Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Допущено до захисту»
Завідувач кафедри
комп'ютерних наук
доктор технічних наук, професор
_____ А.П. БОНДАРЧУК
(підпис)
« ___ » _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «Магістр»
Спеціальність 122 Комп'ютерні науки
Освітня програма 122.00.02 Інформаційно-аналітичні системи

Тема роботи:

«АПК НА ПЛАТФОРМІ ESP32 ДЛЯ МОНІТОРИНГУ ЯКОСТІ ПОВІТРЯ»

Виконав
студент групи ІАСм-1-24-1.4д
Степанець М.В.

_____
(підпис)

Науковий керівник
кандидат технічних наук, доцент
МЕЛЬНИК І. Ю.

_____
(підпис)

Київ – 2025

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

«Затверджую»
Завідувач кафедри
комп’ютерних наук, кандидат
технічних наук, доцент
_____ Машкіна І.В.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

«АПК НА ПЛАТФОРМІ ESP32 ДЛЯ МОНІТОРИНГУ ЯКОСТІ ПОВІТРЯ»

Виконавець: студент групи ІАСм-1-24-1.4д
Степанець Михайло Володимирович

1. Вихідні дані: *Наукові публікації пов'язані з моніторингом повітря та застосуванням IoT; документація для датчиків та мікроконтроллера ESP32; документація для ASP.NET Web API, Microsoft SQL Server та Android Studio; приклади рішень моніторингу повітря у приміщеннях.*

2. Основні завдання: *аналітичний огляд відкритих джерел та рішень на основі мікроконтролерів та з використання IoT. Визначити актуальність, мету, предмет та об’єкт дослідження. Впровадити новизну та визначити основні методи перевірки практичного рішення на працездатність. Розробити структуру та зміст роботи. Розробити архітектуру комплексного додатку SenseData, яка складається з: Esp32 вузла, сервера Web API, SQL-серверу та Android додатку. Створити працюючий прототип системи. Також провести експериментальні випробування всіх шарів на працездатність та перевірити на швидкодію та затримки. Оформити магістерську роботу згідно з методичними рекомендаціями кафедри.*

3. Пояснювальна записка: *Обсяг – до 60 сторінок формату А4 з дотриманням вимог методичних рекомендацій кафедри; містить аналітичний*

огляд, опис всіх трьох шарів архітектури SenseData, методи та результати експериментів, висновки та список використаних джерел.

4. Графічні матеріали: *Схеми архітектури SenseData на всіх шарах, схеми вузла ESP32 та підключення сенсорів в середовищі Fritzing, діаграми jcyjdyb[алгоритмів, ER-діаграма бази даних, екрани мобільного додатка, а також приклади експериментальних графіків.*

5. Додатки: Додаток А, Додаток Б, Додаток В, Додаток Г, Додаток Д.

6. Строк подання роботи на кафедру: *«1» грудня 2025 р.*

Науковий керівник	Виконавець:
к.т.н., доцент	
_____ Мельник І. Ю.	_____ дата
дата	

Анотація кваліфікаційної роботи

Дипломна робота: 134 с., 27 рис., 3 таб., 40 посилань.

Актуальність: Перш за все через зростання міст, тривалого перебування в приміщеннях та мікроклімат кімнати, який має важливе значення для здоров`я та комфорту людини. Більшість кімнат не мають пристроїв для постійного вимірювання основних критеріїв мікроклімату, а вимірювання вручну не завжди можливі. Але дешевизна, доступність та підтримка всіх основних протоколів робить ESP32 ідеальним рішенням для IoT систем та передачі даних.

Об'єкт дослідження: Моніторинг мікроклімату та повітря в приміщеннях.

Предмет дослідження: програмно-апаратна IoT-система SenseData, яка збирає, зберігає, обробляє та показує дані IAQ та поради користувачеві.

Мета роботи: Розробити працюючий прототип SenseData з основними алгоритмами взаємодії, веб-сервер Web API, та Android-додатком, а також перевірити затримки, надійність передачі даних.

Завдання роботи: Розглянути готові рішення в сфері контролю повітря та мікроклімату, основні принципи IoT, основні платформи та датчики для створення робочого вузла, проаналізувати модель безпеки на всіх етапах, розробити архітектура та структуру SenseData.

Методи дослідження: Аналіз відкритих джерел, експерименти з затримками та відсотками успіхів, сценарії навантаження веб серверу Web API.

Практичне значення дослідження: Розробка прототипу який включає: вузол ESP32, безпечний веб-сервер Web API, SQL-сервер для зберігання даних та мобільний застосунок Android для зручної взаємодії з сервером. Прототип можна використовувати для шкіл та офісів як дешевий аналог існуючих рішень, а масштабованість вузла дозволяє розширювати діапазон датчиків тим самим розширювати і корисність вузла.

Ключові слова: Мікроклімат, якість повітря, IoT, ESP32, Web API, SQL Server, Android-застосунок.

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

AES — Advanced Encryption Standard

API — Application Programming Interface

BLE — Bluetooth Low Energy

DTO — Data Transfer Object

ESP32 — Espressif Systems 32-bit Microcontroller

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

IAQ — Indoor Air Quality

IoT — Internet of Things

JSON — JavaScript Object Notation

JWT — JSON Web Token

MVVM — Model–View–ViewModel

NTP — Network Time Protocol

NVS — Non-Volatile Storage

OWM — OpenWeatherMap

REST — Representational State Transfer

SQL — Structured Query Language

UI — User Interface

UX — User Experience

Зміст

ВСТУП

Інформаційні технології сильно змінюють наше життя. Стали доступні можливості створювати мережі з датчиків, мікроконтролерів, механізмами, які взаємодіють між собою. Ці мережі лежать в основі виробництв, логістики, інженерних рішень, систем будівель. Чим менше вплив людського фактора в системі тим краще реалізовано автоматизація, тим надійнішим та ефективнішим є рішення.

Суть поняття IoT об'єднує розумні мережі разом з пристроями. Важливим напрямом є моніторинг мікроклімату та якості повітря в приміщеннях. Юзерам потрібні масштабовані, прості а саме головне надійні та безпечні рішення для збору, обробки, зберігання та візуалізації даних. В поєднанні недорогих мікроконтролерів з мобільним додатком та веб-сервером дозволяє повноцінно реалізувати саме таку систему.

Актуальність роботи зумовлена перш за все через зростання міст, тривалого перебування в приміщеннях та мікроклімат кімнати, який має важливе значення для здоров'я та комфорту людини. Більшість кімнат не мають пристроїв для постійного вимірювання основних критеріїв мікроклімату, а вимірювання вручну не завжди можливі. Але дешевизна, доступність та підтримка всіх основних протоколів робить ESP32 ідеальним рішенням для IoT систем та передачі даних.

Об'єкт дослідження – моніторинг мікроклімату та повітря в приміщеннях.

Предмет дослідження – програмно-апаратна IoT-система SenseData, яка збирає, зберігає, обробляє та показує дані IAQ та поради користувачеві.

Мета роботи – Розробити працюючий прототип SenseData з основними алгоритмами взаємодії, веб-сервер Web API, та Android-додатком, а також перевірити затримки, надійність передачі даних.

Завдання роботи:

1. Розглянути IAQ-показники, сенсорні платформи (DHT11/BME280/BMP280, MQ-2, фотодатчик) та принципи IoT (Wi-Fi, BLE, REST, мобільний клієнт).
2. Обґрунтувати модель безпеки (JWT для API, AES-шифрування Wi-Fi через BLE, ролі власник/гість).
3. Створити архітектуру SenseData: протоколи, DTO, структуру бази даних, механізми кешування/ETag, правила доступу, основні екрани мобільного додатку та їх стани.
4. Реалізувати основні алгоритми: BLE-налаштування, цикл Wi-Fi/POST передачі даних, створення кімнати через BLE, формування порад.
5. Підключити API погоди для врахування зовнішніх умов.
6. Провести експерименти: виміряти затримки сенсор → сервер → телефон, відсоток успішних POST, стабільність BLE-налаштування, повторюваність вимірювань, стійкість Web API до навантажень.
7. Підсумувати результати, вказати на обмеження та можливості розвитку.

Практичне значення. Розробка прототипу який включає: вузол ESP32, безпечний веб-сервер Web API, SQL-сервер для зберігання даних та мобільний застосунок Android для зручної взаємодії з сервером. Прототип можна використовувати для шкіл та офісів як дешевий аналог існуючих рішень, а масштабованість вузла дозволяє розширювати діапазон датчиків тим самим розширювати і корисність вузла.

Методи дослідження. Аналіз відкритих джерел, експерименти з затримками та відсотками успіхів, сценарії навантаження веб серверу Web API.

Структура роботи. Дипломна робота складається зі вступу, трьох розділів основної частини, висновків, списку використаних джерел та додатків. У першому розділі подано аналіз теоретичних основ моніторингу мікроклімату та якості повітря в приміщеннях, а також огляд суміжних рішень і вибір

апаратно-програмного стеку для системи SenseData.

Другий розділ присвячений обґрунтуванню архітектури системи, опису взаємодії ESP32, Web API, SQL Server та мобільного клієнта Android, а також розробці алгоритмів провізіонування, передавання телеметрії та формування порад.

У третьому розділі наведено методику випробувань, результати експериментів для вузла ESP32, веб-API, мобільного застосунку та BLE-провізіонування, а також аналіз отриманих показників продуктивності й надійності системи.

РОЗДІЛ 1 ПОНЯТТЯ МОНІТОРИНГУ МІКРОКЛІМАТУ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд задач моніторингу якості повітря (IAQ) і мікроклімату

Якість повітря у приміщеннях (Indoor Air Quality (Рис 1.1)) суттєво впливає на комфорт, продуктивність і здоров'я людей [1; 2]. Люди проводять понад 80% часу в приміщеннях, тому їхній мікроклімат стає основним середовищем для життя. Сучасні будівлі, які важко продуваються, мають енергоефективні вікна та герметичні двері, що знижує природний обмін повітря [1].

Через збільшення часу перебування вдома, працюючи віддалено, у приміщеннях накопичуються забруднення, коливається вологість і погіршується тепловідчуття. Навіть невеликі зміни в параметрах повітря можуть викликати втому, головний біль, подразнення слизових оболонок, сухість шкіри та зниження розумової діяльності.

Рисунок 1.1 – Структурне значення IAQ[29]

Моніторинг якості повітря в приміщенні (IAQ) включає постійне вимірювання фізичних і хімічних характеристик повітря. Ми будуємо графіки змін у часі та аналізуємо їх, щоб підтримувати комфортні умови [1].

Фізичні параметри включають температуру, вологість, швидкість руху повітря та теплове випромінювання.

Хімічні параметри включають рівень вуглекислого газу (CO_2) як показник вентиляції, дрібні частинки, леткі органічні сполуки та продукти згоряння. Освітленість також може бути врахована як показник денного режиму в приміщенні, що допомагає відрізнити нічні значення і краще зрозуміти поведінку людей [1; 2].

Оцінка IAQ базується на моделях теплового комфорту (PMV/PPD), адаптивних підходах і нормативних діапазонах для окремих показників. У будь-якому випадку, важливо, щоб вимірювання були точними в просторі і часі.

Нормативні документи визначають стандартні діапазони [3-5]:

- температура повітря 20–24 °C для більшості видів діяльності в легкому одязі,
- вологість 40–60 % для забезпечення теплового комфорту і запобігання проблемам зі слизовими оболонками та пліснявою(Рис. 1.2).
- швидкість руху повітря не повинна перевищувати 0,2 м/с, щоб уникнути протягів взимку і надмірного охолодження шкіри.
- рівень CO_2 зазвичай підтримується нижче 800–1000 ppm як показник нормальної вентиляції, враховуючи, що це показник кількості людей і ефективність вентиляції, а не прямий вимір отруєння.
- для дрібних частинок, особливо $PM_{2.5}$, головне – зменшити їх різкі збільшення, викликані приготуванням їжі або курінням, оскільки навіть короткочасні сплески можуть значно збільшити щоденний рівень забруднення.

Рисунок 1.2 – Норми вологості в кімнатах для комфортного перебування[30]

Метод моніторингу показників довкілля варто починати з ретельного вимірювання та перевірки точності даних. Навіть недорогі прилади можуть мати неточності, як-от регулярні відхилення, випадкові зміни, нелінійність та поступові зсуви. Тому важливо:

- спочатку налаштувати прилади;
- перед тривалим використанням порівняти їх із точними приладами;
- час від часу перевіряти їх у незмінних умовах.

Для датчиків газу важливий час нагрівання та стабілізації. Для цифрових вимірювачів температури і вологості слід враховувати нагрівання від плати та інших деталей. Під час обробки даних корисні надійні фільтри: медіанний для усунення різких відхилень, ковзне середнє з різним періодом для згладжування шуму та виявлення трендів, а також перцентилі для визначення основних рівнів [1].

Аналізувати потрібно не окремі показники, а їх сукупність та тривалість. Наприклад:

- «перегрів і сухість» - висока температура та низька вологість протягом певного часу;
- «тривала висока вологість» - ризик утворення конденсату та появи грибка;

- «застій повітря» - рівень CO_2 повільно повертається до звичайного;
- «короткочасна поява диму або летких органічних сполук» - різкі стрибки показників відповідних датчиків.

Для зменшення кількості помилкових спрацювань використовуйте порогові значення з гістерезисом і мінімальним часом перевищення. Це зручно реалізувати у вигляді системи станів, яка запам'ятовує тривалість епізоду та фіксує початок і кінець.

Вологість - важлива тема. Її джерела: люди, приготування їжі, сушіння одягу, рослини. Зменшити її можна вентиляцією та поглинанням вологи матеріалами (гіпсокартон, дерево, тканини). Матеріали згладжують зміни вологості, але сповільнюють її повернення до норми; тривале повернення до норми часто вказує на недостатній повітрообмін. Для запобігання пошкодженням слід вимірювати температуру поверхонь (місця з низькою температурою) і оцінювати ризик утворення конденсату [3–5].

Рівень CO_2 у будинках і навчальних закладах - хороший показник заповненості та ефективності вентиляції. Він не є шкідливим у невеликих кількостях, але його зміни показують, наскільки добре працює вентиляція: швидке зростання і повільне зниження вказує на її неефективність. Корисні показники: швидкість збільшення під час перебування людей та час зниження після їх відходу. Вони дозволяють оцінити вентиляцію без складних вимірювань. У випадку пилу та летких органічних сполук важливо знати їх джерела (приготування їжі, свічки, аерозолі) і звертати увагу не лише на наявність стрибків, але й на швидкість очищення повітря, яка залеж ить від вентиляції.

Щоб результати вимірювань були точними, у мережах датчиків слід враховувати відмінності між ними. Корисні поради:

- порівняння показників між різними пристроями;
- обмін датчиками місцями для виявлення систематичних помилок;

- запис інформації про події (відкриття вікон, увімкнення витяжки, використання зволожувача), щоб відрізняти зміни в середовищі від похибок вимірювання.

Для тривалих спостережень потрібно планувати регулярне налаштування та перевірку приладів, особливо після перерв або ремонтів. Результати слід показувати користувачам у зрозумілій формі. Корисні графіки з лініями, що позначають допустимі межі, позначки подій та індикатори з кольорами (зелений/жовтий/червоний), а також прості пояснення про їх значення та необхідні дії. Важливі повідомлення потрібно виділяти, обмежувати їх кількість і вказувати очікуваний результат дій (наприклад, час, необхідний для повернення до норми при провітрюванні). Інформативність підвищується, коли поточні дані порівнюються з типовими рівнями для певного часу доби або пори року [1; 2].

Відстеження якості повітря та мікроклімату – це комплексний процес, що включає отримання точних вимірювань, їх аналіз для визначення практичних ситуацій та вибір відповідних дій. Ключові вимоги:

1. точність вимірювань,

2. достатня кількість даних у просторі та часі,

3. зрозумілий аналіз, що враховує як короткочасні події, так і загальний вплив.

Такий підхід дозволяє перейти від випадкових вимірів до контрольованого мікроклімату, що сприяє здоров'ю, комфорту та продуктивності в житлових, навчальних і робочих приміщеннях.

1.2 Огляд платформ

Апаратно-програмна платформа визначає точність, стабільність, масштабованість і вартість системи моніторингу. Вибір сенсорів впливає на можливість аналізу часових рядів і розробки рекомендацій.

Для дому та лабораторій підходить мікроконтролер ESP32. Він поєднує обчислювальні можливості з інтерфейсами зв’язку (Wi-Fi/BLE) і периферією (I^2C, SPI, UART, АЦП, таймери) [9; 10; 12; 13]. Це зменшує кількість зовнішніх компонентів, спрощує проєктування та забезпечує безпечне налаштування й телеметрію з обробкою даних.

ESP32 дозволяє синхронізувати опитування різних сенсорів, мінімізуючи затримки. Це покращує точність даних і дозволяє обробляти їх безпосередньо на пристрої (медіанне згладжування, ковзні середні). Також можна обчислювати показники вологості й визначати індикатори подій. Це підвищує якість даних і зменшує залежність від зв’язку. Кільцевий буфер зберігає дані при збоях живлення або мережі [9; 10].

Важлива інженерна підтримка. Платформа Arduino (Arduino IDE/PlatformIO) спрощує розробку й відтворення експериментів [14]. Наявність драйверів для сенсорів зменшує ризики інтеграції та прискорює створення прототипів. Для навчання та дому важлива прозорість рішень, ремонтопридатність і доступність, а не висока продуктивність.

Сенсори слід розглядати як джерела різних сигналів. Цифрові термогігрометри й барометри вимірюють клімат; газові сенсори фіксують епізоди; фотосенсори визначають час доби [9; 10; 11]. Важливо розуміти ролі сенсорів: базові канали забезпечують вимірювання для логіки й показників; індикаторні - визначають події; контекстні - допомагають інтерпретувати зміни.

Зв’язок виконує різні функції: BLE для налаштування та локальних операцій (передача конфігурації, обмін подіями), Wi-Fi для телеметрії на сервер. Така схема розділяє безпеку та надійність: BLE працює близько до пристрою, Wi-Fi забезпечує передачу даних. Важливі офлайн-буфер з повторними спробами відправлення, тайм-аути й синхронізація часу для точної мітки подій.

Живлення впливає на точність і довговічність. Правильний стабілізатор, фільтрація шумів, розташування елементів і вентиляція зменшують самонагрів і

термальні градієнти. Для автономних пристроїв важливі режими сну, для стаціонарних — електромагнітна сумісність. У будь-якому випадку корисне ведення журналу подій (перезавантаження, зміна конфігурації, відмови каналів).

АЦП ESP32 підходить для аналогових каналів, але потребує уваги до нелінійності та шумів. Рекомендується калібрування, надсемплінг і акуратна топологія аналогових трас [12]. Бажано розділити землю аналогового та цифрового сегментів і мінімізувати перехресні наведення.

Інтеграція сенсорів зводиться до інтерфейсів і механіки. На I²C важливі короткі траси; для SPI - чітка топологія, більш детальні відмінності наведені у Таблиці 1.1. [12] Корпус забезпечує дифузію повітря до сенсорів, захист від сонця й мінімізацію потоків повітря. Розміщення елементів враховує теплові міркування: сенсори з підігрівачем розташовуються вище, а точні сенсори - віддалік від стабілізаторів, із вільною конвекцією.

Таблиця 1.1 – Основні відмінності I²C від SPI

Характеристика	**I²C**	**SPI**
Підтримка мультимайстра	Підтримує режим «multi-master»	Не підтримує «multi-master»
Кількість ліній	2 дроти (SDA, SCL)	4 дроти (MOSI, MISO, SCK, SS)
Режим обміну	Напівдуплекс	Повний дуплекс
Швидкодія	Повільніший	Швидший

Планування опитування сенсорів враховує часові масштаби каналів: повільні параметри знімаються рідше, індикаторні — частіше. Узгодження частот, фільтрації та вікон згладження дозволяє зменшити шум, а узгоджена часово-фазова схема запобігає помилкам.

Таким чином, комбінація ESP32 як інтегрованої платформи зв’язку та керування, Arduino-екосистеми як середовища швидкої й відтворюваної розробки, а також набору взаємодоповнювальних сенсорних модулів, підкріплена дисципліною інтерфейсів і живлення, формує збалансовану основу для систем моніторингу мікроклімату [9; 10].

1.3 Аналіз суміжних рішень і вибір стеку

Архітектуру системи для моніторингу мікроклімату можна створити, використовуючи різні інструменти – від готових рішень до налаштованих компонентів. При виборі важливо враховувати не лише технічні характеристики, але й освітні цілі, стабільність інструментів, розвиненість екосистеми, умови ліцензування та можливість повторити експерименти [6; 7].

Для периферійного рівня можна розглянути три варіанти: готові фреймворки (ESPHome/Home Assistant), середовище Arduino та ESP-IDF з RTOS.

Перший варіант дозволяє швидко почати роботу та інтегруватися з системами автоматизації будинку, проте обмежує контроль над часом опитування та обробки сигналів, а також ускладнює створення протоколу взаємодії з сервером. ESP-IDF, навпаки, надає максимум можливостей для керування, але вимагає більше коду для простих задач і є складнішим для новачків. Для навчального прототипу, де важлива швидкість змін, повторюваність і прозорість, доцільно використовувати Arduino з можливістю використання окремих модулів ESP-IDF (наприклад, для шифрування AES або керування енергоспоживанням) [12; 14] . Такий підхід забезпечує доступ до великої кількості бібліотек, передбачуваний процес збирання та достатній контроль над часом процесів, за умови правильного використання таймерів.

Для серверної частини існують різні платформи: Node.js/TypeScript з Express/NestJS, Python з FastAPI/Django, Go з net/http або Gin, а також .NET. При їх порівнянні важливо звертати увагу на пропускну здатність при обробці великих обсягів даних телеметрії, зручність опису контрактів, засоби безпеки (JWT, ролі,

політики), підтримку кешування та асинхронних операцій [17; 18;]. У даному проєкті обрано .NET 9 для Web API з кількох причин.

По-перше, сумісність з Visual Studio та NuGet полегшує створення структури сервісу, інтеграцію JWT, створення документації та впровадження політик доступу[17; 18].

По-друге, сучасні можливості мінімальних API та обробки асинхронних операцій дозволяють приймати великі обсяги даних телеметрії та забезпечувати стабільну швидкість для мобільного застосунку.

По-третє, структура DTO/Services/Repositories у поєднанні з автоматичним генеруванням клієнтів за OpenAPI підтримує повторюваність і контроль змін контрактів, на Рис 1.3 показана детальна структура додатку .NET 9 для Web API [17; 18; 20].

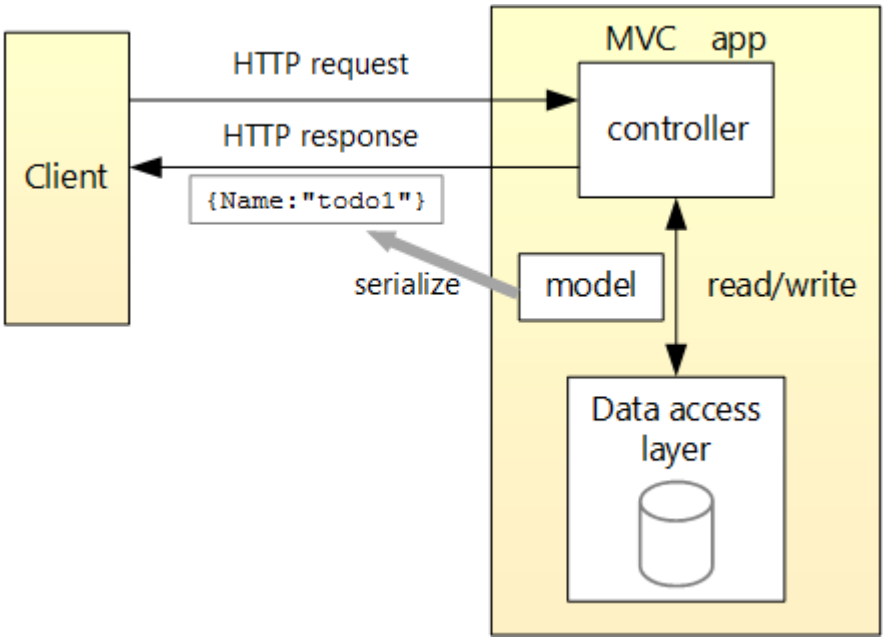

Рисунок 1.3 – Детальна структура додатку .NET 9 для Web API[31]

Альтернативні рішення на Node.js або Python можуть бути простішими для початківців. Все ж, якщо потрібна сувора типізація даних у довгострокових проєктах, .NET дає більш чітку структуру. Це допомагає зменшити кількість помилок, пов'язаних з невідповідністю форматів даних [17; 18].

