну логіку по створенню новго об’єкта

та відправці його на апі, а також логуванню часу, який було витрачено на це. Частина результатів тестування представлена на рисунках 3.17 та 3.18.

```
Passing device data (#57) to API and getting response: 114 ms
Passing device data (#11) to API and getting response: 114 ms
Passing device data (#86) to API and getting response: 204 ms
Passing device data (#80) to API and getting response: 11 ms
Passing device data (#22) to API and getting response: 11 ms
Passing device data (#67) to API and getting response: 132 ms
Passing device data (#23) to API and getting response: 6 ms
Passing device data (#52) to API and getting response: 121 ms
Passing device data (#89) to API and getting response: 138 ms
Passing device data (#33) to API and getting response: 88 ms
Passing device data (#44) to API and getting response: 123 ms
Passing device data (#3) to API and getting response: 142 ms
Passing device data (#69) to API and getting response: 126 ms
Passing device data (#42) to API and getting response: 11 ms
Passing device data (#91) to API and getting response: 13 ms
Passing device data (#55) to API and getting response: 144 ms
```

Рисунок 3.17 – Результати виконання тестування методу для створення записів з девайсів (клієнтський додаток)

```
Inserting device data (#57) to db : 2 ms
Inserting device data (#18) to db : 4 ms
Inserting device data (#11) to db : 5 ms
Inserting device data (#86) to db : 5 ms
Inserting device data (#80) to db : 2 ms
Inserting device data (#22) to db : 6 ms
Inserting device data (#67) to db : 2 ms
Inserting device data (#23) to db : 2 ms
Inserting device data (#52) to db : 9 ms
Inserting device data (#89) to db : 6 ms
Inserting device data (#33) to db : 3 ms
Inserting device data (#44) to db : 8 ms
```

Рисунок 3.18 – Результати виконання тестування методу для створення записів з девайсів (серверний додаток)

Як видно з результатів тестування, навіть у такому випадку створена інфрастуктура пройшла випробування і справляється в досить оптимальний час, який є зручним для клієнтів.

Дослідимо що станеться якшо ми збільшимо кількість запитів з 100 до 600. Це ми можемо перевірити при додаванні записів з сенсорів. Код клієнтської сторони представлений на рисунках 3.19 та 3.20.

```
11  public static void SensorDataOperations(SensorDataClient client, SensorClient sensorClient)
12  {
13      var stopwatch = Stopwatch.StartNew();
14      Console.WriteLine("Started inserting sensor data");
15      var sensors = Task.Run(sensorClient.GetAllSensorsAsync).GetAwaiter().GetResult();
16
17      Parallel.ForEach(sensors, sensor => InsertSensorData(client,sensor.id));
18
19      stopwatch.Stop();
20      Console.WriteLine($"Total time inserting all sensor data: {stopwatch.ElapsedMilliseconds} ms");
21      Console.WriteLine($"Total inserted sensor data: {_totalInsertedData}");
22      Console.WriteLine(Environment.NewLine);
23  }
```

Рисунок 3.19 – Метод для створення нових записів з сенсорів

```
25  private static void InsertSensorData(SensorDataClient client, Guid sensorId)
26  {
27      Parallel.For(0, 100, (i) =>
28      {
29          var stopwatch = Stopwatch.StartNew();
30          var sensorData = new SensorData()
31          {
32              sensor_id = sensorId,
33              charge_level = 100 - i,
34              connection_status = 1,
35              last_charged = DateTime.Now.AddDays(-2),
36              last_connected = DateTime.Now,
37              last_updated = DateTime.Now,
38              permanent_source = true,
39              version = "1.0.0",
40              name = $"#{i}"
41          };
42
43          var result = Task.Run(() => client.CreateSensorDataAsync(sensorData)).GetAwaiter().GetResult();
44          if (result) _totalInsertedData++;
45          stopwatch.Stop();
46          Console.WriteLine(
47              $"Passing sensor data ({sensorData.name}) to API and getting response: {stopwatch.ElapsedMilliseconds} ms");
48      });
49  }
```

Рисунок 3.20 – Метод для створення нових записів з сенсорів та відправки запитів на API

Логіка роботи даних методів мало чим відрізняється від представленої при роботі з даними з девайсів, але варто відзначити, що в даному випадку у зовнішньому методі, зазначеному на рис. 3.19, у рядку 17 використовується Parallel.ForEach для того, щоб ми точно змогли підставити id відповідного

сенсора і щоб у нас не було плутанини в даних. Це було зроблено для комфорту, а також для збільшення навантаження на API та базу даних.

