Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та безпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

TELEGRAM-БОТ ДЛЯ МОНІТОРИНГУ ТЕМПЕРАТУРИ ТА ВОЛОГОСТІ З ВИКОРИСТАННЯМ МІКРОКОНТРОЛЕРУ ESP8266

TELEGRAM BOT FOR MONITORING TEMPERATURE AND HUMIDITY USING ESP8266 MICROCONTROLLER

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

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-42
Кондратюк Назарій Віталійович

(підпис)

Керівник:
к.т.н., доцент
Гринюк Сергій Васильович

(підпис)

Кваліфікаційну роботу
допущено до захисту
« 10 » червня 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

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

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Тарас ТЕРЛЕЦЬКИЙ
« 10 » 01 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Кондратюку Назарію Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Telegram-бот для моніторингу температури та вологості з використанням мікроконтролеру ESP8266*

Керівник роботи *к.т.н., доц. Гринюк Сергій Васильович*

затверджені наказом закладу вищої освіти від «04» січня 2025 року № 11/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 10.06.2025р.

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

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Загальні відомості про інтернет речей
Проєктування структури розробки
Розробка та тестування IoT системи
Висновки

5. Перелік графічного (ілюстративного) матеріалу:
Зображення мікросхем
Функціонально-структурна схема системи
Схеми моделей бази даних
Скріншоти інтерфейсів користувача
Лістинг коду

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Загальні відомості про інтернет речей	*Гринюк С.В., доцент*		
Проєктування структури розробки	*Гринюк С.В., доцент*		
Розробка та тестування IoT системи	*Гринюк С.В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Лавренчук С.В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О.І.,ст.викладач*		

7. Дата видачі завдання 10.01.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми, аналіз предметної області та наявних рішень*	до 05.02.2025 р.	Виконано
2.	*Вибір апаратної та програмної бази для проєкту*	до 28.02.2025 р.	Виконано
3.	*Аналіз та розробка системи моніторингу температури*	до 28.03.2025 р.	Виконано
4.	*Висновки та пропозиції*	до 05.04.2025 р.	Виконано
5.	*Формування списку використаних джерел*	до 10.04.2025 р.	Виконано
6.	*Формування додатків*	до 28.04.2025 р.	Виконано
7.	*Оформлення ілюстративного матеріалу*	до 09.05.2025 р.	Виконано
8.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 13.05.2025 р.	Виконано
9.	*Нормоконтроль*	до 29.05.2025 р.	Виконано
10	*Інструментальна перевірка на академічний плагіат*	до 04.06.2025 р	Виконано
11.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 10.06.2025 р.	Виконано

Здобувач вищої освіти _____ Кондратюк Н.В.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Гринюк С.В.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Кондратюк Н.В. Telegram-бот для моніторингу температури та вологості з використанням мікроконтролеру ESP8266. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп’ютерна інженерія. Луцький національний технічний університет. Луцьк, 2025.

Робота присвячена розробці IoT-системи моніторингу мікрокліматичних параметрів з використанням мікроконтролерної платформи ESP8266. Система забезпечує збір, зберігання та візуалізацію температурно-вологісних показників у реальному часі з можливістю сповіщення через Telegram-бот.

Перший розділ висвітлює теоретичні основи Інтернету речей: архітектуру IoT-систем, особливості апаратних платформ на базі мікроконтролерів, принципи взаємодії між embedded-пристроями та хмарними сервісами, та порівняльний аналіз технічних характеристик популярних мікроконтролерів.

Другий розділ містить обґрунтування вибору технологічного стеку: мікроконтролера ESP8266 з сенсором DHT11, серверної частини на Node.js з використанням Supabase та клієнтського інтерфейсу у вигляді Telegram-бота. Наведено схему бази даних з нормалізованими сутностями для зберігання оперативних та історичних даних.

Третій розділ описує практичну реалізацію системи: прошивку мікроконтролера для циклічного збору даних, REST API сервер для обробки запитів, Telegram-бот з інтерактивним меню. Особливу увагу приділено механізмам відмовостійкості: транзакційним операціям у базі даних, автоматичній повторній передачі даних та кешуванню станів.

Ключові слова: ESP8266, DHT11, Node.js, Telegram Bot API, Supabase, IoT-моніторинг.

ANNOTATION

Kondratiuk N. Telegram bot for monitoring temperature and humidity using ESP8266 microcontroller. Manuscript.

Bachelor's qualification work in the educational program "Computer Engineering" of the specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2025.

The work is dedicated to the development of an IoT-based system for monitoring microclimatic parameters using the ESP8266 microcontroller platform. The system provides real-time collection, storage, and visualization of temperature and humidity data with the capability of sending notifications via a Telegram bot.

The first chapter outlines the theoretical foundations of the Internet of Things, including the architecture of IoT systems, features of hardware platforms based on microcontrollers, and principles of interaction between embedded devices and cloud services. A comparative analysis of technical specifications of popular microcontrollers is presented.