Для зберігання даних телеметрії можна використовувати різні системи, такі як SQL Server, PostgreSQL, InfluxDB, TimescaleDB або MongoDB. У цьому випадку обрали SQL Server, оскільки він має зрозумілу структуру даних, гарантує надійність транзакцій, має прості механізми індексації та розвинені засоби адміністрування [15; 16]. Також, він дозволяє поєднувати необроблені дані з агрегованими даними в межах однієї системи. Для невеликих проєктів з моніторингу SQL Server забезпечує хорошу продуктивність без потреби використовувати окремі технології для зберігання агрегованих даних.

Інші рішення для часових рядів кращі для великих задач обробки даних із вимогами до стиснення, вибірки даних на льоту та політик зберігання. Проте, для поточного проєкту ці переваги не є критичними, а додаткові складнощі в управлінні небажані. Важливим аргументом на користь SQL Server є контроль цілісності даних: телеметрія чітко пов'язана з певними об'єктами (вузлом, кімнатою, користувачем) і правами доступу. Запити з мобільного застосунку можна швидко виконувати завдяки індексам за часом та ідентифікаторами.

Мобільний застосунок можна створити на базі Android/Java, Kotlin, Flutter або React Native. У цьому проєкті обрали Android/Java, оскільки Java відповідає вимогам і дозволяє показати класичні підходи до створення мобільних застосунків без зайвих ускладнень[23]. Java підтримує велику кількість навчальних матеріалів, бібліотек і інструментів для налагодження (Рис 1.4).

Для роботи з мережею використовується Retrofit/OkHttp з перехоплювачами для додавання інформації про авторизацію, повторних запитів у разі помилок та автоматичного оновлення токенів доступу. Хоча Kotlin зараз дуже популярний, Java не обмежує можливості проєкту, оскільки основні бібліотеки повністю

сумісні, а архітектура MVVM дозволяє відокремити різні стани застосунку (завантаження, помилка, вміст) та контролювати відновлення після збоїв[23; 24]. Для відображення графіків часових рядів використовується MPAndroidChart, оскільки ця бібліотека є стабільною, налаштовується та має достатню продуктивність для мобільних пристроїв[25].

Рисунок 1.4 – Великий вибір бібліотек Android Studio[32]

Варто згадати про хмарні платформи IoT, такі як AWS IoT Core, Azure IoT або ThingsBoard. Вони полегшують керування пристроями, маршрутизацію даних, надають брокери повідомлень і засоби безпеки[6; 7]. Проте, у навчальному процесі вони можуть створити залежність від конкретного постачальника послуг, ускладнити відтворення лабораторних робіт, збільшити витрати та не дають студентам глибоко зрозуміти протоколи обміну даними.

Для прошивки пристроїв можна використовувати PlatformIO або Arduino IDE. Обидва варіанти мають достатньо можливостей, але PlatformIO краще підходить для командної розробки та автоматизації, оскільки надає більше гнучкості в управлінні проєктом і залежностями. Arduino IDE простіший у використанні для одиночних проєктів, оскільки забезпечує швидкий цикл

розробки та не потребує глибоких знань[12; 14]. Цей варіант є кращим, коли основна увага приділяється алгоритмам обробки даних і протоколам обміну, а не інфраструктурі збірки.

Отже, вибір Arduino IDE для периферії, .NET 9/Visual Studio для серверного Web API, SQL Server для зберігання телеметрії та Android/Java для мобільного застосунку(Рис 1.5) є оптимальним для досягнення поставленої мети: керованість, відтворюваність і навчальна цінність. Це дозволяє швидко отримати робочий прототип, зберегти чітку структуру архітектури, формалізувати обмін даними та забезпечити якість даних для досліджень без зайвих ускладнень.

Рисунок 1.5 – Вибір стеку для проекту: Arduino IDE, .NET 9, SQL Server, Android Studio

1.4 Безпека: JWT для API, AES для Wi-Fi на ESP32, ролі «власник/гість»

У системах моніторингу мікроклімату, безпека потребує уваги на кожному етапі обробки даних і секретів конфігурації – від початкового налаштування пристрою до збереження даних і надання доступу користувачам. Оскільки дані можуть показувати присутність і звички людей, захист приватності та контроль доступу такі ж важливі, як і захист мережі[6; 7; 21].

Зазвичай, задачі ділять за рівнями:

- На пристроях (ESP32) – потрібен захист секретів Wi-Fi та ідентифікації пристрою.
- Під час передачі даних – шифрування TLS для зв'язку з сервером і обмеження повторної передачі даних конфігурації.

- На сервері – перевірка особистості та прав доступу за допомогою тимчасових токенів, обмеження прав і аудит.
- На стороні користувача – безпечне зберігання токенів і керування сесіями[19; 20].

Перше підключення пристрою до мережі краще робити близько до нього, без використання інтернету. Для цього можна використати BLE-провіжинг, коли телефон тимчасово налаштовує пристрій. Оскільки пароль Wi-Fi є секретними, їх не слід передавати відкрито, навіть на невеликій відстані. Одним зі способів захисту є шифрування даних за допомогою AES-CTR/AES-GCM з унікальним вектором ідентифікації та додатковими даними, що включають ідентифікатор пристрою (chipId) і час. Ключ для шифрування отримують на початку сесії (наприклад, через пін-код, QR-код або BLE-з'єднання з шифруванням) і видаляють після завершення налаштування. На ESP32 пароль Wi-Fi зберігається в пам'яті лише на короткий час у незашифрованому вигляді. У Flash-пам'яті його краще не зберігати зовсім (потрібно буде вводити заново при кожному перезавантаженні) або зберігати зашифрованим ключем, який неможливо отримати ззовні. У готових продуктах варто використовувати можливості чипа: Secure Boot для перевірки цілісності програми, Flash Encryption для зберігання секретів, апаратні генератори випадкових чисел і AES-алгоритми[12; 13; 21].

Детальний приклад перетвореного коду показано на Рис 1.6.

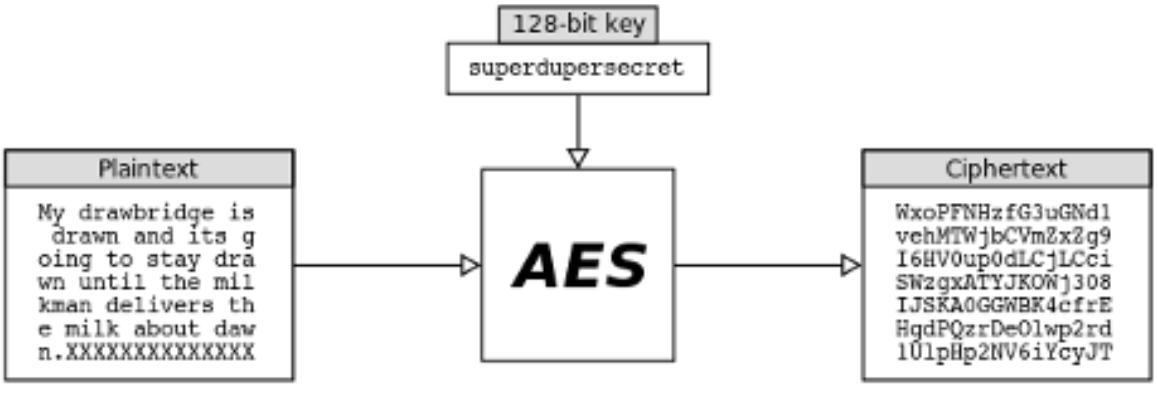

Рисунок 1.6 – Приклад шифрування текста за допомогою AES 128-bit key

Після підключення до мережі пристрій спілкується з сервером через зашифроване з'єднання (TLS) і підтверджує свою особу. Важливо, щоб ідентифікація пристрою була надійною (chipId або сертифікат/ключ), а протокол зв'язку обмежував можливість повторного використання запитів. Рекомендується формувати пакети даних телеметрії з послідовними мітками часу і, якщо потрібно, підписувати їх за допомогою HMAC-підпису, створеного на основі секрету пристрою, відомого серверу. Це знижує ризик підробки повідомлень у погано захищених мережах і дозволяє відрізнити повторно відправлені дані від нових. Потрібно чітко визначити дії у випадку помилок: пристрій повинен повторювати відправлення з поступовим збільшенням інтервалу, зберігаючи порядок пакетів і не дублюючи вже підтверджені дані [9; 10].

На сервері важлива перевірка особистості та прав доступу на основі JSON Web Token (JWT). Є два типи токенів: короткочасний access-токен для доступу до API і довготривалий refresh-токен для відновлення сесії без повторного введення пароля (Рис 1.7). У JWT міститься необхідна інформація: ідентифікатор користувача, ролі/права, ідентифікатор (якщо потрібно розділення на багатьох користувачів), термін дії (iat/nbf/exp) і, за потреби, прив'язка до пристрою (deviceId) для кращого контролю сесій. Access-токен повинен діяти кілька хвилин, refresh-токен – кілька днів/тижнів із можливістю відкликання. Варто використовувати ротацію refresh-токенів (видавати новий при кожному оновленні, а старий робити недійсним), зберігати відбитки (hash) refresh-токенів у базі даних і вести чорний список відкликаних токенів. Такий підхід відповідає принципам найменших прав: клієнт має лише ті права, які потрібні для конкретної дії, а сервер перевіряє права на кожний запит [17; 18; 22].

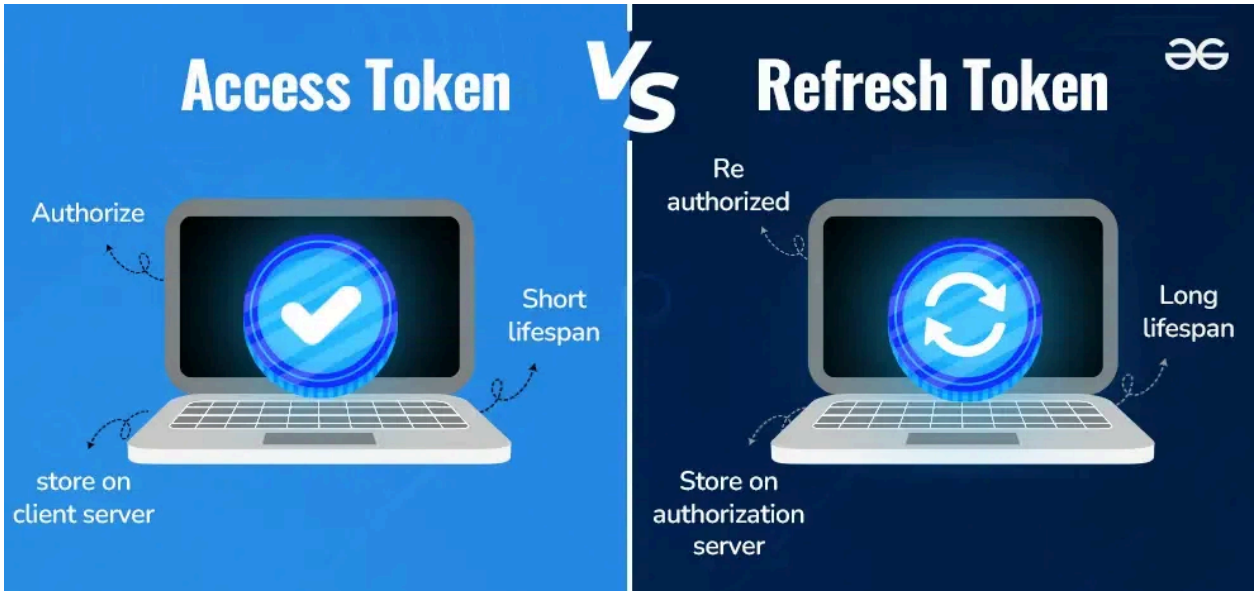

Рисунок 1.7 – Різниця між Access та Refresh токенів[33]

У системі моніторингу мікроклімату права доступу поділено на дві ролі: власник і гість. Власник створює кімнати, додає та видаляє датчики, налаштовує ліміти показників і надає доступ іншим користувачам. Гість може лише переглядати дані певних кімнат або датчиків, не маючи змоги змінювати налаштування або надавати доступ іншим (Рис 1.8).

Рисунок 1.8 – Приклад розділення прав доступу на власник/гість в додатку SenseData

Щоб уникнути несанкціонованого доступу між користувачами, кожний запит до системи повинен включати інформацію про власника. Ідентифікатори кімнат і датчиків перевіряються на відповідність контексту власника.

Гостьовий доступ надається за запрошенням або за допомогою тимчасових QR-кодів з обмеженим набором прав і терміном дії (Рис 1.9). Важливо, щоб скасування доступу відбувалося миттєво: відкликання прав гостя або видалення кімнати має призводити до негайної недійсності токенів, навіть якщо термін їх дії ще не закінчився. Це забезпечується перевіркою токенів на сервері [19–21].

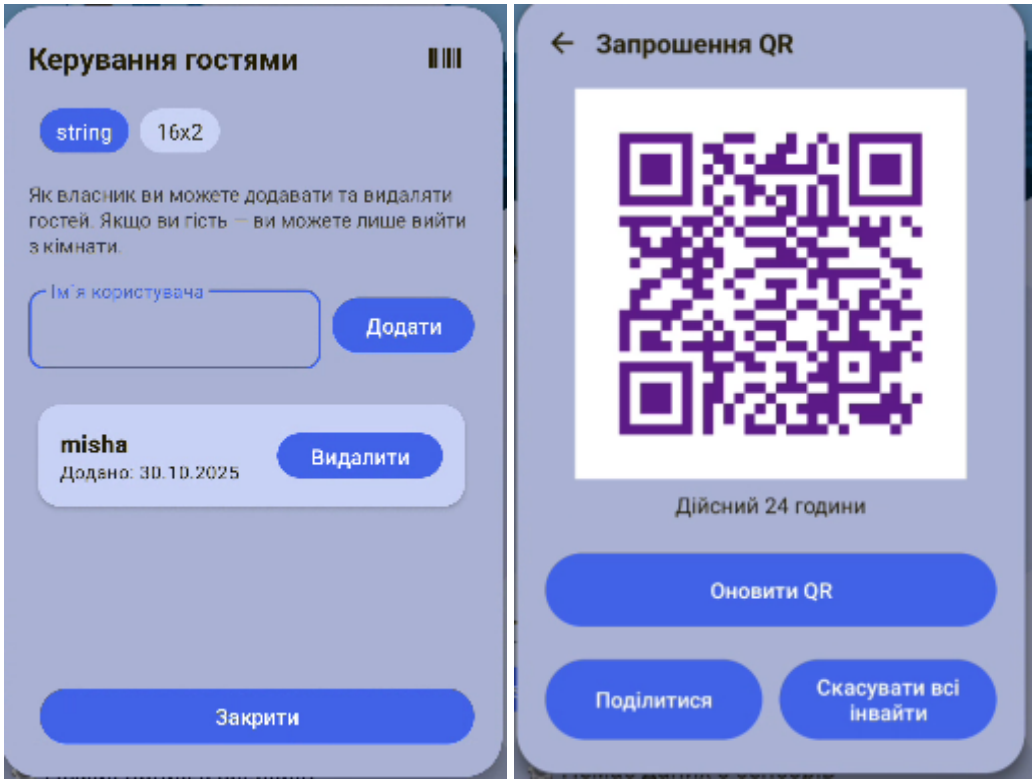

Рисунок 1.9 – Приклад надання доступу рівня гість в додатку SenseData

У мобільному застосунку важливим є безпечне зберігання конфіденційної інформації та реагування на можливі компрометації пристрою. Access-токени зберігаються в оперативній пам'яті, тоді як refresh-токени — у захищеному сховищі платформи (Keystore/Keychain) з автентифікацією за допомогою біометрії або PIN-коду, якщо цього вимагає модель загроз. Обсяг секретів у локальній базі даних повинен бути мінімальним, при цьому кеш вимірювань і довідників не містить персональних даних, які дозволяють ідентифікацію. У разі втрати пристрою, користувач може відкликати всі активні сесії зі свого облікового запису; сервер підтримує централізоване припинення сесій через анульовані refresh-токени. У процесі передачі даних клієнт здійснює перевірку сертифікатів

(включаючи pinning, якщо це можливо) і використовує суворі часові обмеження, щоб уникнути тривалих зависань і неконтрольованих повторних спроб.

Захист бекенду від зловмисного трафіку та помилок інтеграції є важливою складовою безпеки. Політики обмеження швидкості та квотування для ключових точок входу знижують ризики зловживань і DoS-атак. Обмеження розміру пакетів телеметрії та частоти запитів з одного вузла допомагають запобігти перевантаженню. Перевірка вхідних даних (DTO), контроль терміну дії часових міток і перевірка часової послідовності захищають від маніпуляцій з даними та порушень причинно-наслідкового зв'язку. Логи доступу з ідентифікаторами запитів і заголовками кореляції спрощують аудит та розслідування інцидентів. Аналіз відсотка успішних POST-запитів, розподілу затримок і частки помилок 401/403 на користувача допомагає виявляти аномалії в поведінці клієнтів і спроби підвищення привілеїв [19–21].

На периферії ESP32 необхідно розмежовувати привілеї компонентів. Задачі, що працюють із секретами, повинні виконуватися в контрольованих ділянках коду зі строгою перевіркою вхідних даних. Будь-які віддалені команди мають бути або локальними, або підтверджуватися підписаними командами з сервера. Необхідно враховувати сценарій фізичного доступу до пристрою, наприклад, зняття плати, читання Flash-пам'яті або використання відлагоджувальних інтерфейсів. Відключення JTAG у виробничій версії, шифрування Flash-пам'яті та відсутність секретів у відкритому вигляді в NVS зменшують наслідки такої компрометації. Для BLE-налаштування корисним є обмеження кількості спроб і часу доступу після апаратного скидання, щоб запобігти небажаним сесіям [12; 13].

Важливо враховувати типові загрози з боку користувачів, такі як повторне використання паролів, встановлення застосунку на пристроях з root-доступом або надання прав доступу гостям без обмеження терміну дії. Частину цих ризиків можна усунути за допомогою UX-рішень, таких як обов'язкова складність паролів, нагадування про термін дії запрошень, попередження про підозрілі спроби входу, обмеження кількості активних сесій і тихий час для сповіщень, щоб запобігти

повному відключенню сповіщень користувачами. Усі ці технічні та організаційні заходи разом формують надійну архітектуру безпеки: конфіденційні дані не передаються у відкритому вигляді, доступ контролюється токенами з обмеженим терміном дії, привілеї мінімізовані, а системні журнали дозволяють швидко виявляти та локалізувати інциденти без порушення приватності вимірювань [6; 7].

Висновок до Розділу 1

У розділі розглянуто, як науковці та різні нормативи підходять до моніторингу якості повітря в приміщеннях (IAQ) та мікроклімату. Щоб рішення було практичним, воно має відповідати трьом вимогам: точні вимірювання (правильне налаштування приладів, контроль їхньої роботи з часом, вимірювання в різних точках приміщення), достатньо даних за різні проміжки часу (від коротких стрибків до змін протягом сезону) та зрозумілий аналіз для звичайних користувачів (час у межах норми, площі перевищень, інформація про небезпечні комбінації на зразок спека+сухість чи довготривала вологість).

Стандарти визначають, які показники вважаються нормальними. Вони враховують комфорт температури (PMV/PPD), потік повітря та цільові рівні IAQ, і можуть допомогти встановити правильні пороги для дому чи навчальних закладів.

ESP32 має достатньо ресурсів для фільтрації даних, відображення показників на місці та зберігання інформації. Найкраще використовувати трирівневу систему: пристрій вимірювання → Web API (REST) → мобільний додаток. У такому випадку BLE використовується лише для першого налаштування, а дані передаються через Wi-Fi/HTTPS.

Важливо, щоб безпека та конфіденційність були вбудовані в систему з самого початку. Це досягається за допомогою коротких access-токенів (JWT) з чіткими ролями власник/гість, TLS-шифрування, обмеження прав доступу та перевірок. На пристроях використовуються методи шифрування секретів, захист прошивки/Flash та чіткі процедури налаштування. Усі ці заходи разом забезпечують контроль, можливість розширення та відтворюваність результатів, що необхідно для подальшого розвитку, тестування та експериментів.

РОЗДІЛ 2 ПРОЕКТУВАННЯ ПРОГРАМНО-АПАРАТНОГО КОМПЛЕКСУ SENSEDATA

2.1 Постановка задачі та вимоги

Система SenseData розроблена для безперервного та точного вимірювання мікроклімату в житлових, освітніх та офісних приміщеннях, з подальшим перетворенням цих даних у корисні поради для користувача. Основна ідея полягає в тому, щоб перейти від простих показників до розуміння контекстуальних ситуацій і, на основі цього, пропонувати рекомендації, які враховують чіткі правила та поточні погодні умови. Рішення складається з таких частин: периферійні пристрої на базі ESP32 (для збору даних і передачі телеметрії), серверний компонент з REST/Web API (для отримання, зберігання та обробки даних) і мобільний додаток для Android (для первинного налаштування пристроїв, візуалізації даних та управління) [6–8].

На даному етапі не передбачено інтеграцію з системами опалення, вентиляції та кондиціонування повітря (HVAC) або системами розумний дім для безпосереднього керування, а також складні алгоритми машинного навчання. Проте архітектура системи дозволяє розширювати її функціональність у майбутньому без зміни основних принципів роботи.

Система має забезпечувати безпечне отримання та довготривале зберігання даних про температуру, вологість, тиск, наявність диму або легкозаймистих рідин, а також рівень освітленості. Важливою є можливість перегляду історії змін за будь-який період, швидке отримання інформації про поточний стан, порівняння даних між кімнатами та надання корисних порад. Кожна порада повинна містити пояснення причини (який саме показник перевищено та як довго), очікуваний результат (зменшення часу перебування поза межами норми) та прогноз щодо повернення до нормальних умов за умови звичайного провітрювання [1-2; 9-10].

Система призначена для використання в декількох кімнатах одного об'єкта. Кожна кімната має власні налаштування, прив'язку до датчиків і, при потребі,

геолокацію для врахування погодних умов. Один користувач може мати декілька об'єктів. Це враховано в моделі даних та правилах доступу, щоб уникнути дублювання інформації та несанкціонованого розповсюдження прав доступу.

Взаємодія компонентів системи відбувається наступним чином: користувач встановлює додаток, створює об'єкт і кімнати, задає потрібні параметри, а потім налаштовує датчики. Під час налаштування на датчик передаються дані Wi-Fi та інформація про кімнату. Після підключення датчик періодично надсилає дані телеметрії з мітками часу. Сервер отримує ці дані, перевіряє їхню цілісність і порядок, зберігає у базі даних та готує узагальнені дані для швидкого доступу з додатку. Мобільний додаток показує актуальні дані та історію змін, враховує погодні умови, визначає поточні ситуації (наприклад, перегрів і сухість) та надає відповідні поради.

З цього випливають такі вимоги до функціональності: підтримка моделі об'єкт → кімнати → датчики з гнучкими налаштуваннями та чітким визначенням власника датчика; реєстрація датчика за унікальним ідентифікатором; відображення статусу датчика; коректне отримання даних телеметрії з відновленням порядку після збоїв; формування графіків даних за різні періоди; сумісність відповідей із кешуванням. Інтерфейс повинен забезпечувати швидкий перехід між кімнатами, відображення налаштувань на графіках, позначення подій та зрозумілі підказки без зайвих повідомлень. Також передбачено спільний доступ: власник може надати гостьовий доступ до окремих кімнат або датчиків з можливістю його скасування в будь-який момент. Гість має право лише переглядати дані, не змінюючи налаштування та не надаючи доступ іншим.