Результати тестування частково представлені на рисунках 3.21 та 3.22.

```
Passing sensor data (#48) to API and getting response: 6566 ms
Passing sensor data (#49) to API and getting response: 2053 ms
Passing sensor data (#27) to API and getting response: 1027 ms
Passing sensor data (#25) to API and getting response: 1027 ms
Passing sensor data (#13) to API and getting response: 5601 ms
Passing sensor data (#37) to API and getting response: 4074 ms
Passing sensor data (#24) to API and getting response: 7594 ms
Passing sensor data (#13) to API and getting response: 6622 ms
Passing sensor data (#50) to API and getting response: 1021 ms
Passing sensor data (#37) to API and getting response: 5097 ms
Passing sensor data (#84) to API and getting response: 5610 ms
Passing sensor data (#14) to API and getting response: 5613 ms
Passing sensor data (#60) to API and getting response: 7581 ms
```

Рисунок 3.21 – Результати виконання тестування методу для створення записів з сенсорів (клієнтський додаток)

```
Inserting sensor data (#15) to db : 6 ms
Inserting sensor data (#26) to db : 10 ms
Inserting sensor data (#48) to db : 9 ms
Inserting sensor data (#60) to db : 9 ms
Inserting sensor data (#22) to db : 12 ms
Inserting sensor data (#72) to db : 10 ms
Inserting sensor data (#97) to db : 12 ms
Inserting sensor data (#25) to db : 22 ms
Inserting sensor data (#4) to db : 23 ms
Inserting sensor data (#74) to db : 9 ms
```

Рисунок 3.18 – Результати виконання тестування методу для створення записів з сенсорів (серверний додаток)

Як видно з результатів даного тестування, наша система все ще прекрасно справилась, хоча помітно, що час відповіді від сервера зріс. Це можна пояснити кількома факторами:

– на сервер відправлялась більша кількість запитів (600 проти 100 у минулому прикладі) через що навантаження на машину зросло, оскільки серверу

потрібно розбиратись хто «прийшов», а хто вже має «йти» і паралельно ще очікувати відповідь від бази даних чи запис був успішно доданий;

– клієнт і сервер запускаються на одній машині, через що логіка ропаралелювання клієнта впливає на сервер і навпаки.

Але не заважючи на ці фактори система працює досить добре, тож для невеличких рішень як, наприклад, для невеликого домогосподарства вона може підійти, адже не є складною і запити у ній прості.

Проте, є кілька шляхів для покращення спроєктованої системи, які варто було б врахувати в майбутньому:

– подумати про безпеку: на даному етапі, ця система немає жодної базової безпеки навіть від простих sql-ін'єкцій, тому в перспективі варто було б переглянути написане рішення та покращити його захист. Також, оскільки в проєкті є ролі користувачів, варто додати логіку яка буде їх враховувати при виконанні певних запитів. Покращити захист зберігання паролів користувачів та, за можливості, ввести токени для користувачів, які матимуть свою тривалість та ставатимуть неактивними після закінчення цього часу;

– запис багатьох різних даних в одну таблицю: до прикладу, можна було б розгрупувати сенсори краще – розділити різні типи сенсорів на різні таблиці, а не зберігати всі в одній, це покращить розуміння та вибірку можливих даних, спростить мабутні запити;

– зробити краще розділення на зовнішніх запитах: наприклад, не робити загальні запити в таблиці без умов, це не логічно, небезпечно та не клієнтоорієнтовано, адже в такому випадку майбутні клієнти чи користувачі бачитимуть не лише свої записи;

– прибрати не потрібні таблиці: ті таблиці, які можна спростити до базових enum в C# чи ще якогось такого узагальнення не повинні бути в базі даних;

– ввести різноманітні скрипти або заплановані задачі: оскільки ця база даних в перспективі буде займати багато місця через кількість даних які вона зберігає, потрібно ввести якісь тригери чи інші типи запланованих задач, які

періодично очищатимуть базу від надто старих занчень чи перенеситимуть їх в окрему базу даних чи архівуватимуть якимось іншим чином;

– налаштувати резеревне копіювання на сервері: потрібно, щоб не втратити дані у випадку якоїсь критичної помилки чи будь-якого збою роботи;

– розділення даних на менші шматки: у випадку, якщо таких даних буде забагато і їх буде надто важко опрацьовувати;

– валідація даних на сервері: в момент отримання запиту, варто валідувати отримані дані, щоб не пробувати додати в базу даних якісь не правильні чи побиті дані.