Вимоги до продуктивності, надійності та тривалості роботи системи визначаються як швидко, надійно, довго. Сервер повинен стабільно обробляти великий потік даних телеметрії, зберігаючи швидкість відповідей на запити в межах сотень мілісекунд, навіть при фільтрації даних. Система повинна забезпечувати довготривале зберігання даних з можливістю їх узагальнення та обробки без втрати первинної інформації [15; 16]. Важливо, щоб короткочасні

перебої в роботі не впливали на цілісність даних: датчики зберігають дані локально і передають їх на сервер після відновлення зв'язку. Сервер повинен обробляти повторні передачі даних, уникаючи дублювання. Клієнтський додаток повинен швидко запускатися завдяки локальному кешуванню, відображати плавні графіки та інформативні повідомлення про стан системи.

Надійність і зручність підтримки системи забезпечуються стабільними правилами та можливістю спостереження за її роботою. API повинен мати версії та детальний опис з прикладами запитів. Усі компоненти системи генерують ідентифікатори, що дозволяють відстежувати шлях проходження даних від датчика до клієнта. Система реєструє показники успішності передачі даних, затримки, час перебування поза межами норми, частоту спрацювання правил і виконання рекомендацій. Підтримка системи передбачає можливість додавання нових датчиків і параметрів без зміни основних компонентів.

Якість вимірювань і обробки даних контролюється на всіх рівнях. Датчики забезпечують точний час опитування, правильний режим вимірювання, відфільтровують випадкові помилки та реєструють службові події (перезапуски, зміни налаштувань, калібрування). Це дозволяє відрізнити реальні зміни в навколишньому середовищі від технічних проблем. На сервері перевіряється послідовність міток часу, відхиляються застарілі дані, а узагальнення даних виконується за визначеною методикою. Інтервали з великою кількістю пропусків або повторів даних позначаються відповідними індикаторами. Логіка надання рекомендацій враховує тривалість перебування поза межами норми та погодні умови, щоб зменшити кількість непотрібних порад.

Безпека забезпечується на кожному етапі роботи системи. Первинне налаштування виконується локально з шифруванням важливих параметрів. Дані телеметрії та запити на управління передаються через захищені канали зі строгою перевіркою сертифікатів. Для авторизації використовуються короткочасні та довготривалі токени доступу; кожна операція супроводжується перевіркою прав доступу. Гостьовий доступ може бути скасований в будь-який момент. Операційна

безпека включає обмеження на кількість запитів до важливих компонентів системи та аудит доступу до даних [21; 22].

Основні вимоги до системи можна поділити на такі категорії:

- Функціональні:
  - Ієрархія об'єкт → кімнати → датчики (ESP32) з прив'язкою до власника.
  - Налаштування через BLE з використанням chipId.
  - Передача даних телеметрії пакетами з мітками часу.
  - Перегляд поточного стану, графіків, подій і порад.
  - Врахування погодних умов (OWM).
  - Гостьовий доступ.
- Нефункціональні:
  - Швидкість відповідей API в межах 300-500 мс для типових запитів.
  - Офлайн-буферизація даних, повторні спроби передачі з експоненційним збільшенням інтервалу, ідемпотентний прийом даних.
  - Синхронізація часу за допомогою NTP, перевірка послідовності міток часу.
  - Масштабовані сервіси та бази даних.
  - Кешування ETag/If-None-Match.
  - Метрики, трасування та логування.
  - Правила управління очищенням даних.
- Безпека:
  - Шифрування AES під час налаштування через BLE.
  - Використання TLS для всіх API-запитів.
  - Використання JWT (access/refresh токени), ролі owner/guest.
  - Миттєве скасування гостьових прав (revoke).
  - Обмеження на кількість запитів, захист від спаму та brute-force атак.
  - Аудит дій; відсутність зберігання секретної інформації у відкритому вигляді.

Для повноцінного використання системи важливо враховувати доступність і зручність для користувача. Доступність означає, що перебої зі зв'язком не призводять до втрати даних: датчики зберігають дані локально, сервер приймає запізнілі пакети з перевіркою порядку, а клієнт показує останні достовірні дані з позначенням їх часу і уникає помилкових тривог. Зручність використання включає швидке завантаження даних, плавні графіки, зрозумілі пояснення щодо запропонованих дій і їх очікуваного ефекту, передбачуваність порогів і стабільність поведінки системи в схожих ситуаціях. Для цього відповіді сервера містять пояснення, а клієнт використовує зрозумілі візуальні позначення, що дозволяє користувачеві бачити зв'язок між своїми діями і змінами на графіках.

Важливо, щоб вимоги до системи дозволяли масштабування за кількістю користувачів і датчиків без необхідності переробки архітектури. Це досягається завдяки чіткому розмежуванню компонентів, використанню стабільних мінімальних правил, агрегуванню даних різного рівня деталізації, кешуванню на всіх рівнях, прозорій телеметрії самої системи і можливості винесення окремих функцій (погодні умови, генерація рекомендацій, обробка великих обсягів даних) в окремі сервіси. Завдяки цьому SenseData може використовуватись як для навчання і досліджень, так і для реальної експлуатації в різних приміщеннях зі збільшенням кількості кімнат, датчиків і користувачів.

2.2 Архітектура рішення SenseData

2.2.1 Взаємодія ESP32 – Web API – SQL Server – Android

Система включає чотири основні частини:

- ESP32-вузол;
- Web API;
- SQL Server;
- Android-клієнт;

Між ними є цикл: вимірювання → отримання → збереження → показ/поради, який повторюється з різною частотою.

Компоненти системи:

- ESP32 бере дані з датчиків, об'єднує їх у JSON-пакет, додає chipId і відправляє на сервер. Також вузол іноді запитує у сервера інформацію (назву кімнати, зображення, ім'я користувача) і показує її на екрані.
- Web API – це основний вхід. Він перевіряє вузол за допомогою X-Api-Key, дивиться, чи правильні дані, зберігає їх у базі даних і робить зведені дані/рекомендації для користувачів. Для економії трафіку використовується кешування ETag і різні версії контрактів DTO [17; 18].
- SQL Server зберігає дані SensorData, інформацію про власність, дані користувачів і налаштування. Індекси по ChipId і Timestamp дозволяють швидко знаходити дані для графіків [15; 16].
- Android-клієнт отримує дані через HTTPS-запити (JWT), показує історію вимірювань, поточний стан кімнат і текстові поради. Для надійності використовується кешування (ETag) і локальний кеш [23–25].

Детальніше про взаємодію основних частин показано на Рис 2.1

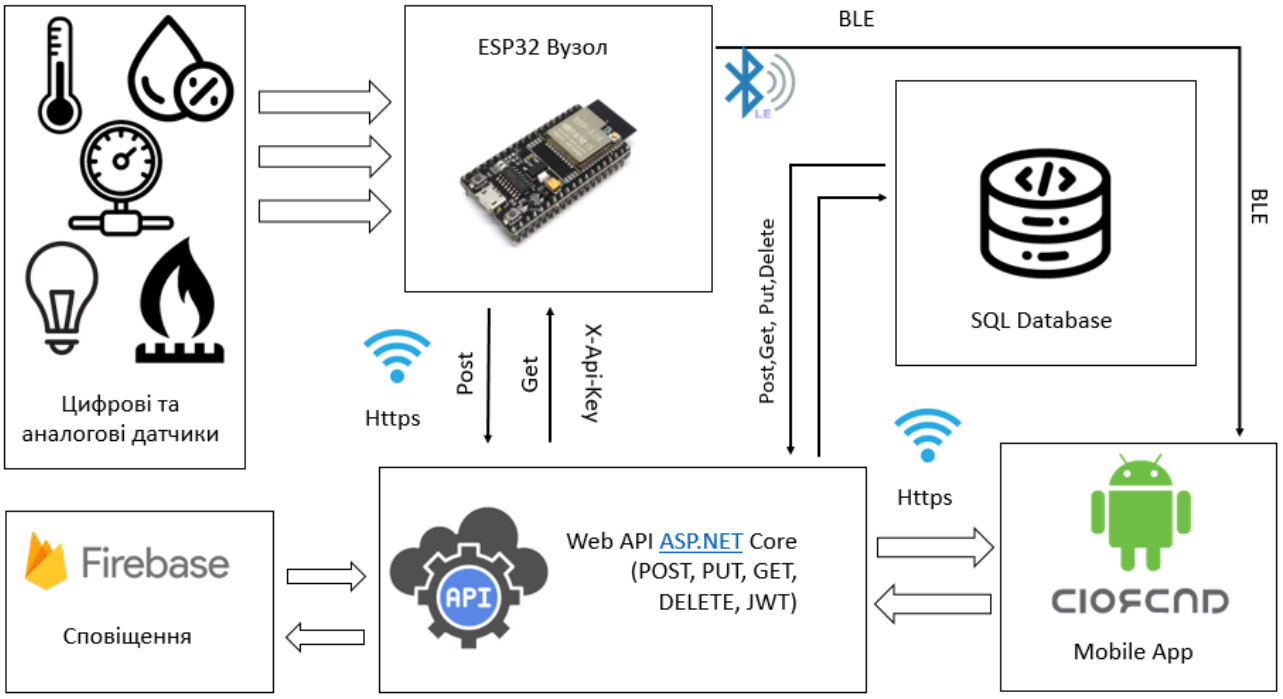

Рисунок 2.1 – Графічне приставлення взаємодії додатку SenseData

2.2.2 Структурна схема вузла ESP32

Вузол SenseData — автономний вимірювач мікроклімату на базі **ESP32-WROOM** із виносною сенсорною зоною, індикацією на **LCD 20×4 (I²C)**, сервісним **BLE-провіжингом**, періодичною телеметрією **HTTP/JSON** та локальним збереженням конфігурації в **NVS (Preferences)** [9; 10]. Конструкція і прошивка побудовані так, аби рознести «гарячі» компоненти (MQ-2, стабілізатор, радіомодуль) від чутливого датчика вологості/температури, дотриматися стабільних таймінгів опитування й забезпечити відновлення роботи після збоїв без втручання користувача.

Склад апаратного вузла

Основу пристрою становлять такі компоненти:

- **ESP32-WROOM-32** — головний обчислювальний модуль з двоядерним процесором Tensilica LX6 (Рис 2.2), що підтримує бездротові інтерфейси Wi-Fi (2.4 ГГц) і BLE [12; 13].

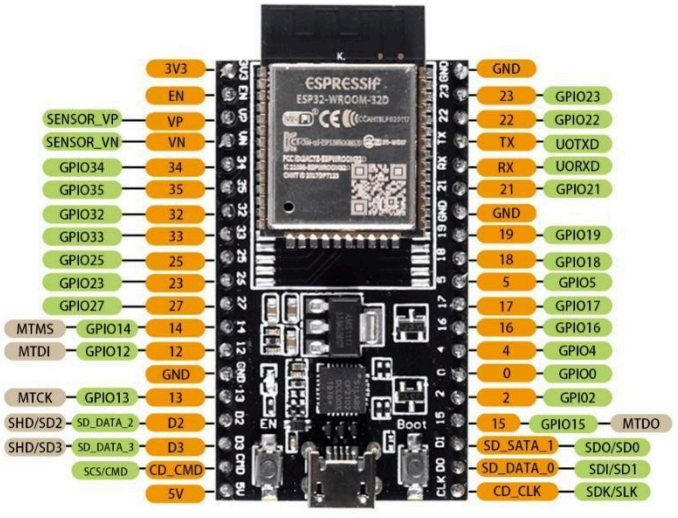

Рисунок 2.2 – Архітектура Esp32 з основними пінами[34]

Мікроконтролер забезпечує роботу вього вузла, обмін даними з сенсорами, збереження налаштувань у пам’яті NVS і комунікацію з сервером. Має вай-фай та блютуз вже вбудований у сам чіп, що є великою перевагою над тим самим Arduino.

- **BME280** — високоточний сенсор, який вимірює температуру, відносну вологість і атмосферний тиск. Підключається по шині **I²C** (SDA — GPIO 21, SCL — GPIO 22)(Рис 2.3).[14].

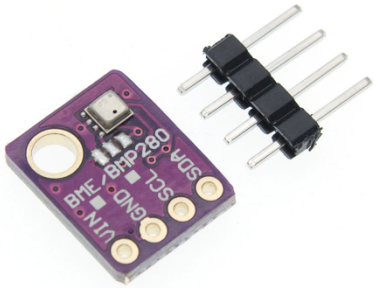

Рисунок 2.3 – Датчик вологості, температури та тиску BME280[35]

- **DHT11** — другий датчик температури та вологості(Рис 2.4), що з'єднується з ESP32 через однопровідний інтерфейс (GPIO 4). Служить як підстраховка до BME280, так як система може працювати навіть коли один датчик вийшов з ладу.[14].

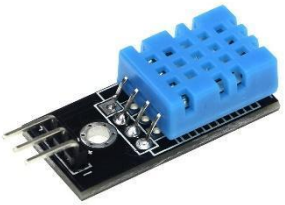

Рисунок 2.4 – Датчик вологості та температури DHT11[35]

- **MQ-2** — газовий сенсор, що реагує на дим, чадний газ і леткі органічні сполуки (Рис 2.5). Підключений до аналогового входу **ADC1_CH6 (GPIO 34)**. Для фільтрації перешкод застосовано RC-ланку. Після ввімкнення потребує прогріву (~20 секунд), точна калібровка займає до 24 годин [12;14].

Рисунок 2.5 – Датчик газу MQ-2[35]

- **LDR (Light Dependent Resistor)** — фоторезистор (Рис 2.6), підключений до **ADC1_CH7 (GPIO 35)** через подільник напруги. Використовується для визначення рівня освітленості приміщення (день/ніч) і для контекстного аналізу мікроклімату [14].

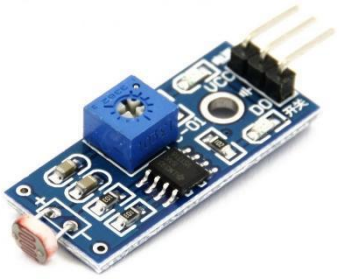

Рисунок 2.6 – модуль світла (LDR)[35]

- **Smoke_PIN (GPIO 25)** та **Light_PIN (GPIO 26)** — цифрові входи, що фіксують стан дискретних датчиків диму або освітлення. Значення LOW трактується як подія.
- **LCD 20×4 (I²C, адреса 0x27)** — рідкокристалічний дисплей на базі контролера PCF8574, який відображає температуру, вологість, тиск, рівень освітленості, стан Wi-Fi, BLE і API (Рис 2.7) [14].

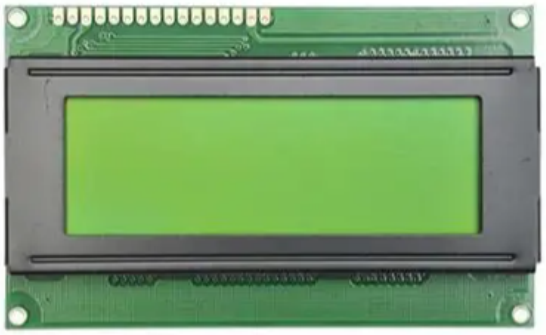

Рисунок 2.7 – LCD 20x4 I2C[35]

- **Живлення 5 В** — здійснюється від USB або зовнішнього адаптера через перетворювач типу **Buck**, який знижує напругу до 3.3 В. Додатково використовується LC-фільтр для зменшення шумів та захист від перенапруги.

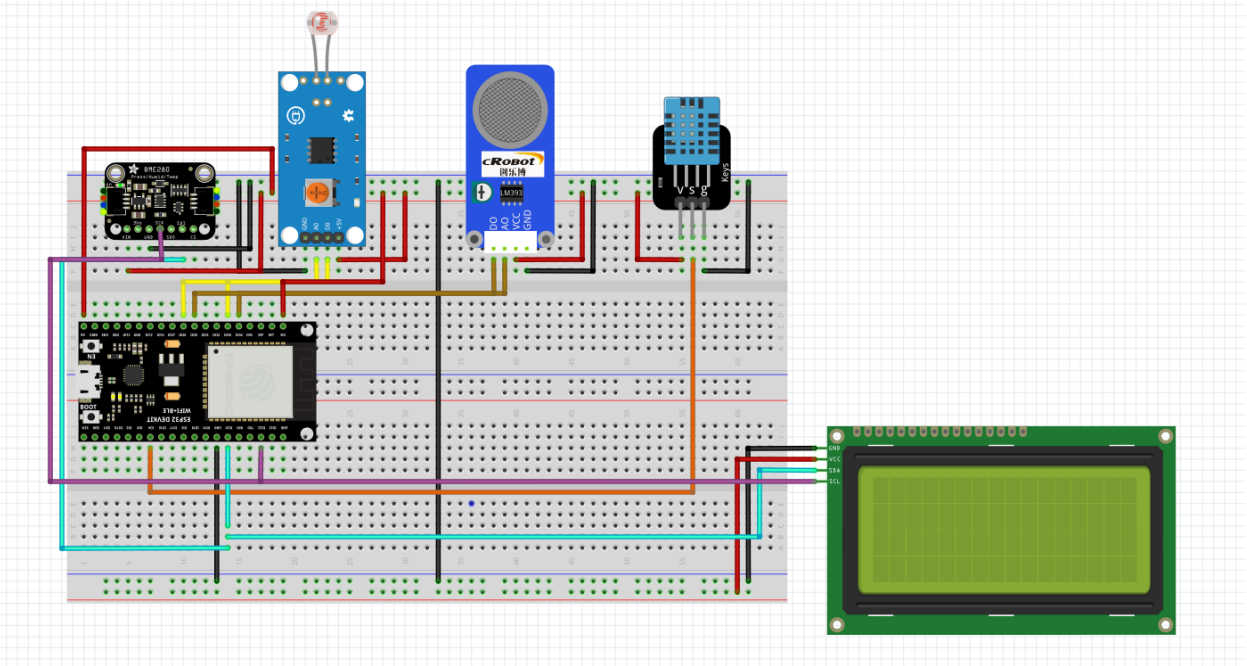

Рисунок 2.8 – Схема підключень вузла ESP32 в середовищі Frizing

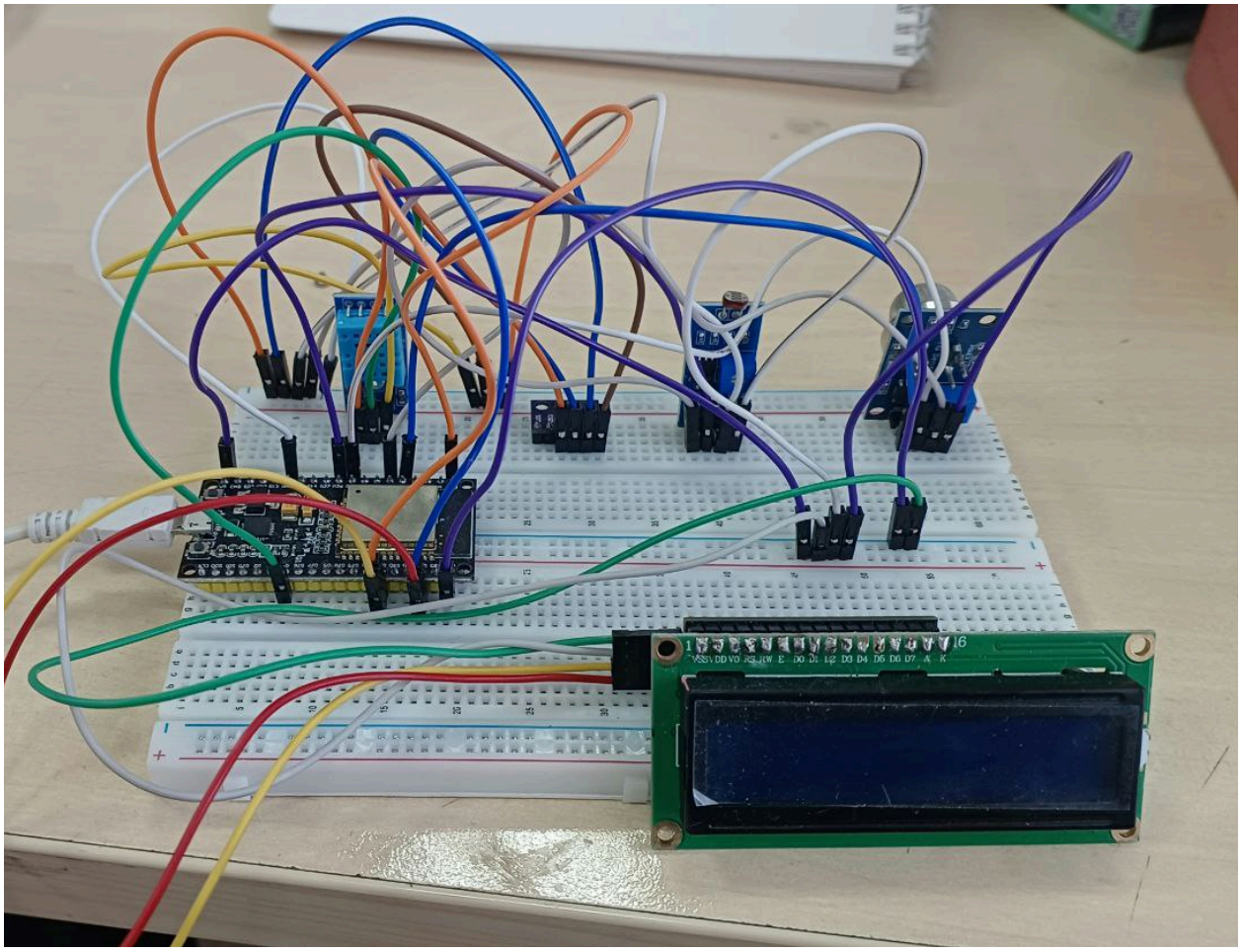

Рисунок 2.9 – Прототип вузла ESP32 реалізований на макетній безпаяльній платі

Детальна схема та прототип вузла ESP32 показано на рис 2.8-2.9.

У результаті така структура дозволяє поєднати високу автономність, точність вимірювань і безпеку передачі даних.

ESP32 виступає центральним елементом вузла, виконуючи роль «мозку», логіку взаємодії з мобільним клієнтом і сервером через стандартні протоколи Wi-Fi та BLE [12; 13].

2.2.3 Сервер: архітектура, безпека, ендпоінти та контракти

Серверний компонент реалізовано як ASP.NET Core (.NET 9) Web API із класичною багатошаровою декомпозицією [17; 18]:

- Controllers — приймання/видача, маршрути, статус-коди, мінімальна оркестрація (див. Додаток Б).

- Services — доменна логіка: валідація й нормалізація телеметрії, обчислення агрегатів, формування порад, застосування політик доступу (див. Додаток В).
- Repositories (EF Core) — параметризовані INSERT/SELECT, робота зі схемою БД, індекси по ChipId, RoomId, UserId, Timestamp (див. Додаток Г) [19; 20].

Детальна взаємодія шарів серверу показано на Рис 2.10

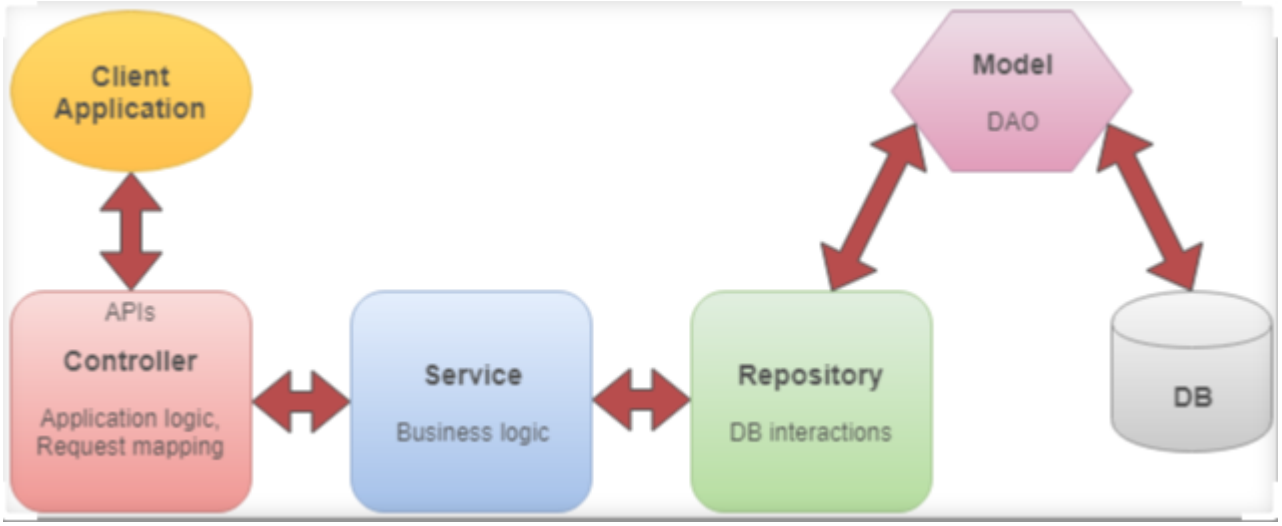

Рисунок 2.10 – Схема взаємодії шарів всередині сервера SenseData[36]

Канал для периферії (ESP32) ізольовано через X-Api-Key; клієнтський доступ відбувається через HTTPS + JWT (ролі owner / guest). Читання оптимізуються умовним кешуванням ETag/If-None-Match.

Модель безпеки:

- Автентифікація: короткоживучі access-токени та довгоживучі refresh-токени (JWT). Ротація refresh при кожному оновленні; відкликані токени зберігаються гешовано.
- Авторизація: політики ролей та приналежності ресурсів (будь-який доступ до кімнати/вузла перевіряє власника; гість має read-only у заданому scope).
- Канал ESP32: X-Api-Key для шлюзу телеметрії; можливий додатковий HMAC-підпис батчів (за конфігурацією) [12; 13].

- Транспорт: лише HTTPS; жорсткі таймаути; повтори запитів до ідемпотентних ендпоінтів не змінюють стан.
- Зменшення навантаження/ризиків: rate limiting на /auth та /sensordata; обмеження розміру батчу; валідація часових міток і монотонності [19–21].

Основні групи ендпоінтів (Рис 2.11)

1) SensorData (канал ESP32, X-Api-Key)

➢ POST /api/sensordata — приймання батчу телеметрії (JSON).
Відповіді: **201** (успіх, batchId), **400** (валідація), **401/403** (ключ), **409** (дубль, ідемпотентність).

➢ GET /api/sensordata/sync/{chipId} — мінімальні метадані вузла для ESP32 (room, image, username) з **ETag → 200** або **304**.

➢ GET /api/sensordata/bychip/{chipId}/day|week — агреговані дані вузла для діагностики.

2) DisplayData (Android, JWT)

➢ GET /api/displaydata/byuser — список кімнат користувача з «останнім зрізом», підтримка **ETag**.

➢ GET /api/displaydata/ownership/{chipId}/{latest|day|week} — дані конкретного вузла для UI.

➢ POST /api/displaydata/ownership — створення кімнати / прив’язки вузла (**201**, **409** при зайнятому chipId).

➢ PUT/DELETE /api/displaydata/ownership — оновлення / видалення кімнати (перевірка прав).

➢ POST /api/displaydata/guest — створення гостьового доступу;

➢ GET/DELETE /api/displaydata/guest/{chipId} — перегляд / відкликання доступу.

➢ POST /api/displaydata/join — прийняття інвайту гостем.

3) Settings / Advice (JWT, owner)

➢ GET /api/settings/advice/{chipId}/latest — актуальні пороги/правила (з **ETag**).

➢ POST /api/settings/adjustment/{chipId} — користувацькі корекції порогів і «тихих годин».

➢ GET /api/settings/effective/{chipId} — обчислені ефективні налаштування.

4) Users / Auth

➢ POST /api/users/register | login | refresh | logout — стандартні потоки, JWT.

➢ GET/PUT /api/users/profile — профіль користувача.

5) Alerts

➢ POST /api/alerts/gas — реєстрація подій «газ/дим» для пуш-сповіщень (**202 Accepted**).

Основні DTO

Charts / Display

➢ SensorPointDto — точка графіка: час t + значення v.

➢ RoomWithSensorDto — кімната + останні значення каналів, пороги, статуси, події.

Recommendations

➢ RecommendationsDto — актуальні поради (текст, причина, пріоритет).

➢ RecommendationHistoryDto — історія виданих порад.

Ownership / Telemetry

➢ SensorDataInDto — вхідна телеметрія від ESP32 (ChipId, T/RH, Pressure, GasDetected, Light, MQ2Analog, LightAnalog, Timestamp).

➢ SensorDataDto — нормалізований запис у БД (дані + createdAt).

➢ SensorOwnershipDto — відповідність ChipId ↔ owner/room.

➢ OwnershipSyncDto — мінімальні метадані для /sync/{chipId}.

➢ AddGuestRequestDto — створення гостьового доступу.

Settings

➢ EffectiveSettingDto — підсумкові пороги, гістерезис, «тихі години».

➢ AdjustmentAbsoluteDto — явні зміни порогів з коментарями (аудит).

Users / Push

- RegisterRequestDto, LoginRequestDto, UserProfileDto, GuestDto — базові структури користувачів/гостей.
- GasAlertState, GasTelemetryDto — стан та події тригера «газ/дим».

Кешування, ETag і інвалідація

➢ Умовний кеш для «останніх зрізів», налаштувань і /sync: сервер повертає ETag; клієнт надсилає If-None-Match; при відсутності змін — 304 Not Modified без тіла.

➢ Короткий серверний кеш агрегатів (ключ: userId + roomId + range + bucket). Запис телеметрії адресно інвалідовує дотичні ключі.

➢ Відповіді для великих діапазонів — зі вказаним Cache-Control (короткий TTL) + варіації за параметрами (Vary).

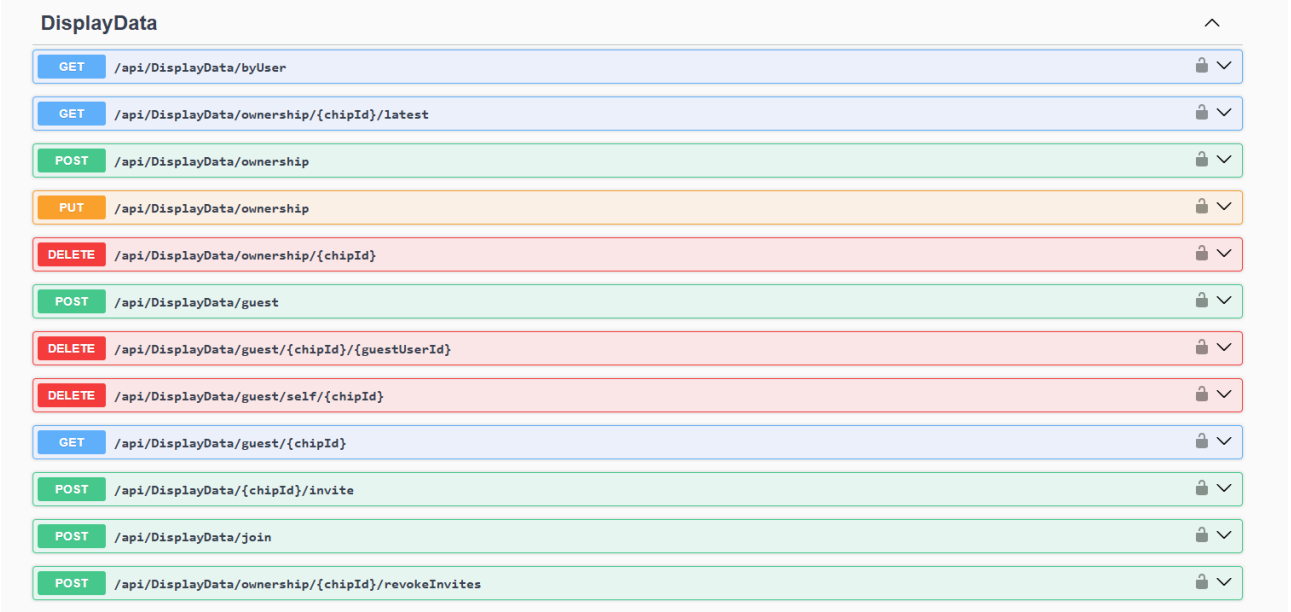

Рисунок 2.11 – Список ендпоінтів SenseData контроллера DisplayData

Серверна частина забезпечує:

- **чітке розділення відповідальностей** (Controllers/Services/Repositories),
- **прозорі контракти DTO** з умовним кешем і пагінацією,
- **двоканальну безпеку** (ApiKey для ESP32, JWT для клієнта) з миттєвим відкликанням гостьових прав,
- **ефективне читання** через ETag і короткий кеш агрегатів,
- **надійне приймання телеметрії** з ідемпотентністю та валідацією часової послідовності, — що в сукупності дає стабільну основу для візуалізації, рекомендацій і масштабування системи SenseData [17–22].

2.2.4 База даних

Сховище телеметрії та довідкових сутностей реалізовано у **SQL Server** як реляційна модель із чітким розведенням часових рядів і довідників. Нормалізація до 3НФ дає змогу зберігати **детальні сирі записи** сенсорів окремо від **власності/доступів** та **налаштувань порогів**, водночас забезпечуючи ефективні

вибірки за часовими інтервалами [15; 16]. Зв’язки і ключі подано на ER-діаграмі(Рис 2.13).

Основні таблиці:

- **SensorData** — журнал сирих вимірювань.
  - Поля (основні): ChipId, показники T/RH/p (BME/DHT), індикатори подій (GasDetected, Light), аналогові канали (Mq2Analog, LightAnalog та їхні відсотки), CreatedAt (UTC).
  - Ключі та індекси: Id (PK); складений індекс по **(ChipId, CreatedAt)** та окремий по CreatedAt для швидких вибірок **latest/day/week** і побудови агрегатів.
  - Призначення: зберігання **щільних рядів** для недавніх періодів і побудова **розріджених агрегатів** на рівні API [15].
- **SensorOwnership** — прив’язка вузла до власника і кімнати.
  - Поля: OwnerId (FK → Users), **унікальний** ChipId, RoomName, ImageName, Version, UpdatedAt/CreatedAt.
  - Обмеження: **унікальність ChipId** гарантує схему «1 плата — 1 власник». Пара Version/UpdatedAt використовується для обчислення ETag у синхронізаційних ендпоінтах [17; 18].
  - Призначення: доменний «контейнер» для відображення кімнат і керування доступами.
- **SensorGuests** — гостьові права доступу до вузла.
  - Поля: ChipId (FK → SensorOwnership за **логічним ключем** ChipId), UserId (FK → Users), CreatedAt.
  - Обмеження: **унікальна пара (ChipId, UserId)** запобігає дублюванню гостя.

- Поведінка: каскадне видалення при видаленні прив’язки власника (SensorOwnership).

➢ **Users** — облікові записи користувачів.

- Поля: Username, Email (обидва **унікальні**), RoleId (FK → Roles), PasswordHash, пари для сесій (RefreshTokenHash, RefreshTokenExpiryTime), Version, CreatedAt/UpdatedAt.
- Призначення: автентифікація/авторизація, аудит дій [17; 18].

➢ **Settings** — базові пороги/повідомлення для параметрів (температура, вологість, «газ/дим»).

- Поля: ParameterName (**унікальний**), LowValue/HighValue, повідомлення для низьких/високих значень.
- Призначення: єдине джерело «дефолтних» правил формування порад.

Допоміжні таблиці:

➢ **SettingsUserAdjustments** (коригування порогів користувачем і/або на рівні вузла з версіонуванням записів)
➢ **ComfortRecommendations** (зафіксовані рекомендації з посиланням на джерельний SensorData)
➢ **GasAlertStates** (поточний стан тригера «газ/дим», PK = ChipId)
➢ **GuestInvites** (одноразові інвайти з токеном і терміном дії)
➢ **Roles** (довідник ролей).

Усі часові поля ведуться в **UTC** (CreatedAt/UpdatedAt з GETUTCDATE()), що спрощує агрегування та коректну побудову часових інтервалів [15; 16].

Поля Version/UpdatedAt у **SensorOwnership** та версійні ключі в **SettingsUserAdjustments** забезпечують **ETag/If-None-Match** і відтворюваність історії конфігурацій.

Основне навантаження лягає на **SensorData**; складений індекс (ChipId, CreatedAt) мінімізує час виконання фільтрів «за вузлом і вікном часу» та підготовку агрегатів , що важливо для телеметрії IoT [15].

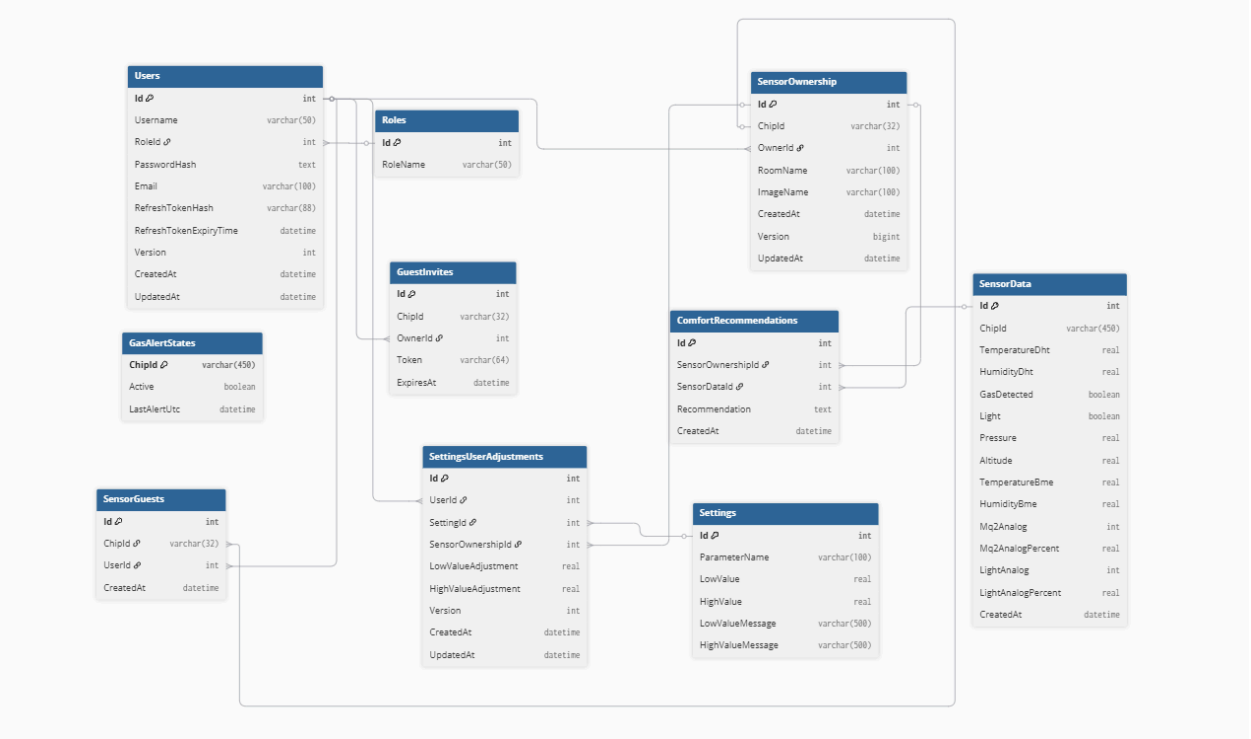

Рисунок 2.13 – Архітектура сховища телеметрії SenseData

2.2.4 Мобільний клієнт Android

Архітектурні принципи (MVVM)

Клієнт розроблено за моделлю MVVM, де чітко розділено обов'язки [23; 24]:

- View (Activity/Fragment) — відповідає лише за показ станів (завантаження, відсутність даних, помилка, вміст) і реагує на дії користувача; мережева взаємодія відсутня.
- ViewModel — збирає дані з репозиторіїв, регулює стан екранів (LiveData/State), виконує прості перетворення даних (форматує дати/одиниці виміру).
- Repository — єдиний шлях доступу до даних; об'єднує віддалені джерела (REST/Web API) та локальне зберігання, визначає стратегію оновлення і об'єднання даних.

- Data Sources — RemoteDataSource (Retrofit) і LocalDataSource (Room).
- Mapper/DTO — використовує явні перетворення транспортних DTO у внутрішні доменні моделі для UI, щоб ізолювати рівень представлення від змін API [23].

Для ін'єкції залежностей використано просту DI-конфігурацію (модулі для OkHttpClient, Retrofit, репозиторіїв, Room). Керування життєвим циклом погоджено з компонентами Android (скасування запитів у onStop()/onCleared()).

Мережева інфраструктура: Retrofit + OkHttp [23; 24]:

- AuthInterceptor — додає заголовок Authorization: Bearer до кожного запиту користувача; для службових запитів (наприклад, BLE-реєстрації) заголовок не додається.
- TokenAuthenticator — обробляє помилки 401 Unauthorized: використовує єдину потокобезпечну чергу оновлення токенів (refresh), з ротацією refresh-токена та повторним виконанням початкового запиту один раз. У разі невдачі – вихід із сесії (forced logout) та виведення зрозумілого повідомлення [22].
- ETag / If-None-Match — для ендпоінтів останній зріз, налаштування, /sync: клієнт зберігає отримані ETag у локальному сховищі; при наступних запитах надсилає If-None-Match та обробляє 304 Not Modified (оновлює тільки час актуальності даних).
- Повторні спроби і ліміт часу на очікування — короткий час очікування для зчитування, довший – для відправлення великих обсягів даних/подій; обмежена кількість повторних спроб зі збільшенням затримки тільки для операцій, які можна безпечно повторювати.

BLE-менеджер

- Права та безпека: перевірка дозволів (Bluetooth/Location), активація BLE, короткий термін доступу після скидання налаштувань вузла; передавання важливих даних у зашифрованому вигляді (сеансовий ключ) [12; 13].

- Сканування → вибір вузла → з'єднання: фільтрування за початком ідентифікатора/сервісу; ліміт часу на з'єднання; декілька спроб з поступовим збільшенням інтервалу.
- GATT-сесія: характеристики для roomName, imageName, ssid, encryptedPassword, username, reset; підтвердження отримання інформації (ACK) з боку вузла; відображення процесу для користувача (progress UI).
- Завершення: команда підтвердження, перевірка переходу на Wi-Fi, закриття GATT; контроль появи вузла у /sync/{chipId} (через Web API). У разі помилки – рекомендації з відновлення (перезавантаження вузла, повторна спроба).

UI/UX (див. Додаток Д).

- Головний екран — список кімнат з інформацією про останній зріз, індикаторами станів (в межах норми/поза межами), позначенням часу отримання даних.
- Екран кімнати — графіки T/RH/тиску з лініями лімітів, маркерами подій (MQ-2, анотації), накладання зовнішніх даних (за потреби). Перемикання інтервалів: останні дані / день / тиждень (серверна агрегація 1хв/5хв/15хв).
- Поради — блок із текстом, причиною спрацювання (який поріг і як довго перевищено), очікуваним результатом; пріоритетність та тихі години.
- Гості та доступи — перегляд активних прав гостей, створення/скасування запрошень (з підтвердженням дії).
- Станові екрани — послідовні стани: завантаження / відсутність даних / помилка; у випадку помилки – чіткі повідомлення з конкретними діями.

Відображення часових рядів (рис 2.14)

- MPAndroidChart: лінійні графіки з позначками порогів і зонами поза діапазоном (затемнення), маркери подій, жести масштабування/прокрутки, вікно для швидкої навігації [25].

- Ефективність: оновлення даних частинами (оновлюються тільки змінені ділянки), попередня обробка на сервері, стандартизований формат SensorPointDto (t, v) на узгодженій часовій сітці.
- Відображення осей: локальний часовий пояс, позначки часу протягом доби, виділення нічного періоду.

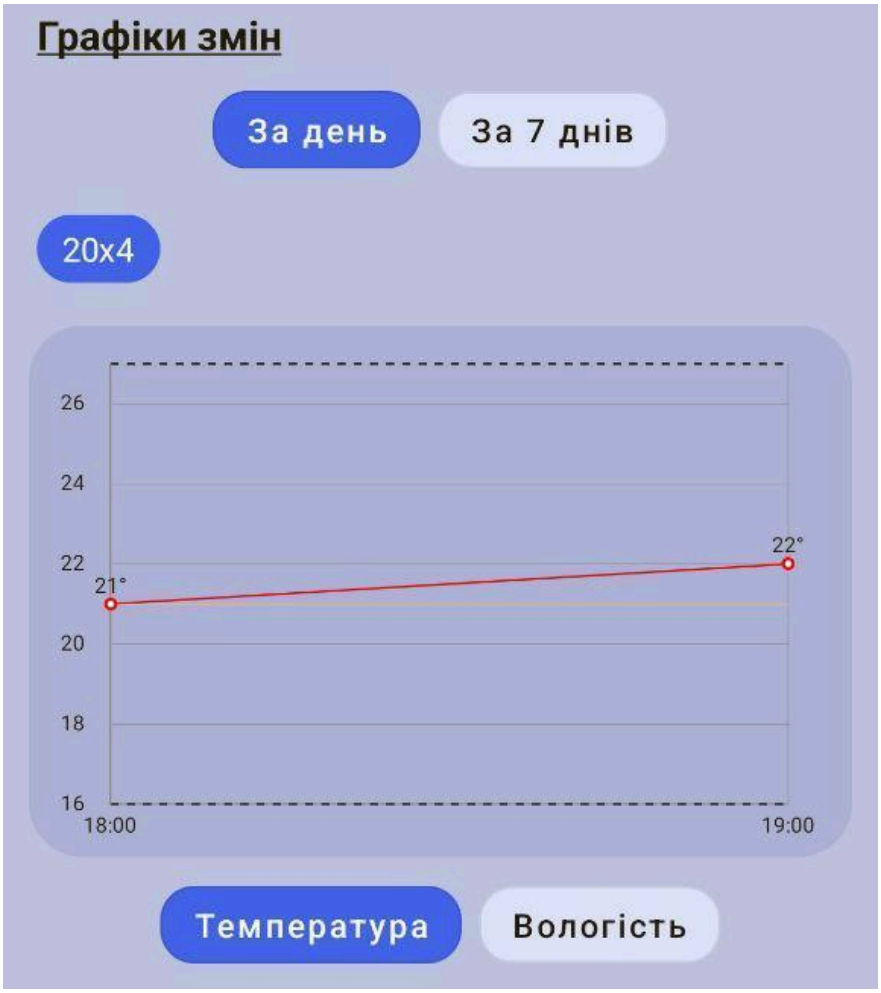

Рисунок 2.14 – Лінійні графіки з позначками MPAndroidChart в додатку SenseData

Обробка помилок і відновлення сесії

- 401/403 — автоматичне оновлення токенів (Authenticator), один повтор початкового запиту; при невдачі – вихід із сесії [22].
- 5xx/ліміт часу вичерпано — декілька повторних спроб для операцій, які не змінюють стан; перехід у режим роботи з кешем з інформаційним повідомленням дані актуальні на ….

- Перевірка даних — контроль відповідності даних реальності (наприклад, межі T/RH), захист від пропусків даних (відмітки про достовірність/повноту від сервера).

Сповіщення

- Локальні сповіщення про події газ/дим або тривалі відхилення RH/T (враховуючи тихі години); загальний центр сповіщень з можливістю вимкнення (Рис 2.15).
- Перехід зі сповіщення веде на відповідну кімнату з автоматичним відкриттям сторінки події.

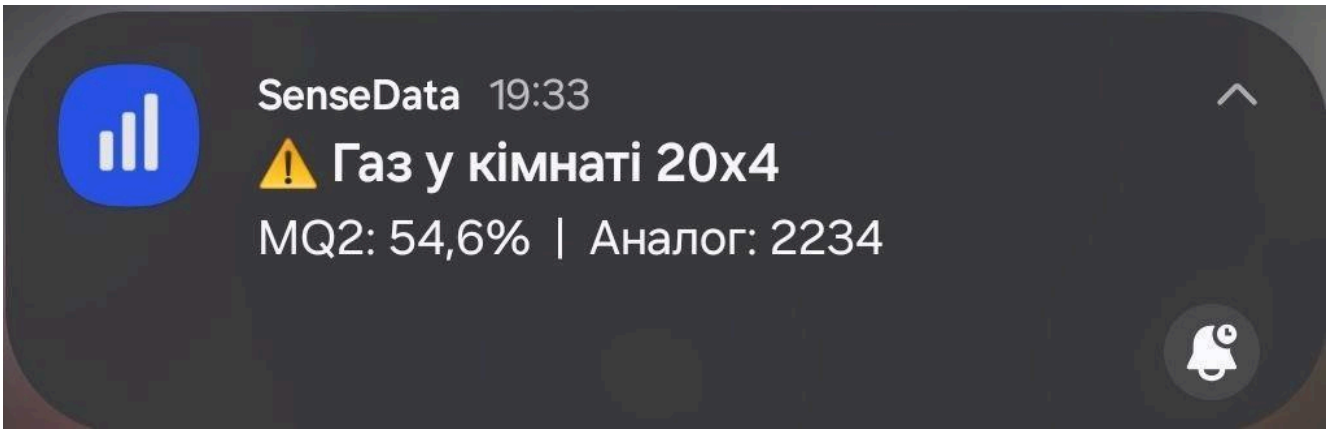

Рисунок 2.15 – Приклад сповіщення про задимлення в додатку SenseData

Запропонована реалізація Android-клієнта дозволяє швидко почати роботу з локального кешу, зменшує використання мережі за допомогою ETag/If-None-Match, стабільно працює при збоях (офлайн-режим + WorkManager), автоматично оновлює токени (Authenticator) та надає зрозумілі поради на основі оброблених даних із сервера [23–25].