Також для безпеки бази даних, завжди можна спробувати налаштувати розділення на кілька існуючих інстансів на різних серверах та зробити балансування навнатження: відправляти поточні запити на той сервер, на якому зараз найменше навантаження. За таким принципом працюють бази даних в GitHub. Те саме можна робити і для API, використовуючи балансування навантаження, системи обміну повідомленнями та інші засоби, хоча в такому випадку варто передивитись архітектуру створеного рішення.

В перспективі, для зручності використання, також можна створити сайт, який матиме доступ до цього API та виводитиме зручне керування девайсами, адже при архітектурі бази даних можливість зберігати різноманітні конфігурації девайсів, а також зміни їх передбачалась, а також показуватиме статистику у вигляді графіків. Це покращить досвід користування розробленою системою.

# ВИСНОВКИ

Обробка великої кількості подій у режимі реального часу є критичною вимогою для багатьох сучасних додатків і систем, включаючи фінансові сервіси, онлайн-ігри, системи IoT (Інтернет речей), соціальні мережі та моніторинг систем безпеки. Для забезпечення ефективного функціонування таких систем потрібні бази даних, які можуть швидко і надійно обробляти величезні обсяги даних у реальному часі.

За результатами проведеного дослідження варто зробити висновок, що вибір бази даних для обробки великої кількості подій у режимі реального часу залежить від специфічних вимог до швидкості, масштабованості та надійності. Комбінація різних технологій може забезпечити комплексне рішення, здатне задовольнити вимоги сучасних додатків у реальному часі. З урахуванням цих критеріїв, можна створити ефективну, масштабовану та надійну інфраструктуру для обробки подій у реальному часі, що дозволить вчасно реагувати на зміни та аналізувати дані швидко і ефективно.

Під час виконання кваліфікаційної роботи проведено аналіз існуючих рішень для обробки даних в реальному часі, а саме представлено різні типи архітектури прикладного програмного інтерфейсу (API) для взаємодії клієнтів (пристроїв) з базою даних.

Розроблено модель бази даних, що відповідає потребам обробки великих потоків даних, на основі окремих об’єктів визначених в процесі аналізу поставленого завдання.

Спроєктовано та реалізовано прототип бази даних та інтерфейси для взаємодії пристроїв чи/або клієнтів з нею, а саме: створено RESTful API на базі технології .NET – Minimal API з використанням ORM Dapper.

Оцінено ефективність та масштабованість розробленої бази даних, на основі синтетичних даних, що були створені власноруч та відкритих даних, що були взяті з відкритих джерел для додавання реалізму та репрезентації різноманітності інформації.

Як бачимо з отриманої інформації, основними критеріями вибору бази даних для обробки великої кількості подій є: швидкість та продуктивність, масштабованість, автоматичне балансування навантаження, надійність та відмовостійкість, підтримка реального часу, інтеграція аналітики. Для обробки великої кількості подій у режимі реального часу необхідно використовувати сучасні бази даних та інструменти, які забезпечують високу продуктивність, масштабованість, надійність та можливість обробки даних у реальному часі.

На основі дослідження можемо зробити висновок, що обробка великої кількості подій у режимі реального часу вимагає від баз даних високої продуктивності, масштабованості, і надійності. Для таких сценаріїв особливо підходять системи з можливістю горизонтального масштабування, низькою затримкою доступу до даних, і підтримкою високої швидкості запису та обробки подій.

Розроблена система має потенціал для ефективного використання в малих IoT сервісах, призначених для домогосподарств. Вона здатна обробляти та зберігати дані, що надходять від різноманітних пристроїв та датчиків, розміщених у будинку. Завдяки своїм можливостям, ця система не тільки забезпечує надійне зберігання даних, але й дозволяє проводити узагальнення та аналіз отриманої інформації.

# ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Reddy M. API design for C++. Elsevier. 472 c. URL: https://tailieumienphi.vn/doc/ebook-api-design-for-c-martin-reddy-0487tq.html (дата звернення: 06.04.2024).

2. What is an API? A beginner's guide to api *Postman*. URL: https://www.postman.com/what-is-an-api/ (дата звернення: 06.04.2024).