2.3 Алгоритми

2.3.1 Алгоритм вузла ESP32

Код поділений на блоки, кожен з них має свій функціонал. Повний лістинг коду (Додаток А)

Блок А — setup()

- ➢ Формування ідентичності: з efuse MAC обчислюється ChipId та bleName = "ESP32_<id>".
- ➢ Підготовка інтерфейсів: запуск BLE (GATT-сервер, сервіс і характеристика WRITE/WRITE_NR/NOTIFY), I²C, LCD 20×4, DHT11, спроба ініціалізації BME280 (0x76), налаштування пінів MQ-2/Light.
- ➢ Прогрів MQ-2 ~20 с для стабілізації показів.
- ➢ Зчитування NVS (namespace: "config") ключа configured.
    - ➢ Якщо true — негайна спроба підключення до Wi-Fi з параметрів у NVS.
    - ➢ Якщо false — вузол залишається у режимі провізіонування (BLE-реклама активна).

Блок B — BLE-провізіонування (частковий JSON-patch)

- ➢ Обробка WRITE у характеристику: приймається JSON із будь-якою підмножиною полів
  ssid, password(AES-зашифрований), username, roomName, imageName, apiKey, reset.
- ➢ Якщо reset=true — повне очищення Preferences у config.
- ➢ Політика збереження — «патч»: у NVS записуються **лише присутні** поля; відсутні **не затираються**.
- ➢ Перевірка apiKey за довжиною (16–128); шифрований пароль зберігається як enc_pwd.
- ➢ Ознака готовності: configured=true, якщо є **обидва** ssid і enc_pwd.
- ➢ Після запису вузол відправляє NOTIFY із ChipId.
- ➢ Якщо надійшла валідна Wi-Fi-пара (ssid+password) — негайний WiFi.begin(); планується «ранній» sync через ~5 с.

Блок C — Керування Wi-Fi

- ➢ Підключення STA з очищенням попередніх станів; таймаут очікування ~15 с.
- ➢ Перевірка каналу раз на **60 с**: якщо configured=true і WL_CONNECTED==false — виконується reconnectWithSavedWifi() (читає ssid/enc_pwd із NVS та ініціює підключення).
- ➢ Успішне підключення розблоковує фонові задачі SYNC і POST.

Блок D — Синхронізація метаданих кімнати/власника (ETag)

- ➢ Виклик: GET /api/SensorData/sync/{chipId} із заголовками X-Api-Key та, за наявності, If-None-Match: <own_etag>.
- ➢ Обробка:
  - ➢ 200 OK — частково оновлюються username, roomName, imageName у NVS; зберігається новий ETag як own_etag; службовий прапор own_provisional=false.
  - ➢ 304 Not Modified — лише own_provisional=false, без змін полів.
  - ➢ Інші коди — вважаються помилкою синхронізації.
- ➢ Планування наступного SYNC:
  успіх — через **30 хв**; помилка — через **5 хв**.

Блок E — Телеметрія (періодичний POST)

- ➢ Період відправлення — **кожні 10 хв**.
- ➢ Знімаються покази з DHT11 і BME280; обробляються цифрові події (GasDetected, Light) та аналогові канали (MQ2Analog, LightAnalog) + нормовані відсотки.
- ➢ Непридатні значення (NaN) серіалізуються як null, аби спростити серверну валідацію.

- ➢ Виклик: POST /api/SensorData з заголовками Content-Type: application/json та X-Api-Key.
- ➢ Після успіху фіксується lastTime; пропущені інтервали **не** наздоганяються (буферизації батчів у цій версії немає).

Блок F — LCD-відображення

- ➢ Оновлення — **кожні 10 с**.
- ➢ На старті показуються статуси **BLE/Wi-Fi/API**; після стабілізації (BLE+Wi-Fi OK) переходить у режим показу вимірювань.
- ➢ Робочий екран: температура/вологість (DHT/BME), тиск (hPa), індикатори Gas/Light, roomName (обрізання/скрол за потреби).

Блок G — Зберігання конфігурації (NVS)

- ➢ namespace: "config", ключі:
  ssid, enc_pwd, api_key, username, roomName, imageName,
  configured (bool), own_etag, own_provisional (bool).
- ➢ Всі оновлення — **інкрементні**, у дусі JSON-patch: відсутні поля не модифікують попередні значення.

Блок H — Таймінги та розклад задач

- ➢ LCD_REFRESH_MS = **10 с** — оновлення індикації.
- ➢ WIFI_CHECK_MS = **60 с** — контроль каналу та реконект.
- ➢ POST_INTERVAL_MS = **10 хв** — період телеметрії.
- ➢ SYNC_OK_PERIOD_MS = **30 хв**, SYNC_FAIL_PERIOD_MS = **5 хв** — періоди повторів SYNC.
- ➢ «Ранній» SYNC — **~5 с** після прийому нової Wi-Fi-пари через BLE.

Детальний алгоритм роботи коду див. Рис 2.16

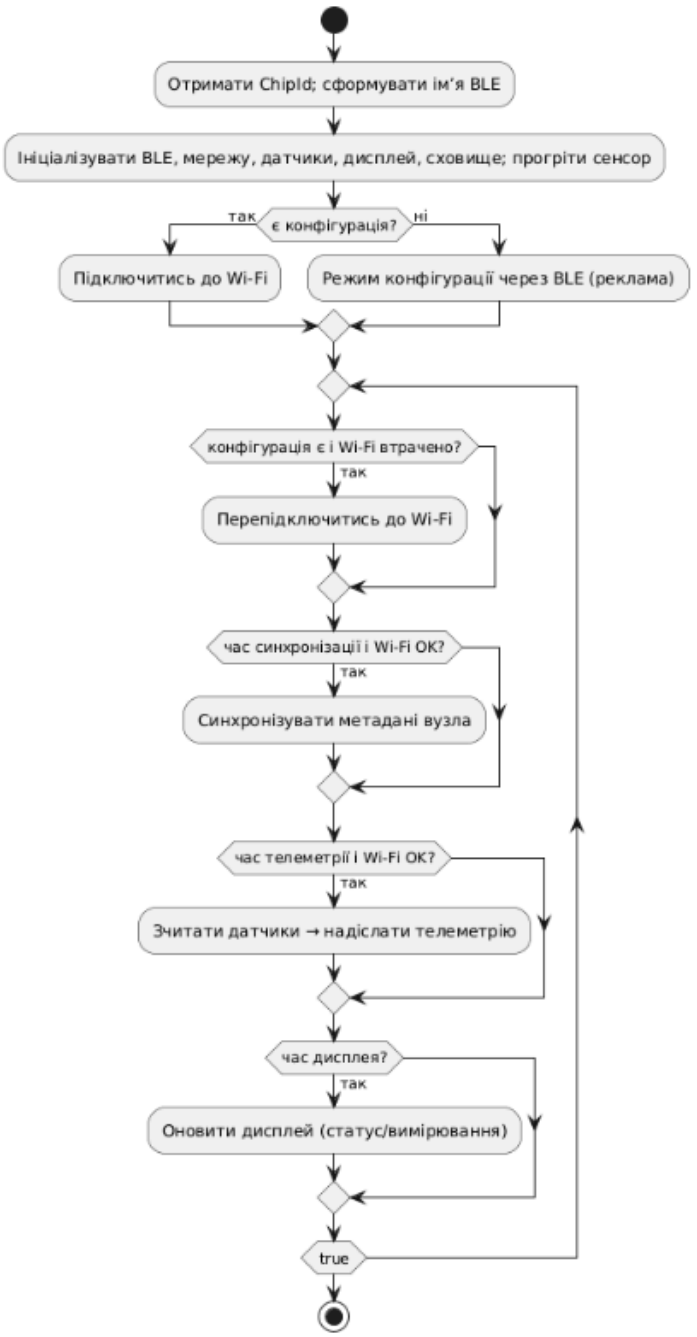

Рисунок 2.16 – Алгоритм роботи вузла ESP32

2.3.2 Алгоритм BLE-провізіонування

У системі **SenseData** початкове налаштування вузла ESP32 здійснюється через короткочасний бездротовий канал **Bluetooth Low Energy (BLE)**. Цей етап називається *провіжингом* — процесом передачі конфігураційних параметрів від мобільного застосунку до пристрою перед підключенням до Wi-Fi [9; 10].

Механізм BLE-провіжингу забезпечує безпечний, швидкий і безручний спосіб

ініціалізації вузла без потреби фізичного підключення до комп’ютера або ручного введення мережевих параметрів.

Провіжинг — це взаємодія двох сторін: **Android-додатка**, який формує JSON-конфігурацію, та **ESP32-вузла**, який приймає, зберігає і застосовує ці налаштування.

Процес має наступну логіку:

1. **Пошук пристрою**

   Після натискання кнопки «Додати кімнату» додаток починає BLE-сканування у межах найближчих пристроїв.

   Усі вузли рекламуються під іменами формату ESP32_xxxxxxxxxxxx, де суфікс xxxxxxxxxxxx відповідає 12 символам унікального ChipId.

2. **Встановлення з’єднання**

   Користувач обирає один із знайдених пристроїв. Android-додаток під’єднується до сервісу з UUID:

   SERVICE_UUID = "12345678-1234-1234-1234-123456789abc"

   CHARACTERISTIC_UUID = "abcd1234-5678-1234-5678-abcdef123456"

З’єднання встановлюється в режимі *write+notify* для двостороннього обміну короткими пакетами [12; 13].

3. **Формування JSON-пакета**

   На етапі створення кімнати користувач заповнює поля:

   - **roomName** — назва кімнати;
   - **imageName** — символічне зображення;
   - **ssid** — ім’я Wi-Fi мережі;
   - **password** — пароль Wi-Fi, який перед передачею шифрується алгоритмом **AES-128**;

- **reset** — логічне поле для повного очищення конфігурації вузла (true/false);

Приклад сформованого JSON-пакета:

{

"roomName": "LivingRoom",

"imageName": "living.png",

"ssid": "MyHomeWiFi",

"password": "U2FsdGVkX1+R4y9Bsnl0==",

"username": "misha",

"apiKey": "f68b1a6d-c2f8-4e3a-9b5d-723fae8e9324",

"reset": false

}

4. **Передача конфігурації через BLE**

   JSON надсилається як один write-пакет у характеристику BLE.

   Пристрій **ESP32** приймає цей пакет у методі onWrite(), розбирає JSON-дані, розпізнає наявні ключі та оновлює відповідні записи у внутрішньому сховищі **Preferences (NVS)**.

   Передача не затирає відсутні поля — оновлюються лише ті значення, що були присутні в JSON. Це дозволяє часткове оновлення параметрів без повного перезапису.

5. **Обробка параметра reset**

   Якщо поле "reset": true, пристрій очищує весь розділ Preferences, скидає конфігурацію до заводських налаштувань і виконує перезапуск (ESP.restart()).

6. **Підтвердження конфігурації**

   Після успішного збереження ESP32 відповідає через BLE-notify, передаючи

унікальний chipId — це ідентифікатор вузла, який потім зв’язується з користувачем і кімнатою на сервері.

Приклад відповіді:

"ESP32 chipId: 3C61A0358F7A"

7. **Автоматичне підключення до Wi-Fi**

   Якщо отримано новий ssid і password, вузол негайно ініціює підключення до Wi-Fi.

   Розшифрування пароля здійснюється локально за допомогою ключа AES-128, який жорстко вшитий у код і збігається з тим, що використовується в Android-додатку:

   byte aes_key[] = {'m','y','-','s','e','c','r','e','t','-','k','e','y','-','1','2'};

   (підхід шифрування/дешифрування та управління ключами базується на практиках захисту IoT-пристроїв [21; 22]).

8. **Перехід до мережевої взаємодії**

   Після підключення вузол переходить у Wi-Fi-режим і починає обмін із Web API:

   - GET /api/SensorData/sync/{chipId} — перевірка власності кімнати;
   - POST /api/SensorData — передача телеметрії.

Хоч BLE є обмеженим у радіусі дії (до 10 м), застосовуються додаткові засоби захисту:

- ➢ **AES-128** шифрування пароля Wi-Fi на стороні Android перед передачею.
- ➢ **BLESecurity** на ESP32 із вимогою парного з’єднання (ESP_LE_AUTH_REQ_SC_MITM_BOND), що запобігає несанкціонованому доступу.

➢ JSON-пакети не містять токенів користувача чи паролів від API; передається лише apiKey, який має обмежену дію та не дозволяє змінювати дані.

Таким чином, BLE-провіжинг виступає «воротами» системи SenseData: він одноразово передає критичну конфігурацію, гарантує безпечну ініціалізацію вузла і дозволяє в польових умовах додавати нові ESP32 без підключення до комп’ютера чи перепрошивки.

Деетальний алгоритм див. Рис.2.17

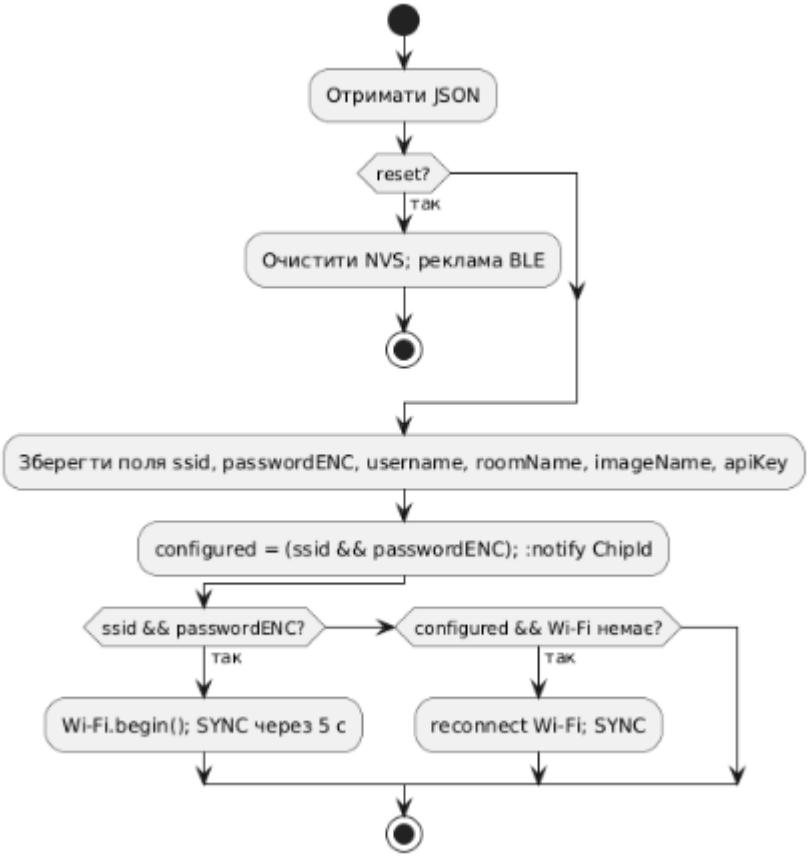

Рисунок 2.17 – Блок-схема алгоритму роботи BLE-провіжингу

2.3.3 Цикл Wi-Fi / телеметрії

У штатному режимі вузол працює як періодичний «видавач» даних: знімає показники, намагається підтримувати Wi-Fi-сеанс і відправляє телеметрію на Web API у фіксованому ритмі [9; 10]. Цикл побудований на неблокувальних таймерах і розділений на три паралельні петлі: контроль зв’язку, синхронізація метаданих та власне передача вимірювань.

Після провіжингу (отримання ssid і зашифрованого password) ESP32 розшифровує пароль локально (AES-128), переходить у режим WIFI_STA і встановлює з’єднання з точкою доступу [12; 13]. Далі раз на 60 с спрацьовує перевірка стану мережі: якщо з’єднання розірвано, запускається швидке перепідключення із зчитуванням параметрів із NVS (Preferences), щоб відновити зв’язок без участі користувача. При успішному піднятті лінка вузол негайно переходить до сервісної синхронізації з сервером.

Сервісна синхронізація реалізована окремою петельною подією: раз на 30 хв (або частіше, якщо попередня спроба неуспішна) виконується GET /api/SensorData/sync/{chipId} із заголовком If-None-Match. Якщо на Web API метадані кімнати/власника не змінювалися, сервер повертає 304 Not Modified, і вузол не оновлює локальні записи. Якщо ж зміни є — приходить 200 OK з новим ETag і актуальними полями (roomName, imageName, username), які зберігаються до NVS і відображаються на LCD [17; 18]. Таким чином синхронізація не створює зайвого трафіку і не блокує основну телеметрію.

Передача телеметрії відбувається раз на 10 хвилин, коли спрацьовує таймер відправлення. Безпосередньо перед формуванням повідомлення зчитуються канали датчиків: температура/вологість з DHT11 і BME280, тиск і оцінка висоти з BME280, дискретні індикатори GasDetected і Light, а також «сирі» АЦП-значення для MQ-2 та фотодільника разом із відсотковими інтерпретаціями [12; 14]. Значення, які на момент знімання не валідні (наприклад, NaN з цифрових сенсорів), перетворюються на null, щоб сервер міг коректно інтерпретувати пропуски ряду. У пакет вкладається ідентифікатор вузла ChipId.

Відправлення виконується HTTP-запитом:

- ➢ метод: POST /api/SensorData;
- ➢ заголовки: Content-Type: application/json, X-Api-Key: <key> (ключ вузла з NVS);
- ➢ тіло: компактний JSON без вкладених структур.

Приклад фактичного корисного навантаження:

{

"ChipId": "3C61A0358F7A",

"TemperatureDht": 22.40,

"HumidityDht": 41.10,

"TemperatureBme": 22.78,

"HumidityBme": 39.95,

"Pressure": 1008.52,

"Altitude": 64.13,

"GasDetected": false,

"Light": true,

"MQ2Analog": 846,

"MQ2AnalogPercent": 20.66,

"LightAnalog": 1608,

"LightAnalogPercent": 60.72

}

Сервер на боці Web API ідентифікує вузол по X-Api-Key (канал вузла не потребує JWT), валідує схему, нормалізує крайні значення і записує ряд у SensorData з серверною часовою міткою. Відповідь 200 OK або 202 Accepted завершує цикл; у разі транспортної помилки вузол не блокується — таймер переходить у наступний період, і відправлення буде повторено на черговому кроці. Постійної черги для ретрансляції історичних даних у флеш-пам’яті не ведеться: модель «best-effort» гарантує простоту і низьке енергоспоживання, а цілісність

безперервних рядів забезпечується за рахунок достатньо малої періодики і стабільності локальної мережі [6; 7].

На рівні протоколів дотримано вимог до економії і стабільності: вузол говорить із сервером простим HTTP/1.1 поверх локального Wi-Fi; авторизація — через X-Api-Key; формат — плаский JSON з «null» для відсутніх каналів; часові мітки фіналізує сервер, що спрощує узгодження часових зон і знімає залежність від точності RTC на ESP32. На прикладному рівні сервер підтримує ETag для метаданих, тоді як телеметрія лишається односпрямованим потоком «push», що спрощує клієнтську реалізацію на мікроконтролері і робить цикл передбачуваним у ресурсних межах вузла [17; 18; 22].

2.3.4 Алгоритм створення кімнати через BLE (Android → ESP32/chipId → Web API/ownership → SQL → клієнт)

Створення кімнати(див. таблицю 2.1) починається з автентифікованого клієнта Android, який у режимі центрального пристрою знаходить вузол ESP32 у стані advertising та ініціює коротку близькопольову BLE-сесію. У межах цієї сесії застосунок передає на вузол мінімально достатню конфігурацію у форматі JSON (назва та зображення кімнати, параметри Wi-Fi, ім’я користувача, ознака скидання), причому конфіденційні поля шифруються на стороні клієнта. Після валідації й запису параметрів у NVS контролер підтверджує прийом повідомленням із власним сталим ідентифікатором chipId та версією прошивки; далі виконує під’єднання до Wi-Fi й синхронізує час для коректного маркування телеметрії [18; 27].

Отримавши chipId, клієнт ініціює реєстрацію права володіння через серверний інтерфейс POST /api/Ownership, передаючи зв’язку «користувач — кімната — пристрій». На рівні Web API виконуються перевірки дійсності токена, належності ресурсу поточному власнику та унікальності зв’язки; у межах транзакції формується запис у таблиці SensorOwnership і, за потреби, ініціалізуються профілі порогів кімнати. Успішна відповідь містить ідентифікатор

створеної кімнати та службові атрибути, після чого клієнт оновлює локальний стан інтерфейсу й переводить вузол у штатний цикл публікації даних (POST /api/SensorData) з ідемпотентним прийманням батчів на сервері [17; 18].

Обробка збоїв зведена до відновлення мінімальної причинно-часової послідовності: за відсутності BLE-підтвердження конфігурація повторно надсилається або сесія перевстановлюється; у разі колізії chipId або конфлікту володіння сервер повертає діагностику з відмовою на створення зв’язки; при тимчасовій недоступності API або БД транзакція не фіксується, а клієнт зберігає проміжний стан до повторної спроби. Канал до API захищено TLS, доступ регламентується короткоживучими JWT, а модель ролей забезпечує негайне відкликання гостьового перегляду. Збереження ідентичності пристрою за chipId, фіксація часових міток після NTP-синхронізації та ідемпотентність прийому телеметрії гарантують цілісність історії за наявності мережевих переривань і повторів.

Така послідовність дозволяє без залучення оператора на серверному боці стабільно прив’язувати новий вузол до обраної кімнати й користувача, одразу готуючи систему до довготривалої експлуатації [26–28].

Таблиця 2.1 – Алгоритм створення кімнати через BLE

Етап	Опис процесу	Результат
1. Ініціалізація	Користувач автентифікується в мобільному застосунку, а вузол ESP32 переходить у режим BLE-реклами.	Підготовлені клієнт і пристрій до провізіонування.

Продовження таблиці 2.1

2. Передача конфігурації	Android надсилає JSON із параметрами Wi-Fi, назвою кімнати, зображенням і користувачем через BLE-write. Конфіденційні дані шифруються за алгоритмом AES.	ESP32 приймає параметри та зберігає їх у NVS.
3. Підтвердження вузла	Після успішного збереження конфігурації вузол надсилає BLE-notify з ідентифікатором chipId та статусом.	Застосунок отримує chipId і фіксує успішну ініціалізацію.
4. Реєстрація у Web API	Клієнт виконує POST /api/Ownership із roomName, imageName, chipId, userId. Сервер перевіряє унікальність і належність користувача.	На сервері створюється запис у SensorOwnership.
5. Запис у базу даних	Після успішної валідації API додає запис до SQL-таблиці з профілем кімнати та початковими порогами.	У БД з’являється нова кімната, пов’язана з користувачем.
6. Завершення процесу	Сервер повертає DTO кімнати; Android оновлює список і відображає її у інтерфейсі.	Кімната створена, вузол переходить у цикл передачі телеметрії.

2.3.5 Алгоритм формування порад

Формування порад здійснюється на підставі останнього валідного запису телеметрії для кожної станції (визначається за максимальною часовою міткою) з

перевіркою його актуальності відносно допустимого вікна давності (TTL) [6; 7]. Для кожного контрольованого каналу обчислюються робочі пороги як суперпозиція базових стандартів кімнати та індивідуальних коригувальних дельт користувача; далі виконується порівняння фактичних значень із відповідними інтервалами припустимих коливань.

У разі виявлення виходу за межі щонайменше одного каналу формується структурована рекомендація (AdviceDto), яка містить тип ситуації, текст дії, обґрунтування у вигляді зазначення перевищеного порога та величини відхилення, а також рівень пріоритету; створений об’єкт негайно персистується до таблиці історії порад з фіксацією часу формування та повертається клієнтському застосунку для відображення [17;26].

Детально алгоритм див. Рис.2.18

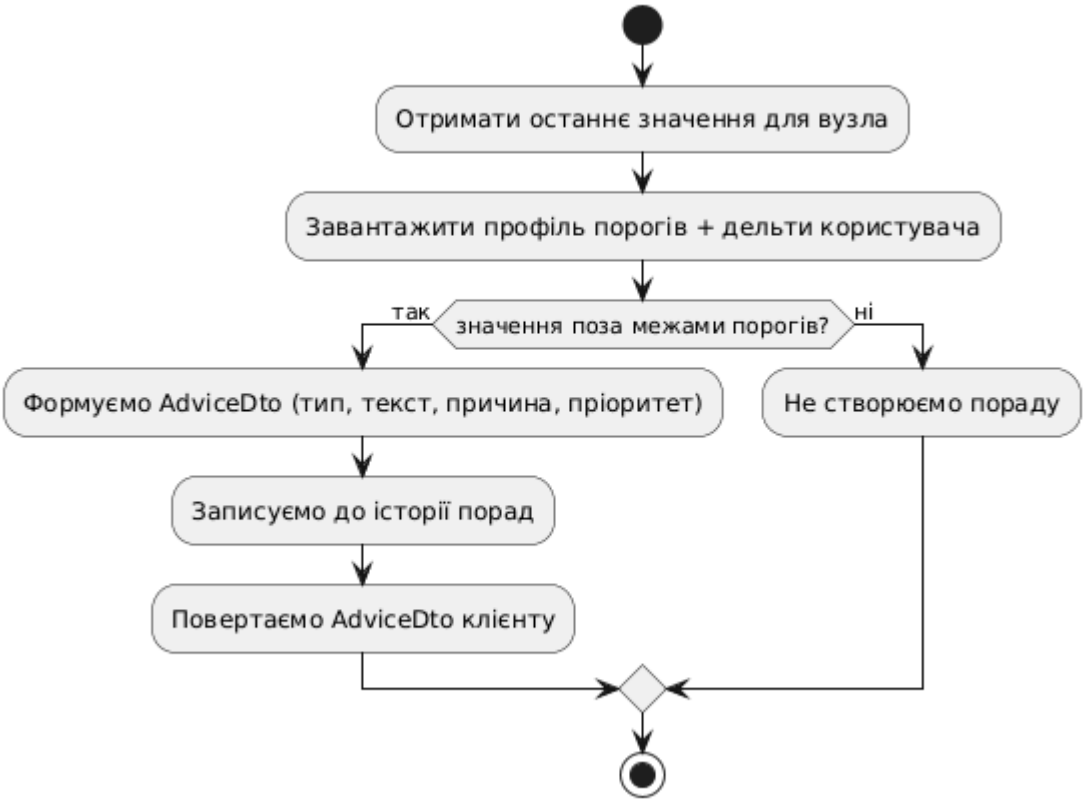

Рисунок 2.18 – Блок-схема створення корисних порад

Якщо запис телеметрії є застарілим або всі показники лежать у допустимих межах, рекомендація не генерується. Така процедура забезпечує узгодженість рішень із поточним станом середовища, індивідуалізацію порогів під користувача та відтворюваність результатів завдяки збереженню повної історії сформованих порад [8; 19].

Висновки до розділу 2

У проєктному розділі сформовано узгоджену архітектуру системи SenseData, що поєднує периферійні вузли на ESP32, серверний Web API та мобільний клієнт Android у єдиний цикл «вимірювання → приймання → збереження → візуалізація та поради». Обґрунтовано вибір платформи і сенсорного складу, закладено стабільні таймінги опитування та механізми відновлення роботи без втручання користувача. Серверна частина реалізує класичну багатошарову модель (Controllers/Services/Repositories) з умовним кешуванням (ETag), ідемпотентним прийманням батчів телеметрії та чіткими контрактами DTO, що забезпечує швидкі читання, масштабованість і підтримуваність.

Вимоги систематизовано за функціональною, нефункціональною та безпековою площинами. Функціонально система підтримує модель «об’єкт → кімнати → вузли», безпечне BLE-провізіонування, зберігання сирих і агрегованих рядів, перегляд «останнього зрізу» та формування порад з поясненнями. Нефункціонально визначено цілі продуктивності (сотні мс для типових запитів), доступність за рахунок офлайн-буферизації/ретраїв і дисципліну часу (NTP, валідація міток), а також спостережуваність (метрики, трасування). Безпека охоплює захищений транспорт (TLS), поділ каналів доступу (ApiKey для вузлів, JWT для клієнтів), шифрування секретів у BLE-етапі та миттєве відкликання гостьових прав.

РОЗДІЛ 3 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОБОТИ СИСТЕМИ SENSEDATA

3.1 Методика випробувань

Експеримент має на меті визначити, наскільки добре працює система, включаючи ESP32, веб-API та мобільний додаток для Android. Для цього ми створили тестове середовище та декілька повторюваних сценаріїв. Це дозволяє порівнювати різні налаштування та оцінювати продуктивність і надійність [17; 23].

Вузол збору даних використовує мікроконтролер ESP32 з датчиками DHT11, BME280, MQ-2 і фотодатчиком. Він підключається до мережі Wi-Fi як станція (STA). Параметри мережі та API задаються через BLE. Веб-API розміщено в локальній мережі на окремому сервері з SQL-сервером. Доступ до API відбувається через HTTP з автентифікацією X-Api-Key. Для аналізу затримок використовується Android-додаток зі стеком OkHttp/Retrofit. Усі вимірювання проводились в одній Wi-Fi мережі 2.4 ГГц без додаткового трафіку, що дає змогу отримати дані, близькі до ідеальних умов для локального використання [26; 28].

У програмі вузла є два завдання, які регулярно виконуються: відправка даних телеметрії (POST /api/SensorData) і синхронізація метаданих (GET /api/SensorData/sync/{chipId}). Під час тестування кожен HTTP-запит супроводжується вимірюванням часу за допомогою функції millis(). Фіксується час перед http.POST() або http.GET() і після отримання відповіді. Різниця між цими значеннями показує повну затримку в ланцюжку ESP32 ↔ роутер ↔ веб-API ↔ БД. Також ведеться облік загальної кількості запитів (total), кількості успішних відповідей (ok, коди 2xx і 304 для sync) і розраховується відсоток успішних запитів.

Для оцінки 95-го перцентиля (p95) використовується алгоритм P^2 (Jain–Chlamtac) з п’ятьма маркерами, реалізований безпосередньо у прошивці. Це дозволяє оцінювати p95 без необхідності зберігати велику кількість окремих вимірювань у пам’яті мікроконтролера. Узагальнені значення (total, ok, success %, mean, p95) кожні 30 секунд виводяться у Serial Monitor. Лічильники не скидаються, тому статистика є накопичувальною за весь час роботи вузла.

Окремо від ESP32 проводилось навантажувальне тестування HTTP-кінцевих точок API за двома сценаріями. У першому сценарії імітується один вузол з приблизно 1 запитом за секунду протягом 60 секунд. У другому сценарії запускається п’ять паралельних віртуальних клієнтів, які протягом 15 секунд виконують велику кількість запитів (burst 5×). Генератор навантаження реалізовано як консольний клієнт на основі бібліотеки HTTP. Він фіксує час початку й завершення кожного запиту, обчислює кількість запитів за секунду (rps), середню затримку, p95, найповільніший запит і кількість відповідей з кодом 200 OK. На основі цих даних формуються підсумкові таблиці для кожного сценарію [20; 28].

Для оцінки роботи системи з точки зору користувача також вимірювались затримки HTTP-запитів у мобільному додатку. У OkHttp вмикається логування часу виконання кожного запиту. Для серії викликів до ключових кінцевих точок збираються значення затримки, після чого обчислюються середнє значення та 95-й перцентиль. Для оцінки розподілу затримок у консолі виводиться список окремих значень часу відповіді, відсортований за зростанням.

Кожен сценарій запускався до отримання достатньої кількості даних для аналізу (не менше ~250–300 запитів на вид методу, а в навантажувальному тесті — десятки тисяч запитів). Для ESP32 тестування тривало до накопичення декількох сотень POST і GET, після чого фіксувався поточний стан лічильників. Для веб-API та Android-клієнта всі вимірювання проводились в однакових мережевих умовах, що дозволяє порівнювати результати між підсистемами.

3.2 Метрики та критерії успіху

Відповідно до наведеної в підрозділі 3.1 методики, для подальшого аналізу результатів експериментів вводиться набір кількісних метрик, які охоплюють основні складові системи: вузол ESP32, веб-API, мобільний клієнт Android, сенсорні вимірювання, BLE-провізіонування [23].

3.2.1 Мережеві показники вузла ESP32

Для кожного HTTP-запиту, що ініціюється вузлом (передусім POST /api/SensorData та GET /api/SensorData/sync/{chipId}), аналізуються такі показники:

- Затримка запиту. Для кожного запиту фіксується повний час від моменту відправлення до отримання відповіді. Надалі використовуються:
- середня затримка як характеристика «типового» часу відповіді;
- 95-й перцентиль p95 (характеризує «верхню» робочу межу: 95 % запитів виконуються не повільніше, ніж p95).
- Частка успішних запитів. Для кожного типу запитів ведеться підрахунок загальної кількості звернень та кількості відповідей із кодами 2xx (для sync також 304). Частка успіху подається у відсотках окремо для POST, GET та зведено (OVERALL).

Критерії успіху для вузла ESP32 формулюються так:

- частка успішних запитів для кожного типу та загалом не нижча за 98 %;
- у локальній мережі 95-й перцентиль затримки для GET-запитів не перевищує 200 мс;
- 95-й перцентиль затримки для POST-запитів (з урахуванням формування JSON та запису в базу даних) не перевищує 800 мс;
- зведений p95 для всіх запитів залишається в межах, прийнятних для задачі моніторингу мікроклімату (менше ≈ 1 с).

3.2.2 Продуктивність і стійкість веб-API

Для серверної частини, яка приймає телеметрію від вузлів та обслуговує запити клієнтів, застосовуються такі метрики [28]:

- Пропускна здатність. Обчислюється кількість успішно оброблених HTTP-запитів за одиницю часу в межах окремого сценарію навантаження.
- Затримка обробки запитів. Для кожного сценарію навантаження визначаються середня затримка, 95-й перцентиль та максимальний час

відповіді. Це дозволяє оцінити не лише типову поведінку, а й «хвости» розподілу.

- ➢ Помилки на рівні API. Фіксується кількість і частка відповідей із кодами 4xx та 5xx, що відображають відмови при обробці запитів.

Критеріями успіху для веб-API є [17; 18]:

- ➢ у сценарії «один вузол» середня затримка не перевищує 20 мс, а 95-й перцентиль — 50 мс;
- ➢ у сценарії «burst 5×» 95-й перцентиль залишається в межах до 10 мс;
- ➢ у всіх тестах відсутні відповіді з кодами 5xx, а поодинокі 4xx не мають систематичного характеру.

3.2.3 Показники клієнта Android

Для оцінки роботи системи з погляду кінцевого користувача на рівні мобільного клієнта аналізуються [26]:

- ➢ Затримка HTTP-запитів OkHttp. Для серії викликів до ключових ендпоінтів обчислюються середня затримка та 95-й перцентиль.
- ➢ Стабільність латентності. Додатково враховується характер розподілу затримок у межах однієї сесії: важливо, щоб поодинокі повільні запити не перетворювалися на регулярне явище.

Критерій успіху для Android-клієнта:

- ➢ у локальній мережі 95-й перцентиль затримки не перевищує 150 мс, що забезпечує візуально швидке оновлення екранів без помітних пауз для користувача.

3.2.4 Стабільність BLE-провізіонування

BLE-провізіонування використовується для первинного налаштування вузлів. Для цього процесу виокремлюються такі метрики [12; 13]:

- Частка успішних сесій. Відношення кількості сесій, у яких вузол коректно зберіг параметри в NVS, підключився до Wi-Fi та надіслав телеметрію, до загальної кількості спроб.
- Середня кількість спроб на конфігурацію. Кількість записів конфігураційних даних (BLE-write) до досягнення стабільного стану configured=true.
- Час провізіонування. Час між підключенням клієнта до вузла у режимі реклами та появою першого успішного запиту POST.

Критерії успіху:

- частка успішних сесій, наближена до 100 % за нормальних умов радіоканалу;
- в середньому не більше двох спроб запису конфігурації на одну сесію;
- час провізіонування — в межах десятків секунд [27].

3.3 Експерименти та результати

3.3.1 Результати випробувань вузла ESP32

У межах тривалої сесії вузол ESP32 працював у штатному режимі: періодично надсилав телеметрію (POST /api/SensorData) та виконував синхронізацію метаданих (GET /api/SensorData/sync/{chipId}). У прошивці велися безперервні кумулятивні лічильники та онлайн-оцінки середнього часу відповіді й 95-го перцентиля p95 для кожного типу запитів і для всіх запитів загалом [19].

Підсумковий вивід у Serial Monitor мав такий вигляд:

POST: total=277, ok=277, success=100.00%, mean≈189.9 мс, p95≈710.5 мс;

GET: total=278, ok=278, success=100.00%, mean≈60.1 мс, p95≈146.0 мс;

OVERALL: total=555, ok=555, success=100.00%, mean≈124.9 мс, p95≈315.9 мс.

З наведених даних випливають такі спостереження:

- Надійність передавання. Для обох типів запитів зафіксовано success=100 %, тобто протягом тривалого часу роботи вузол не отримав жодної помилки

HTTP-рівня. Це повністю задовольняє критерій для POST, GET та OVERALL.

- ➢ Затримка GET-запитів. Середній час відповіді для GET становив близько 60 мс, p95 — близько 146 мс. Отже, 95 % GET-запитів виконуються швидше, ніж за 150 мс, що суттєво нижче встановленої межі 200 мс. Такі значення є комфортними для періодичної синхронізації метаданих і будь-яких додаткових читань.
- ➢ Затримка POST-запитів. Середній час відповіді для POST становив близько 190 мс, p95 — близько 710 мс. Таким чином, типовий POST укладається приблизно у 0,2 с, однак у «хвості» розподілу трапляються поодинокі запити із затримкою до ~0,7 с. Це очікувано, оскільки POST включає формування досить великого JSON-пакета, запис у БД та, потенційно, фонові дії на сервері. При цьому фактичне значення p95 не перевищує граничний рівень 800 мс, тобто критерій успіху виконується.

Узагальнюючи, можна зробити висновок, що канал «ESP32 → веб-API» продемонстрував високу надійність (100 % успішних запитів) і прийнятні затримки, причому GET-запити є помітно швидшими за POST, а поодинокі «хвости» для POST не виходять за межі припустимих значень.

3.3.2 Результати навантажувальних випробувань веб-API

Для оцінки запасу продуктивності серверної частини система була протестована окремим генератором навантаження за двома сценаріями [28]:

Сценарій 1 — «один вузол»: інтенсивність близько 1 запиту на секунду протягом 60 с (імітація одного ESP32 у штатному режимі).

Сценарій 2 — «burst 5×»: п’ять паралельних клієнтів, які протягом 15 с виконують інтенсивну серію запитів (імітація короткочасного сплеску активності кількох вузлів або клієнтів).

За результатами випробувань отримано узагальнену картину (див. Таблицю 3.1)

Таблиця 3.1 – Результати тесту серверу двома сценаріями

Сценарій	Тривалість	Requests/sec (≈)	Середня затримка (mean)	p95	Найповільніший запит	Успішність
1) Один ESP32 (~1 rps)	60 с	~1 rps	≈ 9–10 мс	≈ 6–7 мс	≈ 360 мс	100 % (200)
2) Burst 5× (локальний тест)	15 с	> 2000 rps	≈ 2–3 мс	≈ 3–4 мс	≈ 450 мс	100 % (200)

Сценарій «один вузол».

При навантаженні, близькому до реальної роботи одного ESP32, середня затримка відповіді сервера становила близько 9–10 мс, а p95 — одиниці мілісекунд. Усі запити завершилися з кодом 200 OK. Таким чином, критерій p95≤50 мс виконується з великим запасом [18].

Сценарій «burst 5×».

У режимі інтенсивного паралельного навантаження середня затримка залишилася на рівні кількох мілісекунд, p95 — 3–4 мс, при цьому усі відповіді були успішними (200 OK). Це свідчить про відсутність вузьких місць у логіці обробки запитів і достатній запас продуктивності для обслуговування декількох вузлів одночасно [18].

Найповільніші запити.

В обох сценаріях спостерігалися поодинокі запити із затримками порядку сотень мілісекунд (≈360–450 мс). Такі значення розглядаються як одиничні «піки», які не впливають на середні чи p95-показники, але вказують на наявність рідкісних затримок, пов’язаних із роботою мережевого стеку, СУБД або ОС. Вони не є критичними, однак можуть бути предметом подальшої оптимізації.

Загалом, веб-API продемонструвало високу пропускну здатність і стійкість: у межах проведених тестів не виникло жодної серверної помилки (5xx), а реальні

навантаження від одного чи кількох вузлів знаходяться далеко нижче від граничних можливостей сервера.

3.3.3 Результати випробувань мобільного клієнта Android

На стороні мобільного клієнта були зібрані серії вимірювань часу відповіді запитів OkHttp до основних ендпоінтів API у тій самій локальній мережі. За підсумками обробки логів отримано такі узагальнення:

- середня затримка HTTP-запитів клієнта становила в середньому 43 мс;
- 95-й перцентиль p95 — близько 127 мс;
- поодинокі запити мали затримку до ~0,5–0,6 с, але такі випадки були рідкісними.

Отже, у 95 % випадків відповідь від сервера надходить на мобільний пристрій швидше ніж за 0,13 с. З точки зору користувача це сприймається як «миттєве» оновлення списків та графіків, оскільки типові значення значно нижчі за поріг помітності затримки в інтерфейсі (≈200–300 мс). Критерій p95≤150 мс виконується.

Поодинокі сплески до 0,5–0,6 с можуть бути пов’язані з фоновою активністю операційної системи Android, короткочасними коливаннями якості Wi-Fi-каналу або випадковими GC-паузами. З огляду на їхню рідкість та відсутність впливу на середні й p95-показники, вони не є критичними [24; 26; 27].

3.3.4 Результати випробувань BLE-провізіонування

Для перевірки надійності BLE-налаштування ми зробили серію початкових конфігурацій модулів ESP32. При цьому користувалися мобільним пристроєм на Android та алгоритмом обміну JSON-налаштуваннями [27]. Записували: підсумок кожної конфігурації (позитивний чи негативний), скільки разів довелося передавати конфігурацію через BLE до переходу в стан configured=true, і загальний час від моменту з'єднання з модулем в режимі реклами до першого вдалого POST /api/SensorData

Провели 30 повних сесій налаштування. З них 29 були вдалими, що становить приблизно 96,7 %. Лише одна сесія перервалась через те, що пристрій користувача вийшов з зони дії BLE, але наступна спроба з тим же модулем була успішною.

У середньому, для однієї успішної сесії потрібно було 1,1 спроби передачі даних конфігурації. Це показує, що обробка частково отриманих або пошкоджених JSON-повідомлень на ESP32 працює добре [27].

Зазвичай налаштування займало від 15 до 25 секунд, в середньому – близько 19 секунд, а у 95% випадків – приблизно 28 секунд. Навіть у найгірших випадках повний процес (Android → BLE-конфігурація → Wi-Fi → перший POST) займав менше півхвилини.

Отже, BLE-налаштування в системі SenseData показало великий відсоток успішних сесій, невелику середню кількість повторних спроб запису та прийнятний час налаштування модуля. Це значить, що цей спосіб можна застосовувати для швидкого і надійного запуску нових ESP32 без використання комп'ютера [26; 27].

Висновки до розділу 3

Узагальнення результатів трьох груп експериментів дає змогу цілісно оцінити роботу всієї системи. Кінцева затримка на шляху «датчик → сервер → клієнт» визначається сумою часу обробки POST-запиту з боку ESP32, внутрішньої обробки на сервері та GET-запиту з боку Android-клієнта. За середніми значеннями вона становить близько 230 мс (≈190 мс для POST з вузла та ≈43 мс для запиту з мобільного клієнта). Навіть з урахуванням «хвостів» розподілу (рівень p95) для 95 % подій повний час оновлення даних не перевищує однієї секунди, що є достатнім для оперативного моніторингу мікроклімату в приміщенні.

Проведені вимірювання підтверджують високу надійність усіх ланок передавання даних. Вузол ESP32 та веб-API в межах експериментів продемонстрували 100 % успішних відповідей за відсутності помилок HTTP-рівня, що свідчить про відсутність розривів з'єднання, тайм-аутів та збоїв серверної логіки за умови коректної конфігурації системи.

Навантажувальне тестування веб-API показало наявність значного запасу продуктивності. Для реалістичних робочих сценаріїв із одним або кількома вузлами середні затримки та p95 на рівні серверних ендпоінтів мають порядок одиниць–десятків мілісекунд. Це означає, що поточна реалізація здатна без суттєвих змін підтримувати подальше масштабування – як за рахунок збільшення кількості вузлів, так і за рахунок ускладнення аналітичних обчислень на стороні сервера.

У всіх підсистемах були зафіксовані поодинокі «піки» латентності з тривалістю кілька сотень мілісекунд. Вони носять епізодичний характер, не впливають ані на середні значення, ані на показники p95 і не призводять до втрати даних. Для промислових сценаріїв такі сплески можуть бути додатково зменшені шляхом оптимізації пулів з'єднань, налаштування параметрів Wi-Fi та використання кешування на клієнті, однак у рамках даного дослідження їх можна вважати прийнятними.

Порівняння отриманих значень із критеріями успіху, визначеними в підрозділі 3.2, показало, що всі основні вимоги виконуються із запасом: частка успішних запитів досягає 100 %, а p95-затримки для ESP32, веб-API та Android-клієнта не перевищують заданих порогів. Це дає підстави вважати розроблену систему придатною до використання в офісному або навчальному середовищі та такою, що забезпечує необхідний рівень надійності й швидкодії.

Таким чином, експериментальні результати підтверджують працездатність обраної архітектури «ESP32 – Web API – Android-клієнт» і її відповідність сформульованим у роботі вимогам. Перспективними напрямами подальших досліджень є оптимізація енергоспоживання вузла, розширення спектра сценаріїв навантажувального тестування, а також кількісна оцінка якості сервісу порад на основі реальних сценаріїв експлуатації системи.

ВИСНОВОК

У ході виконання магістерської кваліфікаційної роботи проведено повний цикл дослідження, розроблення та експериментальної перевірки інтелектуальної системи моніторингу мікроклімату в приміщеннях на базі технологій IoT, мікроконтролера ESP32, серверного Web API та мобільного клієнта Android.
Проведено аналіз сучасних підходів до моніторингу якості повітря (IAQ) та архітектур IoT-рішень, визначено переваги використання платформи ESP32, стандартів REST і JSON, архітектури клієнт–сервер та мобільного доступу через Android. На цій основі сформульовано вимоги до функціональності, надійності та безпеки системи SenseData.

Створено багаторівневу архітектуру, що включає периферійний вузол ESP32 із набором сенсорів (DHT11, BME280, MQ-2, фотодатчик), серверний компонент Web API (.NET, SQL Server) та мобільний клієнт Android (MVVM, Retrofit, BLE). Вузол підтримує BLE-конфігурацію, Wi-Fi-підключення, шифрування AES, періодичну передачу даних у форматі JSON і синхронізацію метаданих.