3. ISO/IEC 2382:2015. Information technology Vocabulary. Чинний від 2022-10-01. Вид. офіц. 2022. URL: https://www.iso.org/standard/63598.html (дата звернення: 06.04.2024).

4. Типи баз даних *Дія.Освіта – IT-студії*. URL: https://it-osvita.diia.gov.ua/task/item/2ce93257-df68-4072-bca6-65163b9f4278 (дата звернення: 06.04.2024).

5. Gruber M. Understanding SQL. San Francisco : Sybex. 434 c. URL: https://archive.org/details/isbn_9780895886446 (дата звернення: 06.04.2024).

6. Що таке SQL та Бази даних? *aCode*. URL: https://acode.com.ua/sql-intro/#toc-3 (дата звернення: 06.04.2024).

7. Stack overflow survey. *Stack Overflow Insights* URL: https://survey.stackoverflow.co/ (дата звернення: 06.04.2024).

8. Nirav K. Top 6 most popular API architecture styles you need to know (with pros, cons, and use cases). *DEV Community*. URL: https://dev.to/kanani_nirav/top-6-most-popular-api-architecture-styles-you-need-to-know-with-pros-cons-and-use-cases-564j (дата звернення: 06.04.2024).

9. Андрій Денисенко. Вступ до REST API – RESTful вебсервіси. *Робот Дрімс*. URL: https://robotdreams.cc/uk/blog/466-vstup-do-rest-api-restful-vebservisi (дата звернення: 06.04.2024).

10. Simple object access protocol (SOAP): A messaging protocol *CQR*. URL: https://cqr.company/wiki/protocols/simple-object-access-protocol-soap-a-messaging-protocol/ (дата звернення: 06.04.2024).

11. What is GraphQL? *Hygraph*. URL: https://hygraph.com/learn/graphql (дата звернення: 06.04.2024).

12. Характерник Я. Приклад gRPC-мікросервісу на Go. *DOU.UA*. URL: https://dou.ua/lenta/articles/grpc-microservice-go/ (дата звернення: 06.04.2024).

13. WebSocket. *JavaScript*. URL: https://uk.javascript.info/websocket (дата звернення: 06.04.2024).

14. What is a webhook? *Red Hat* URL: https://www.redhat.com/en/topics/automation/what-is-a-webhook (дата звернення: 06.04.2024).

15. Muneeruddin M. Event-Driven architecture & modern trends. *LinkedIn* URL: https://www.linkedin.com/pulse/event-driven-architecture-modern-trends-mohd-muneeruddin/ (дата звернення: 12.04.2024).

16. What emerging event-driven architecture trends should you know about? *LinkedIn* URL: https://www.linkedin.com/advice/3/what-emerging-event-driven-architecture-5gzbe (дата звернення: 12.04.2024).

17. Стек у програмуванні: основи та застосування. *FoxmindEd*. URL: https://foxminded.ua/stek-u-prohramuvanni/ (дата звернення: 12.04.2024).

18. Як розробник повного стеку використовує діаграми Tech Stack. *ITpedia* URL: https://uk.itpedia.nl/2023/09/10/effectieve-samenwerking-hoe-een-full-stack-developer-tech-stack-diagrammen-gebruikt/ (дата звернення: 12.04.2024).

19. Як правильно вибрати стек для свого проєкту *WEZOM* URL: https://wezom.com.ua/ua/blog/tehnologicheskij-stek-proekta (дата звернення: 12.04.2024).

20. A dapper tutorial for C# and .NET core *Learn Dapper* URL: https://www.learndapper.com/ (дата звернення: 12.04.2024).

21. Database management systems (DBMS) comparison: mysql, postgr. *AltexSoft*. URL: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/ (дата звернення: 22.04.2024).

22. Creating, altering, and removing stored procedures - SQL server. *Microsoft Learn* URL: https://learn.microsoft.com/en-us/sql/relational-databases/server-management-objects-smo/tasks/creating-altering-and-removing-stored-procedures?view=sql-server-ver16 (дата звернення: 22.04.2024).

23. What is CRUD? *Codecademy*. URL: https://www.codecademy.com/article/what-is-crud (дата звернення: 22.04.2024).

24. What are ACID Transactions? *Databricks*. URL: https://www.databricks.com/glossary/acid-transactions (дата звернення: 22.04.2024).