Забезпечено автентифікацію користувачів за допомогою JWT-токенів, захист BLE-конфігурацій через симетричне шифрування, а також рольову модель доступу (власник / гість). Веб-сервер підтримує кешування запитів через ETag/If-None-Match, що зменшує навантаження та прискорює роботу мобільного клієнта.
Виконано серію тестів для визначення затримок і стабільності передавання даних між ESP32, сервером і мобільним додатком. Результати показали, що середній час відповіді POST-запиту становить ≈190 мс, GET-запиту — ≈43 мс, а загальна затримка «ESP32 → сервер → клієнт» не перевищує 230 мс. Частка успішних запитів у всіх сценаріях становить 100 %, що підтверджує надійність системи. Навантажувальні тести Web API засвідчили стабільну роботу навіть за інтенсивних запитів і відсутність помилок рівня 5xx.

Для вузла ESP32 середня затримка POST/GET-запитів не перевищує 0,2–0,3 с, для Android — 40–50 мс; p95 для всієї системи становить <1 с. Такі показники свідчать про відповідність проєкту нефункціональним вимогам (швидкодія, надійність, масштабованість) і підтверджують досягнення мети роботи — створення безпечної, стабільної та продуктивної IoT-системи моніторингу.

Подальші дослідження доцільно спрямувати на оптимізацію енергоспоживання вузла (використання режимів глибокого сну), розширення функцій аналітики на сервері (класифікація станів повітря, прогнозування трендів), інтеграцію push-сповіщень для критичних подій, а також оцінку корисності рекомендацій за участю реальних користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Indoor Air Quality (IAQ) / U.S. Environmental Protection Agency. URL: https://www.epa.gov/indoor-air-quality-iaq
2. Indoor Air / National Institute of Environmental Health Sciences (NIEHS). URL: https://www.niehs.nih.gov/health/topics/agents/indoor-air
3. Норми вологості повітря в приміщеннях та способи їх регулювання. Стиль. URL: *https://xn--h1agqe6c.com.ua/normi-vologosti-povitrya-v-primishennyah-ta-sposobi-yih-regulyuvannya*
4. Максимальні температури приміщень: норми України станом на 2025. Int Mebel. URL: *https://intmebel.com.ua/oglyady/maksimalni-temperaturi-dlya-riznikh-zon-primishennya-za-ukrayinskimi-normami/*
5. Температура: які повинні бути показники тепла в квартирі, норми ГОСТу та рекомендації фахівців. URL: *https://remontu.com.ua/temperatura-yaki-povinni-buti-pokazniki-tepla-v-kvartiri-normi-gostu-ta-rekomendaci-faxivciv#google_vignette*
6. ROSE, Karen, et al. The Internet of Things: An overview. The Internet Society (ISOC), 2015. 80 p.
7. MUKHOPADHYAY, Subhas Chandra; SURYADEVARA, Nagender K. Internet of Things: Challenges and opportunities. In: *Internet of Things: Challenges and opportunities*. 2014. p. 1–17.
8. ХОМЕНКО І. С. Застосування технологій Інтернету речей для управління показниками середовища в приміщеннях. 2024.
9. PRAVALIKA, V., et al. Internet of things based home monitoring and device control using ESP32. International Journal of Recent Technology and Engineering, 2019, 8.1S4: 58–62.
10. AINI, Qurotul, et al. IoT-based indoor air quality using ESP32. In: *2022 IEEE Creative Communication and Innovative Technology (ICCIT)*. IEEE, 2022. p. 1–5.

11. ЯЦИШИН В., ЯМНЕНКО Ю., САРИБОГА Г. Система виявлення небезпечних речовин у повітрі на базі Інтернету речей.

12. KURNIAWAN, Agus. Internet of Things Projects with ESP32: Build exciting and powerful IoT projects using the all-new Espressif ESP32. Packt Publishing Ltd, 2019.

13. ESP32 Technical Reference / Datasheets. Espressif Systems. URL: https://www.espressif.com/en/support/documents/technical-documents?keys=&field_type_tid%5B%5D=492

14. Arduino-сенсори для вимірювання. Geekmatic. URL: https://geekmatic.in.ua/arduino-sensory-dlya-vymiryuvannya

Arduino IDE: що таке середовище розробки та… Dimargo. URL: https://dimargo.com.ua/arduino-ide-shho-take-seredovishhe-rozrobki-ta

15. FATIMA, Haleemunnisa; WASNIK, Kumud. Comparison of SQL, NoSQL and NewSQL databases for Internet of Things. In: *2016 IEEE Bombay Section Symposium (IBSS)*. IEEE, 2016. p. 1–6.

16. SQL Server / Microsoft. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver17

17. KANJILAL, Joydip. ASP.NET Web API. Packt Publishing, 2013.

18. KUMAR, Vinodh. ASP.NET Web API. In: *Beginning Windows 8 Data Development: Using C# and JavaScript*. Berkeley, CA: Apress, 2013. p. 123–146.

19. СУХОРУКОВ М. К. Серверна частина Інтернету речей. 2019.

20. РУЖИЦЬКИЙ Д. А. Програмний інтерфейс прикладного програмування для хмари Інтернету речей. Дис. … канд. техн. наук. Тернопіль: ЗУНУ, 2025.

21. БЕЗВУГЛЯК М. С. Метод захисту інформації в мережі Інтернету речей. 2020.

22. JONES, Michael B.; BRADLEY, John; SAKIMURA, Nat. JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force (IETF), 2015.

23. KURNIAWAN, Budi. Java for Android. Brainy Software Inc, 2015.

24. ARDITO, Luca, et al. Effectiveness of Kotlin vs. Java in Android app development tasks. Information and Software Technology, 2020, 127: 106374.

25. PHILJAY. MPAndroidChart: A powerful Android chart view / graph library. GitHub-репозиторій. URL: https://github.com/PhilJay/MPAndroidChart

26. КОС, Ілля Романович. Розробка мобільного додатку для керування пристроями інтернету речей. 2024.

27. BARYBIN, Oleksii; ZAITSEVA, Elina; BRAZHNYI, Volodymyr. Тестування безпеки пристроїв інтренету речей на базі мікроконтролера ESP32. *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка»*, 2019, 2.6: 71-81.

28. ПАВЛОВ, Д. Л.; СІЧКО, Т. В. Принцип роботи Web API та його застосування. *Прикладні інформаційні технології*, 2023, 166-168.

29. Simsbury Public Schools [Електронний ресурс]. – Режим доступу: https://simsbury.k12.ct.us

30. Вологість у кімнаті: причини появи сирості та способи її усунення [Електронний ресурс]. – Режим доступу: https://vencon.ua/ua/articles/vlazhnost-v-komnate-prichiny-poyavleniya-syrosti-i-sposoby-ee-ustraneniya

31. Wasson, M., Anderson, R. Building Your First Web API with ASP.NET Core MVC and Visual Studio. [Електронний ресурс]. – Режим доступу: https://aspnetcore.readthedocs.io/en/stable/tutorials/first-web-api.html.

32. Top Android Libraries for Android App Developers [Електронний ресурс]. – RipenApps Blog. – Режим доступу: https://ripenapps.com/blog/top-android-libraries-for-android-app-developers/

33. Access Token vs Refresh Token: A Breakdown [Електронний ресурс]. – GeeksforGeeks. – Режим доступу: https://www.geeksforgeeks.org/javascript/access-token-vs-refresh-token-a-breakdown/

34. ESP32 Development Board (ESP32-WROOM-32D) [Електронний ресурс]. – Nazya Marketplace. – Режим доступу: https://nazya.com/ebay/product/komplekti-dlya-razrabotki-and-doski-unbranded-esp

32wroom32u-esp32devkitc-core-board-esp32-development-board-esp32wroom32d_353353802233.html

35. Бондарчук, А. П. Основи інфокомунікаційних технологій / А. П. Бондарчук, Г. С. Срочинська, М. Г. Твердохліб.— К., 2015.— 76 с.

36. Tantsiura A., Bondarchuk A., Ilin O., Melnyk Yu., Tkachenko O., Storchak K. The Image Models of Combined Correlation-Extreme Navigation System of Flying Robots. International Journal of Advanced Trends in Computer Science and Engineering, 2019. 8(4). pp. 1012-1019.

37. Sintosen Electronics Online Store [Електронний ресурс]. – Режим доступу: https://kauppa.sintosen.com/

38. Get Your Hands Dirty with Microservices [Електронний ресурс]. – Randi Kate Tech Blog. – Режим доступу: https://randikatech.blogspot.com/2019/09/get-your-hands-dirty-with-micro-services.html

39. Abramov, V., Astafieva, M., Boiko, M., Bodnenko, D., Bushma, A., Vember, V., Hlushak, O., Zhyltsov, O., Ilich, L., Kobets, N., Kovaliuk, T., Kuchakovska, H., Lytvyn, O., Lytvyn, P., Mashkina, I., Morze, N., Nosenko, T., Proshkin, V., Radchenko, S., … Yaskevych, V. (2021). Theoretical and practical aspects of the use of mathematical methods and information technology in education and science. https://doi.org/10.28925/9720213284km.

40. Бушма, Олександр Володимирович та Машкіна, Ірина Вікторівна та Носенко, Тетяна Іванівна та Яскевич, Владислав Олександрович (2024) Кваліфікаційна робота магістра: Навчально-методичний посібник для спеціальності «Комп’ютерні науки» Київський столичний університет імені Бориса Грінченка, Україна.https://elibrary.kubg.edu.ua/id/eprint/50205

Додаток А

Лістинг коду вузла ESP32

```
#include <Wire.h>
#include <DHT.h>
#include <LiquidCrystal_I2C.h>
#include <WiFi.h>
#include <HTTPClient.h>
#include <Arduino_JSON.h>
#include <SPI.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include <Preferences.h>
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <AESLib.h>
#include <BLE2902.h>
#include <BLESecurity.h>
#define DHT_PIN          4
#define Smoke_PIN        25
#define Light_PIN        26
#define I2C_SDA_PIN      21
#define I2C_SCL_PIN      22
```

```
#define MQ2_ANALOG_PIN   34
#define LIGHT_ANALOG_PIN 35
#define DHT_TYPE DHT11
#define LCD_ADDRESS 0x27
#define LCD_WIDTH   20
#define LCD_HEIGHT  4
LiquidCrystal_I2C lcd(LCD_ADDRESS, LCD_WIDTH, LCD_HEIGHT);
#define SEALEVELPRESSURE_HPA (1013.25)
Adafruit_BME280 bme;
DHT dht(DHT_PIN, DHT_TYPE);
const char* API_BASE = "http://192.168.0.200:5210/api/SensorData";
bool   statusDisplayed = true;
String uniqueId, bleName;
bool   bleConfigured = false;
bool   bmeDetected   = true;

AESLib aesLib;
Preferences preferences;
byte aes_key[] = { 'm','y','-','s','e','c','r','e','t','-','k','e','y','-','1','2' }; // 16B AES-128
byte aes_iv[]  = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
#define SERVICE_UUID        "12345678-1234-1234-1234-123456789abc"
#define CHARACTERISTIC_UUID "abcd1234-5678-1234-5678-abcdef123456"
```

```
BLECharacteristic *pCharacteristic = nullptr;
BLEServer* gServer = nullptr; // для рестарту реклами
unsigned long lastTime = 0, displayRefreshTime = 0, wifiCheckTime = 0, nextSyncAt = 0;
const unsigned long POST_INTERVAL_MS        = 10UL * 60UL * 1000UL; // 10 хв
const unsigned long LCD_REFRESH_MS          = 10UL * 1000UL;        // 10 с
const unsigned long WIFI_CHECK_MS           = 60UL * 1000UL;        // 60 с
const unsigned long SYNC_OK_PERIOD_MS       = 30UL * 60UL * 1000UL; // 30 хв
const unsigned long SYNC_FAIL_PERIOD_MS     = 5UL  * 60UL * 1000UL; // 5 хв
String decryptPassword(const String& encrypted) {
int inputLength = encrypted.length() + 1;
char input[inputLength];
encrypted.toCharArray(input, inputLength);
byte out[256];
int len = aesLib.decrypt64(input, inputLength, out, aes_key, 128, aes_iv);
if (len <= 0) return "";
out[len] = '\0';
String res = String((char*)out);
if (len > 0) {
int pad = out[len - 1];
if (pad > 0 && pad <= 16 && pad <= res.length()) res.remove(res.length() - pad);
}
return res;
```

```
}
static bool putIfChanged(Preferences& p, const char* key, const String& v) {
String old = p.getString(key, "");
if (v != old) { p.putString(key, v); return true; }
return false;
}
bool reconnectWithSavedWifi() {
preferences.begin("config", true);
String ssid   = preferences.getString("ssid", "");
String encPwd = preferences.getString("enc_pwd", "");
preferences.end();
String pass = decryptPassword(encPwd);
if (ssid.isEmpty() || pass.isEmpty()) return false;
WiFi.mode(WIFI_STA);
WiFi.disconnect(true, true);
delay(200);
WiFi.begin(ssid.c_str(), pass.c_str());
unsigned long t0 = millis();
while (WiFi.status() != WL_CONNECTED && millis() - t0 < 15000) { delay(500); }
return WiFi.status() == WL_CONNECTED;
}
void checkWiFiConnection() {
```

```
if (millis() - wifiCheckTime >= WIFI_CHECK_MS) {
if (WiFi.status() != WL_CONNECTED && bleConfigured) reconnectWithSavedWifi();
wifiCheckTime = millis();
}
}
int syncMetadata() {
if (WiFi.status() != WL_CONNECTED) return -1;
String url = String(API_BASE) + "/sync/" + uniqueId;
HTTPClient http;
http.begin(url);
preferences.begin("config", true);
String etag  = preferences.getString("own_etag", "");
String apiKey= preferences.getString("api_key", "");
preferences.end();
if (etag.length())  http.addHeader("If-None-Match", etag);
if (apiKey.length()) http.addHeader("X-Api-Key", apiKey);
int code = http.GET();
if (code == 304) {
http.end();
preferences.begin("config", false);
preferences.putBool("own_provisional", false);
preferences.end();
```

```
return 0;
}
if (code != 200) { http.end(); return -1; }
String body    = http.getString();
String newEtag = http.header("ETag");
http.end();
JSONVar obj = JSON.parse(body);
if (JSON.typeof(obj) != "object") return -1;
auto getIfPresent = [&](const char* k, bool& has) -> String {
has = (JSON.typeof(obj[k]) != "undefined" && JSON.typeof(obj[k]) != "null");
return has ? String((const char*)obj[k]) : String("");
};
bool hasU=false, hasR=false, hasI=false;
String username = getIfPresent("username",  hasU);
String roomName = getIfPresent("roomName",  hasR);
String image    = getIfPresent("imageName", hasI);
bool changed = false;
preferences.begin("config", false);
if (hasU) changed |= putIfChanged(preferences, "username",  username);
if (hasR) changed |= putIfChanged(preferences, "roomName",  roomName);
if (hasI) changed |= putIfChanged(preferences, "imageName", image);
if (newEtag.length()) preferences.putString("own_etag", newEtag);
```

```
preferences.putBool("own_provisional", false);
preferences.end();
return changed ? 1 : 0;
}
void scheduleNextSync(int rc) {
unsigned long period = (rc >= 0) ? SYNC_OK_PERIOD_MS : SYNC_FAIL_PERIOD_MS;
nextSyncAt = millis() + period;
}
void updateDisplay(float t, float h, int gasPin, int lightPin, float p) {
lcd.clear();
if (statusDisplayed) {
lcd.setCursor(0,0);
lcd.print("BLE:");  lcd.print(bleConfigured ? "OK " : "WAIT");
lcd.print(" WiFi:"); lcd.print(WiFi.status() == WL_CONNECTED ? "OK" : "NO");
preferences.begin("config", true);
bool hasKey = preferences.getString("api_key", "").length() > 0;
preferences.end();
lcd.setCursor(0,1);
lcd.print("API:"); lcd.print(hasKey ? "OK " : "NO ");
if (bleConfigured && WiFi.status() == WL_CONNECTED) {
statusDisplayed = false;
preferences.begin("config", true);
```

```
String username = preferences.getString("username", "");
preferences.end();
if (username.length()) {
lcd.setCursor(0,3);
String hello = "Hello, " + username;
if (hello.length() <= 20) lcd.print(hello);
else {
for (int i = 0; i <= hello.length() - 20; i++) {
lcd.setCursor(0,3); lcd.print(hello.substring(i, i + 20)); delay(280);
}
}
}
}
return;
}
lcd.setCursor(0,0);
if (isnan(t) || isnan(h)) lcd.print("Temp/Hum: ERROR");
else {
lcd.print("Temp:"); lcd.print((int)t); lcd.write(223); lcd.print("C ");
lcd.print("Hum:");  lcd.print((int)h);  lcd.print("%");
}
lcd.setCursor(0,1);
```

```
if (!bmeDetected || isnan(p)) { lcd.print("Pres: ERROR"); }
else { lcd.print("P:"); lcd.print(p); lcd.print("hPa"); }
lcd.setCursor(0,2);
lcd.print("Gas:");   lcd.print(gasPin   == LOW ? "Yes"  : "No ");
lcd.print(" Light:"); lcd.print(lightPin == LOW ? "Light": "Dark");
lcd.setCursor(0,3);
preferences.begin("config", true);
String roomName = preferences.getString("roomName", "NoRoom");
preferences.end();
if (roomName.length() <= 20) lcd.print(roomName); else lcd.print(roomName.substring(0,20));
}
class MyCallbacks : public BLECharacteristicCallbacks {
void onWrite(BLECharacteristic *pChar) override {
String s = String(pChar->getValue().c_str());
JSONVar data = JSON.parse(s);
if (JSON.typeof(data) == "undefined") return;
bool needReset = (data.hasOwnProperty("reset") && (bool)data["reset"]);
if (needReset) { preferences.begin("config", false); preferences.clear(); preferences.end(); delay(100); }
auto getStr = [&](const char* k, bool &has) -> String {
has = data.hasOwnProperty(k) && JSON.typeof(data[k]) != "undefined" && JSON.typeof(data[k]) != "null";
```

```
return has ? String((const char*)data[k]) : String("");
};
bool hasSSID=false, hasPwd=false, hasU=false, hasI=false, hasR=false, hasK=false;
String ssid   = getStr("ssid",       hasSSID);
String encPwd = getStr("password",   hasPwd);
String user   = getStr("username",   hasU);
String img    = getStr("imageName",  hasI);
String room   = getStr("roomName",   hasR);
String key    = getStr("apiKey",     hasK);
preferences.begin("config", false);
if (hasSSID) preferences.putString("ssid",      ssid);
if (hasPwd)  preferences.putString("enc_pwd",   encPwd);
if (hasU)    preferences.putString("username",  user);
if (hasI)    preferences.putString("imageName", img);
if (hasR)    preferences.putString("roomName",  room);
if (hasK && key.length() >= 16 && key.length() <= 128) preferences.putString("api_key", key);
bool cfgReady = preferences.getString("ssid","").length() && preferences.getString("enc_pwd","").length();
preferences.putBool("configured", cfgReady);
preferences.end();
bleConfigured = cfgReady;
pChar->setValue(uniqueId.c_str());
```

```
pChar->notify();
if (hasSSID && hasPwd) {
String plain = decryptPassword(encPwd);
if (plain.length()) {
WiFi.mode(WIFI_STA); WiFi.disconnect(true,true); delay(200);
WiFi.begin(ssid.c_str(), plain.c_str());
}
nextSyncAt = millis() + 5000;
} else if (cfgReady && WiFi.status() != WL_CONNECTED) {
reconnectWithSavedWifi();
nextSyncAt = millis() + 5000;
}
}
};
class MyServerCallbacks : public BLEServerCallbacks {
void onConnect(BLEServer* s) override { }
void onDisconnect(BLEServer* s) override { s->startAdvertising(); }
};
void setup() {
Serial.begin(115200);
uint64_t chipid = ESP.getEfuseMac();
char id[13]; sprintf(id, "%04X%08X", (uint32_t)(chipid >> 32), (uint32_t)chipid);
```

```
uniqueId = String(id); bleName = "ESP32_" + uniqueId;
BLEDevice::init(bleName.c_str());
gServer = BLEDevice::createServer();
gServer->setCallbacks(new MyServerCallbacks());
BLEService *svc = gServer->createService(SERVICE_UUID);
pCharacteristic = svc->createCharacteristic(
CHARACTERISTIC_UUID,
BLECharacteristic::PROPERTY_WRITE                                    |
BLECharacteristic::PROPERTY_WRITE_NR                                 |
BLECharacteristic::PROPERTY_NOTIFY
);
pCharacteristic->setCallbacks(new MyCallbacks());
pCharacteristic->addDescriptor(new BLE2902());
svc->start();
BLEAdvertising *adv = BLEDevice::getAdvertising();
adv->addServiceUUID(SERVICE_UUID);
adv->setScanResponse(true);
adv->setMinPreferred(0x06);
gServer->startAdvertising();

BLESecurity *sec = new BLESecurity();
sec->setAuthenticationMode(ESP_LE_AUTH_REQ_SC_MITM_BOND);
sec->setCapability(ESP_IO_CAP_OUT);
```

```
sec->setInitEncryptionKey(ESP_BLE_ENC_KEY_MASK |
ESP_BLE_ID_KEY_MASK);
preferences.begin("config", true);
bleConfigured = preferences.getBool("configured", false);
preferences.end();
Wire.begin(I2C_SDA_PIN, I2C_SCL_PIN);
lcd.init(); lcd.backlight();
lcd.setCursor(0,0);
lcd.print(bleConfigured ? "Init sensors..." : "BLE config mode");
dht.begin();
pinMode(Smoke_PIN, INPUT);
pinMode(Light_PIN, INPUT);
delay(20000); // прогрів MQ-2
if (!bme.begin(0x76)) { bmeDetected = false; }
if (bleConfigured) reconnectWithSavedWifi();
}
void loop() {
checkWiFiConnection();
if (millis() - displayRefreshTime >= LCD_REFRESH_MS) {
float t = dht.readTemperature();
float h = dht.readHumidity();
int gas  = digitalRead(Smoke_PIN);
int lite = digitalRead(Light_PIN);
```

```
float p  = bme.readPressure() / 100.0F; // hPa

updateDisplay(t, h, gas, lite, p);

displayRefreshTime = millis();

}

if (WiFi.status() == WL_CONNECTED && bleConfigured && millis() >= nextSyncAt) {

int rc = syncMetadata();

scheduleNextSync(rc);

}

if (WiFi.status() == WL_CONNECTED && bleConfigured && (millis() - lastTime) > POST_INTERVAL_MS) {

// читання сенсорів

float tDht = dht.readTemperature();

float hDht = dht.readHumidity();

int gas    = digitalRead(Smoke_PIN);

int lite   = digitalRead(Light_PIN);

float tBme = bme.readTemperature();

float hBme = bme.readHumidity();

float pBme = bme.readPressure() / 100.0F;

float aBme = bme.readAltitude(SEALEVELPRESSURE_HPA);

int   mq2Val   = analogRead(MQ2_ANALOG_PIN);

int   lightVal = analogRead(LIGHT_ANALOG_PIN);

float mq2Pct   = mq2Val   * 100.0 / 4095.0;
```

```
float lightPct = 100.0 - (lightVal * 100.0 / 4095.0);

String json = "{";

json += "\"ChipId\":\"" + uniqueId + "\",";

json += "\"TemperatureDht\":"; json += isnan(tDht) ? "null" : String(tDht, 2); json += ",";

json += "\"HumidityDht\":";     json += isnan(hDht) ? "null" : String(hDht, 2); json += ",";

json += "\"TemperatureBme\":"; json += isnan(tBme) ? "null" : String(tBme, 2); json += ",";

json += "\"HumidityBme\":";    json += isnan(hBme) ? "null" : String(hBme, 2); json += ",";

json += "\"Pressure\":";       json += isnan(pBme) ? "null" : String(pBme, 2); json += ",";

json += "\"Altitude\":";       json += isnan(aBme) ? "null" : String(aBme, 2); json += ",";

json += "\"GasDetected\":";    json += (gas  == LOW ? "true" : "false"); json += ",";

json += "\"Light\":";          json += (lite == LOW ? "true" : "false"); json += ",";

json += "\"MQ2Analog\":"           + String(mq2Val)   + ",";

json += "\"MQ2AnalogPercent\":"    + String(mq2Pct, 2)+ ",";

json += "\"LightAnalog\":"         + String(lightVal) + ",";

json += "\"LightAnalogPercent\":"  + String(lightPct, 2);

json += "}";

HTTPClient http;

http.begin(String(API_BASE));            // ← саме BASE (без /sync)

http.addHeader("Content-Type", "application/json");
```

```
preferences.begin("config", true);
String apiKey = preferences.getString("api_key", "");
preferences.end();
if (apiKey.length()) http.addHeader("X-Api-Key", apiKey);
int code = http.POST(json);
http.end();
lastTime = millis();
}
}
```

Додаток Б

Лістинг коду контроллера DisplayData

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using MySensorApi.DTO;
using MySensorApi.DTO.Guests;
using MySensorApi.DTO.SensorData;
using MySensorApi.Services;
using MySensorApi.Infrastructure.Concurrency;
using MySensorApi.DTO.User;
namespace MySensorApi.Controllers
{
[Authorize]
[ApiController]
```

```
[Route("api/[controller]")]
public class DisplayDataController : BaseController
{
private readonly IOwnershipService _ownership;
public DisplayDataController(IOwnershipService ownership) => _ownership = ownership;
[HttpGet("byUser")]
[ProducesResponseType(typeof(IEnumerable<RoomWithSensorDto>), StatusCodes.Status200OK)]
public async Task<ActionResult<IEnumerable<RoomWithSensorDto>>> GetRoomsForUser(CancellationToken ct)
{
var userId = RequireUserIdOr401();
var items = await _ownership.GetRoomsForUserAsync(userId, ct);
return Ok(items);
}

[HttpGet("ownership/{chipId}/latest")]
[ProducesResponseType(typeof(RoomWithSensorDto), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<ActionResult<RoomWithSensorDto>> GetLatestByChip([FromRoute] string chipId, CancellationToken ct)
```

```
{
var userId = RequireUserIdOr401();
var dto = await _ownership.GetLatestForUserAsync(chipId, userId, ct);
if (dto is null)
{
var exists = await _ownership.ChipExistsAsync(chipId, ct);
return exists
? Forbid("Немає доступу до цього chipId.")
: NotFound();
}
return Ok(dto);
}
[HttpPost("ownership")]
[ProducesResponseType(typeof(RoomWithSensorDto), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
public async Task<IActionResult> CreateOwnership([FromBody] SensorOwnershipDto
dto, CancellationToken ct)
{
if (dto is null ||
string.IsNullOrWhiteSpace(dto.ChipId) ||
string.IsNullOrWhiteSpace(dto.RoomName) ||
string.IsNullOrWhiteSpace(dto.ImageName))
```

```
return BadRequest("Потрібні поля: ChipId, RoomName, ImageName.");
var ownerId = RequireUserIdOr401();
try
{
var room = await _ownership.CreateAsync(ownerId, dto, ct);
return CreatedAtAction(nameof(GetLatestByChip), new { chipId = room.ChipId }, room);
}
catch (InvalidOperationException ex) // chip зайнятий
{
return Conflict(new { message = ex.Message, dto.ChipId });
}
}
// Оновлення RoomName/ImageName (тільки власник, підтримка ETag через If-Match)
[HttpPut("ownership")]
[ProducesResponseType(typeof(object), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
[ProducesResponseType(StatusCodes.Status412PreconditionFailed)]
public async Task<IActionResult> UpdateOwnership(
[FromBody] SensorOwnershipDto dto,
```

```
[FromHeader(Name = "If-Match")] string? ifMatch,
CancellationToken ct)
{
if (dto is null || string.IsNullOrWhiteSpace(dto.ChipId))
return BadRequest("ChipId is required");
var ownerId = RequireUserIdOr401();
try
{
var (updated, newEtag) = await _ownership.UpdateAsync(ownerId, dto, ifMatch, ct);
if (updated && !string.IsNullOrEmpty(newEtag))
Response.Headers.ETag = newEtag;

return updated ? Ok(new { etag = newEtag }) : NoContent();
}
catch (KeyNotFoundException)
{
return Forbid("Лише власник може оновлювати кімнату.");
}
catch (PreconditionFailedException ex)
{
return StatusCode(StatusCodes.Status412PreconditionFailed, ex.Message);
}
```

```
}

[HttpDelete("ownership/{chipId}")]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status403Forbidden)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public async Task<IActionResult> DeleteOwnership([FromRoute] string chipId, CancellationToken ct)

{

var ownerId = RequireUserIdOr401();

try

{

await _ownership.DeleteAsync(chipId, ownerId, ct);

return NoContent();

}

catch (KeyNotFoundException)

{

return NotFound();

}

catch (UnauthorizedAccessException)

{

return Forbid("Лише власник може видаляти кімнату.");

}

}
```

```
[HttpPost("guest")]
[ProducesResponseType(typeof(GuestDto), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status409Conflict)]
public async Task<IActionResult> AddGuest([FromBody] AddGuestRequestDto req, CancellationToken ct)
{
if (req is null || string.IsNullOrWhiteSpace(req.ChipId) || string.IsNullOrWhiteSpace(req.Username))
return BadRequest("ChipId та Username обов'язкові");

var ownerId = RequireUserIdOr401();
try
{
var guest = await _ownership.AddGuestAsync(ownerId, req, ct);
return CreatedAtAction(
nameof(GetGuests),
new { chipId = req.ChipId },
guest
);
}
```

```
catch (KeyNotFoundException ex)
{
return NotFound(ex.Message);
}
catch (UnauthorizedAccessException ex)
{
return Forbid(ex.Message);
}
catch (InvalidOperationException ex)
{
return Conflict(new { message = ex.Message });
}
}
[HttpDelete("guest/{chipId}/{guestUserId:int}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> RemoveGuest([FromRoute] string chipId, [FromRoute] int guestUserId, CancellationToken ct)
{
var ownerId = RequireUserIdOr401();
try
{
```

```
await _ownership.RemoveGuestAsync(ownerId, chipId, guestUserId, ct);
return NoContent();
}
catch (KeyNotFoundException)
{
return NotFound("ChipId не знайдено або гість відсутній.");
}
catch (UnauthorizedAccessException)
{
return Forbid("Лише власник може видаляти гостей.");
}
}

[HttpDelete("guest/self/{chipId}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> RemoveSelfGuest([FromRoute] string chipId, CancellationToken ct)
{
var userId = RequireUserIdOr401();
var removed = await _ownership.RemoveSelfFromGuestsAsync(userId, chipId, ct);
if (!removed)
return NotFound("Ви не є гостем цієї плати.");
```

```
return NoContent();
}
[HttpGet("guest/{chipId}")]
[ProducesResponseType(typeof(IEnumerable<GuestDto>), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
public async Task<ActionResult<IEnumerable<GuestDto>>> GetGuests([FromRoute] string chipId, CancellationToken ct)
{
var ownerId = RequireUserIdOr401();
try
{
var guests = await _ownership.GetGuestsAsync(ownerId, chipId, ct);
return Ok(guests);
}
catch (UnauthorizedAccessException)
{
return Forbid("Лише власник може переглядати список гостей.");
}
}
[HttpPost("{chipId}/invite")]
[ProducesResponseType(typeof(GuestInviteDto), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
```

```
public async Task<ActionResult<GuestInviteDto>> CreateInvite([FromRoute] string chipId, CancellationToken ct)
{
var ownerId = RequireUserIdOr401();
try
{
var dto = await _ownership.CreateInviteAsync(ownerId, chipId, ct);
return Ok(dto);
}
catch (UnauthorizedAccessException ex)
{
return Forbid(ex.Message);
}
}
[HttpPost("join")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
public async Task<IActionResult> JoinByToken([FromBody] GuestJoinRequestDto req, CancellationToken ct)
{
if (req is null || string.IsNullOrWhiteSpace(req.Token))
```

```
return BadRequest("Потрібен токен запрошення.");
var userId = RequireUserIdOr401();
try
{
await _ownership.JoinByTokenAsync(userId, req, ct);
return Ok();
}
catch (UnauthorizedAccessException ex)
{
// токен взагалі не знайдено → 401
return Unauthorized(new { error = ex.Message });
}
catch (InvalidOperationException ex)
{
// прострочений / вже використаний → 400
return BadRequest(new { error = ex.Message });
}
}
[HttpPost("ownership/{chipId}/revokeInvites")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status403Forbidden)]
public async Task<IActionResult> RevokeInvites([FromRoute] string chipId,
CancellationToken ct)
```

```
{
var ownerId = RequireUserIdOr401();
try
{
await _ownership.RevokeInvitesAsync(ownerId, chipId, ct);
return NoContent();
}
catch (UnauthorizedAccessException ex)
{
return Forbid(ex.Message);
}
}
}
}
```

Додаток В

Лістинг коду сервісу DisplayData

```
using Microsoft.EntityFrameworkCore;
using MySensorApi.Data;
using MySensorApi.Domain; // для ChipId.Normalize
using MySensorApi.DTO;
using MySensorApi.DTO.Guests;
using MySensorApi.DTO.SensorData;
using MySensorApi.DTO.User;
```

```
using MySensorApi.Infrastructure.Concurrency;

using MySensorApi.Infrastructure.Repositories.Interfaces;

using MySensorApi.Models;

using MySensorApi.Services.Utils;

namespace MySensorApi.Services

{

public interface IOwnershipService

{

Task<List<RoomWithSensorDto>> GetRoomsForUserAsync(int userId, CancellationToken ct = default);

Task<RoomWithSensorDto?> GetLatestForUserAsync(string chipId, int userId, CancellationToken ct = default);


Task<RoomWithSensorDto> CreateAsync(int ownerId, SensorOwnershipDto dto, CancellationToken ct = default);

Task<(bool updated, string? newEtag)> UpdateAsync(int ownerId, SensorOwnershipDto dto, string? ifMatch, CancellationToken ct = default);

Task DeleteAsync(string chipId, int ownerId, CancellationToken ct = default);


Task<bool> HasAccessAsync(string chipId, int userId, CancellationToken ct = default);

Task<bool> ChipExistsAsync(string chipId, CancellationToken ct = default);

Task<GuestDto> AddGuestAsync(int ownerId, AddGuestRequestDto req, CancellationToken ct = default);
```

```
Task RemoveGuestAsync(int ownerId, string chipId, int guestUserId, CancellationToken ct = default);

Task<bool> RemoveSelfFromGuestsAsync(int userId, string chipId, CancellationToken ct = default);

Task<IEnumerable<GuestDto>> GetGuestsAsync(int ownerId, string chipId, CancellationToken ct = default);

Task<GuestInviteDto> CreateInviteAsync(int ownerId, string chipId, CancellationToken ct = default);

Task JoinByTokenAsync(int userId, GuestJoinRequestDto dto, CancellationToken ct = default);

Task RevokeInvitesAsync(int ownerId, string chipId, CancellationToken ct = default);


Task<(OwnershipSyncDto? dto, string? etag, DateTime? lastModified)> GetSyncForEspAsync(string chipId, CancellationToken ct = default);

}

public sealed class OwnershipService : IOwnershipService

{

private readonly IOwnershipRepository _own;

private readonly IGuestRepository _guest;

private readonly ISensorDataRepository _data;

private readonly AppDbContext _db;

public OwnershipService(

IOwnershipRepository own,

IGuestRepository guest,
```

```
ISensorDataRepository data,
AppDbContext db)
{
_own = own;
_guest = guest;
_data = data;
_db = db;
}
public async Task<IEnumerable<GuestDto>> GetGuestsAsync(int ownerId, string chipId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var ownership = await _own.GetByChipForOwnerAsync(norm, ownerId, ct);
if (ownership == null)
throw new UnauthorizedAccessException("Лише власник може переглядати гостей.");
var guests = await _guest.GetByChipAsync(norm, ct);
return guests.Select(g => new GuestDto
{
UserId = g.UserId,
Username = g.User?.Username ?? "(невідомо)",
AddedAt = g.CreatedAt
});
```

```
}
public async Task<GuestDto> AddGuestAsync(int ownerId, AddGuestRequestDto req, CancellationToken ct = default)
{
var norm = ChipId.Normalize(req.ChipId);
var o = await _own.GetByChipAsync(norm, ct);
if (o is null)
throw new KeyNotFoundException("ChipId не знайдено.");
if (o.OwnerId != ownerId)
throw new UnauthorizedAccessException("Лише власник може додавати гостей.");
var user = await _db.Users.FirstOrDefaultAsync(u => u.Username == req.Username, ct);
if (user is null)
throw new KeyNotFoundException($"Користувач '{req.Username}' не знайдений.");
if (user.Id == ownerId)
throw new InvalidOperationException("Власник не може додати самого себе у гості.");
if (await _guest.ExistsAsync(norm, user.Id, ct))
throw new InvalidOperationException("Користувач уже є гостем цієї плати.");
var entity = new SensorGuest
{
ChipId = norm,
UserId = user.Id,
```

```
CreatedAt = DateTime.UtcNow
};
await _guest.AddGuestAsync(entity, ct);
await _guest.SaveChangesAsync(ct);
return new GuestDto
{
UserId = user.Id,
Username = user.Username,
AddedAt = entity.CreatedAt
};
}
public async Task RemoveGuestAsync(int ownerId, string chipId, int guestUserId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var o = await _own.GetByChipAsync(norm, ct);
if (o is null)
throw new KeyNotFoundException("ChipId не знайдено.");
if (o.OwnerId != ownerId)
throw new UnauthorizedAccessException("Лише власник може видаляти гостей.");
await _guest.RemoveGuestAsync(norm, guestUserId, ct);
await _guest.SaveChangesAsync(ct);
}
```

```
public async Task<bool> RemoveSelfFromGuestsAsync(int userId, string chipId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var isGuest = await _guest.IsGuestAsync(norm, userId, ct);
if (!isGuest) return false;
await _guest.RemoveGuestAsync(norm, userId, ct);
await _guest.SaveChangesAsync(ct);
return true;
}
public async Task<GuestInviteDto> CreateInviteAsync(int ownerId, string chipId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var o = await _own.GetByChipForOwnerAsync(norm, ownerId, ct);
if (o == null)
throw new UnauthorizedAccessException("Лише власник може створювати запрошення.");
await _guest.DeleteInvitesByChipAsync(norm, ct);
var token = Guid.NewGuid().ToString("N");
var invite = new GuestInvite
{
ChipId = norm,
```

```
OwnerId = ownerId,
Token = token,
ExpiresAt = DateTime.UtcNow.AddHours(24)
};
await _guest.AddInviteAsync(invite, ct);
await _guest.SaveChangesAsync(ct);
return new GuestInviteDto { Token = token, ExpiresAt = invite.ExpiresAt };
}

public async Task JoinByTokenAsync(int userId, GuestJoinRequestDto dto, CancellationToken ct = default)
{
if (string.IsNullOrWhiteSpace(dto?.Token))
throw new InvalidOperationException("Порожній токен.");
var invite = await _guest.GetInviteByTokenAsync(dto.Token, ct);
if (invite == null || invite.ExpiresAt < DateTime.UtcNow)
throw new UnauthorizedAccessException("Недійсне або прострочене запрошення.");
if (await _guest.ExistsAsync(invite.ChipId, userId, ct))
return;
var entity = new SensorGuest
{
ChipId = invite.ChipId,
UserId = userId,
```

```
CreatedAt = DateTime.UtcNow
};
await _guest.AddGuestAsync(entity, ct);
await _guest.SaveChangesAsync(ct);
}
public async Task RevokeInvitesAsync(int ownerId, string chipId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var o = await _own.GetByChipForOwnerAsync(norm, ownerId, ct);
if (o == null)
throw new UnauthorizedAccessException("Лише власник може скасувати запрошення.");
await _guest.DeleteInvitesByChipAsync(norm, ct); // цей метод додаєш у репозиторій гостей
await _guest.SaveChangesAsync(ct);
}
public async Task<RoomWithSensorDto> CreateAsync(int ownerId, SensorOwnershipDto dto, CancellationToken ct = default)
{
if (string.IsNullOrWhiteSpace(dto.ChipId) ||
string.IsNullOrWhiteSpace(dto.RoomName) ||
string.IsNullOrWhiteSpace(dto.ImageName))
```

```
throw new InvalidOperationException("Потрібні поля: ChipId, RoomName, ImageName");

var norm = ChipId.Normalize(dto.ChipId);

if (await _own.ExistsChipAsync(norm, ct))

throw new InvalidOperationException("Ця плата вже має власника (chipId зайнятий).");

var o = new SensorOwnership

{

OwnerId = ownerId,

ChipId = norm,

RoomName = dto.RoomName.Trim(),

ImageName = dto.ImageName.Trim(),

Version = 1,

UpdatedAt = DateTime.UtcNow

};

await _own.AddAsync(o, ct);

try

{

await _own.SaveChangesAsync(ct);

}

catch (DbUpdateException)

{
```

```
throw new InvalidOperationException("Ця плата вже має власника (chipId зайнятий).");
}
var latest = await _data.GetLatestByChipIdAsync(norm, ct);
return Map(o, latest, isOwner: true);
}
public async Task<List<RoomWithSensorDto>> GetRoomsForUserAsync(int userId, CancellationToken ct = default)
{
var owned = await _own.GetOwnedByAsync(userId, ct);
var guestLinks = await _guest.GetByUserAsync(userId, ct);
var guestOwnerships = guestLinks
.Where(g => g.Ownership != null && g.Ownership.OwnerId != userId)
.Select(g => g.Ownership!)
.GroupBy(o => o.ChipId)
.Select(g => g.First())
.ToList();
var result = new List<RoomWithSensorDto>();
foreach (var o in owned)
{
SensorData? latest = null;
try { latest = await _data.GetLatestByChipIdAsync(o.ChipId, ct); }
catch (TaskCanceledException) { }
```

```
var guestCount = await _guest.CountByChipAsync(o.ChipId, ct);
result.Add(new RoomWithSensorDto
{
Id = o.Id,
ChipId = o.ChipId,
RoomName = o.RoomName,
ImageName = o.ImageName,
Temperature = latest?.TemperatureDht,
Humidity = latest?.HumidityDht,
Pressure = latest?.Pressure,
IsOwner = true,
GuestCount = guestCount
});
}
foreach (var o in guestOwnerships)
{
SensorData? latest = null;
try { latest = await _data.GetLatestByChipIdAsync(o.ChipId, ct); }
catch (TaskCanceledException) { }

result.Add(new RoomWithSensorDto
{
```

```
Id = o.Id,
ChipId = o.ChipId,
RoomName = o.RoomName,
ImageName = o.ImageName,
Temperature = latest?.TemperatureDht,
Humidity = latest?.HumidityDht,
Pressure = latest?.Pressure,
IsOwner = false,
GuestCount = 0
});
}
return result;
}
public async Task<RoomWithSensorDto?> GetLatestForUserAsync(string chipId, int userId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var ownership = await _own.GetByChipAsync(norm, ct);
if (ownership is null) return null;
var allowed = ownership.OwnerId == userId || await _guest.IsGuestAsync(norm, userId, ct);
if (!allowed) return null;
var latest = await _data.GetLatestByChipIdAsync(norm, ct);
```

```
return Map(ownership, latest, ownership.OwnerId == userId);

}

public async Task<(bool updated, string? newEtag)> UpdateAsync(int ownerId, SensorOwnershipDto dto, string? ifMatch, CancellationToken ct = default)

{

if (string.IsNullOrWhiteSpace(dto.ChipId))

throw new InvalidOperationException("ChipId обов'язковий.");

var norm = ChipId.Normalize(dto.ChipId);

var o = await _own.GetByChipForOwnerAsync(norm, ownerId, ct);

if (o is null) throw new KeyNotFoundException("Цей chipId вам не належить.");


var currentEtag = $"\"{o.Id}-{o.Version}\"";

if (!string.IsNullOrEmpty(ifMatch) && !string.Equals(ifMatch, currentEtag, StringComparison.Ordinal))

throw new PreconditionFailedException("If-Match не збігається з поточним ETag.");

var changed = false;

if (!string.IsNullOrWhiteSpace(dto.RoomName) && dto.RoomName.Trim() != o.RoomName)

{

o.RoomName = dto.RoomName.Trim();

changed = true;

}
```

```
if (!string.IsNullOrWhiteSpace(dto.ImageName) && dto.ImageName.Trim() != o.ImageName)
{
o.ImageName = dto.ImageName.Trim();
changed = true;
}
if (!changed) return (false, currentEtag);
o.Version++;
o.UpdatedAt = DateTime.UtcNow;
await _own.SaveChangesAsync(ct);

return (true, $"\"{o.Id}-{o.Version}\"");
}
public async Task DeleteAsync(string chipId, int ownerId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var o = await _own.GetByChipAsync(norm, ct);
if (o is null) throw new KeyNotFoundException("ChipId не знайдено.");
if (o.OwnerId != ownerId) throw new UnauthorizedAccessException("Лише власник може видаляти.");
_own.Remove(o);
await _own.SaveChangesAsync(ct);
```

```
}
public async Task<bool> HasAccessAsync(string chipId, int userId, CancellationToken ct = default)
{
var norm = ChipId.Normalize(chipId);
var o = await _own.GetByChipAsync(norm, ct);
if (o is null) return false;
if (o.OwnerId == userId) return true;
return await _guest.IsGuestAsync(norm, userId, ct);
}

public Task<bool> ChipExistsAsync(string chipId, CancellationToken ct = default) =>
_own.ExistsChipAsync(ChipId.Normalize(chipId), ct);
public async Task<(OwnershipSyncDto? dto, string? etag, DateTime? lastModified)> GetSyncForEspAsync(string chipId, CancellationToken ct = default)
{
var o = await _own.GetByChipAsync(ChipId.Normalize(chipId), ct);
if (o is null) return (null, null, null);
var etag = $"\"{o.Id}-{o.Version}\"";
var lastModified = o.UpdatedAt.ToUniversalTime();
var dto = new OwnershipSyncDto
{
Username = o.Owner?.Username ?? string.Empty,
```

```
RoomName = o.RoomName ?? string.Empty,
ImageName = o.ImageName ?? string.Empty
};
return (dto, etag, lastModified);
}
private static RoomWithSensorDto Map(SensorOwnership o, SensorData? s, bool isOwner) => new RoomWithSensorDto
{
Id = o.Id,
ChipId = o.ChipId,
RoomName = o.RoomName,
ImageName = o.ImageName,
Temperature = s?.TemperatureDht,
Humidity = s?.HumidityDht,
Pressure = s?.Pressure,
IsOwner = isOwner,
GuestCount = o.Guests?.Count ?? 0
};
}
public sealed class PreconditionFailedException : Exception
{
public PreconditionFailedException(string message) : base(message) { }
}
```

```
}
```

Додаток Г

Лістинг коду репозиторію DisplayData

```
using Microsoft.EntityFrameworkCore;
using MySensorApi.Data;
using MySensorApi.Infrastructure.Repositories.Interfaces;
using MySensorApi.Models;
namespace MySensorApi.Infrastructure.Repositories
{
public sealed class OwnershipRepository : IOwnershipRepository
{
private readonly AppDbContext _db;
public OwnershipRepository(AppDbContext db) => _db = db;
public Task<SensorOwnership?> GetByChipAsync(string chipId, CancellationToken ct = default) =>
_db.SensorOwnerships
.Include(o => o.Owner)
.Include(o => o.Guests)
.AsNoTracking()
.FirstOrDefaultAsync(o => o.ChipId == chipId, ct);
public Task<SensorOwnership?> GetByChipForOwnerAsync(string chipId, int ownerId, CancellationToken ct = default) =>
_db.SensorOwnerships
```

```
.Include(o => o.Guests)

.FirstOrDefaultAsync(o => o.ChipId == chipId && o.OwnerId == ownerId, ct);

public Task<List<SensorOwnership>> GetOwnedByAsync(int ownerId, CancellationToken ct = default) =>

_db.SensorOwnerships

.AsNoTracking()

.Where(o => o.OwnerId == ownerId)

.ToListAsync(ct);

public Task<bool> ExistsChipAsync(string chipId, CancellationToken ct = default) =>

_db.SensorOwnerships.AsNoTracking().AnyAsync(o => o.ChipId == chipId, ct);

public Task AddAsync(SensorOwnership entity, CancellationToken ct = default) =>

_db.SensorOwnerships.AddAsync(entity, ct).AsTask();

public void Remove(SensorOwnership entity) => _db.SensorOwnerships.Remove(entity);

public Task<int> SaveChangesAsync(CancellationToken ct = default) => _db.SaveChangesAsync(ct);

}

}
```

Додаток Д

Вигляд основних сторінок додатку SenseData