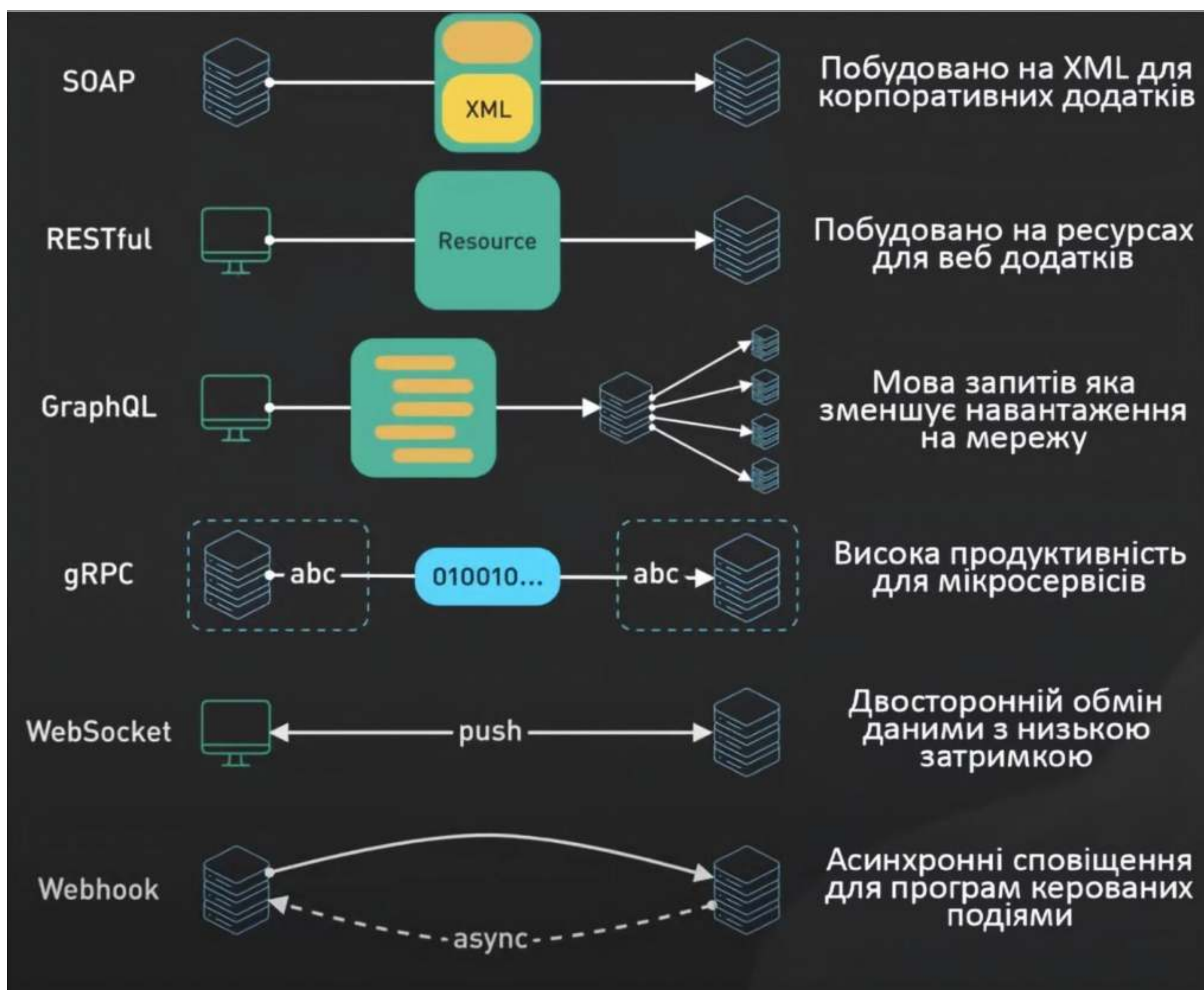
25. Що таке HTTP endpoint? *Ruby Developers* URL: https://rubydevelopers.org/t/http-endpoint/135 (дата звернення: 24.04.2024).

26. Tutorial: create a minimal API with ASP.NET core. *Microsoft Learn* URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-8.0&tabs=visual-studio (дата звернення: 23.04.2024).

27. API documentation & design tools for teams *Swagger*. URL: https://swagger.io/ (дата звернення: 26.04.2024).

# ДОДАТКИ

## Додаток А
## Схематичне зображення взаємодії напопулярніших архітектур API в клієнт-серверній побудові додатків



# Додаток Б
## Схема спроєктованої бази даних