The second chapter justifies the choice of the technological stack: the ESP8266 microcontroller with a DHT11 sensor, a backend based on Node.js using Supabase, and a client interface implemented as a Telegram bot. A database schema with normalized entities for storing real-time and historical data is provided.

The third chapter describes the practical implementation of the system, including firmware development for the microcontroller to perform cyclic data collection, a REST API server to handle requests, and a Telegram bot with an interactive menu. Special attention is given to fault-tolerance mechanisms such as transactional operations in the database, automatic data retransmission, and state caching.

Keywords: ESP8266, DHT11, Node.js, Telegram Bot API, Supabase, IoT monitoring.

ЗМІСТ

ВСТУП

Актуальність теми. Моніторинг мікрокліматичних параметрів набуває ключового значення в умовах зростання вимог до якості життя та оптимізації виробничих процесів. Інтеграція IoT-технологій дозволяє створювати масштабовані системи реального часу, що особливо актуально для сільського господарства, медицини, логістики та розумних міст.

Метою роботи є розробка енергоефективної IoT-системи моніторингу температури та вологості з використанням мікроконтролерної платформи ESP8266, хмарної бази даних Supabase та Telegram-бота для інтерактивної візуалізації даних.

Об’єкт дослідження – процеси збору, передачі та аналізу сенсорних даних в IoT-екосистемах.

Предмет дослідження – програмно-апаратна система моніторингу на базі ESP8266 з інтеграцією у хмарну інфраструктуру.

Завдання:

– проаналізувати існуючі рішення в галузі IoT-моніторингу мікроклімату та обґрунтувати вибір апаратної платформи та програмного стеку;

– розробити прошивку мікроконтролера для циклічного збору даних з сенсора DHT11;

– реалізувати серверну частину з механізмами транзакційного зберігання даних;

– створити Telegram-бота з інтерактивним інтерфейсом для відображення даних;

– провести комплексне тестування системи.

Структура й обсяг роботи. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел із 21 найменування. Повний обсяг роботи становить 40 сторінок, 26 рисунків та 2 таблиці.

РОЗДІЛ 1
ЗАГАЛЬНІ ВІДОМОСТІ ПРО ІНТЕРНЕТ РЕЧЕЙ

1.1 Основні поняття та сфери застосування IoT систем

Інтернет речей – це концепція, що передбачає створення мережі взаємопов'язаних пристроїв, які здатні збирати, передавати та обмінюватися даними між собою, а також з хмарними системами. Такі пристрої зазвичай оснащені вбудованими технологіями – сенсорами, мікропроцесорами, програмним забезпеченням і засобами зв'язку, та можуть охоплювати як побутові об'єкти, так і складне промислове обладнання. IoT активно впроваджується в різних галузях для підвищення ефективності діяльності, покращення якості обслуговування клієнтів, підтримки прийняття рішень і зростання бізнес-цінності [1].

Технологія IoT забезпечує передачу даних мережею без необхідності прямої участі людини, як у вигляді взаємодії між людьми, так і між людиною та комп'ютером. Під «річчю» в контексті IoT розуміється будь-який фізичний або біологічний об'єкт, що має унікальну IP-адресу та здатний передавати дані. Це може бути як людина з імплантованим кардіомонітором, тварина з біочипом, так і автомобіль, обладнаний сенсорами для виявлення зниження тиску в шинах.

Функціонування IoT-системи базується на зборі даних сенсорами, інтегрованими в пристрої. Ці дані передаються через шлюз IoT до прикладного програмного забезпечення або серверної частини для подальшої обробки й аналізу. У типовій екосистемі IoT вирізняють чотири ключові компоненти: сенсори або пристрої для збору даних; мережеву інфраструктуру для забезпечення зв'язку між пристроями та шлюзами; системи аналітики для обробки отриманої інформації; а також графічний інтерфейс користувача, що забезпечує керування та моніторинг пристроїв через веб-інтерфейс або мобільний додаток [2].

У процесі аналізу даних використовуються лише релевантні показники, що дозволяє виявляти закономірності, формувати рекомендації й попереджати

потенційні збої ще до їх виникнення. Крім того, локальна обробка інформації дозволяє зменшити обсяг переданих до хмари даних, що оптимізує споживання пропускної здатності мережі. Пристрої можуть як взаємодіяти між собою без участі людини, так і бути налаштовані або керовані користувачем. Вибір протоколів зв'язку та типів мереж залежить від конкретного сценарію використання.

У сучасних умовах інтернет речей стає невід'ємним елементом цифрової трансформації, особливо в таких секторах як виробництво, транспорт, енергетика, сільське господарство та інфраструктура (рис. 1.1). IoT сприяє стратегічному переосмисленню підходів до управління бізнесом, надаючи організаціям інструменти для оптимізації діяльності та підвищення конкурентоспроможності.

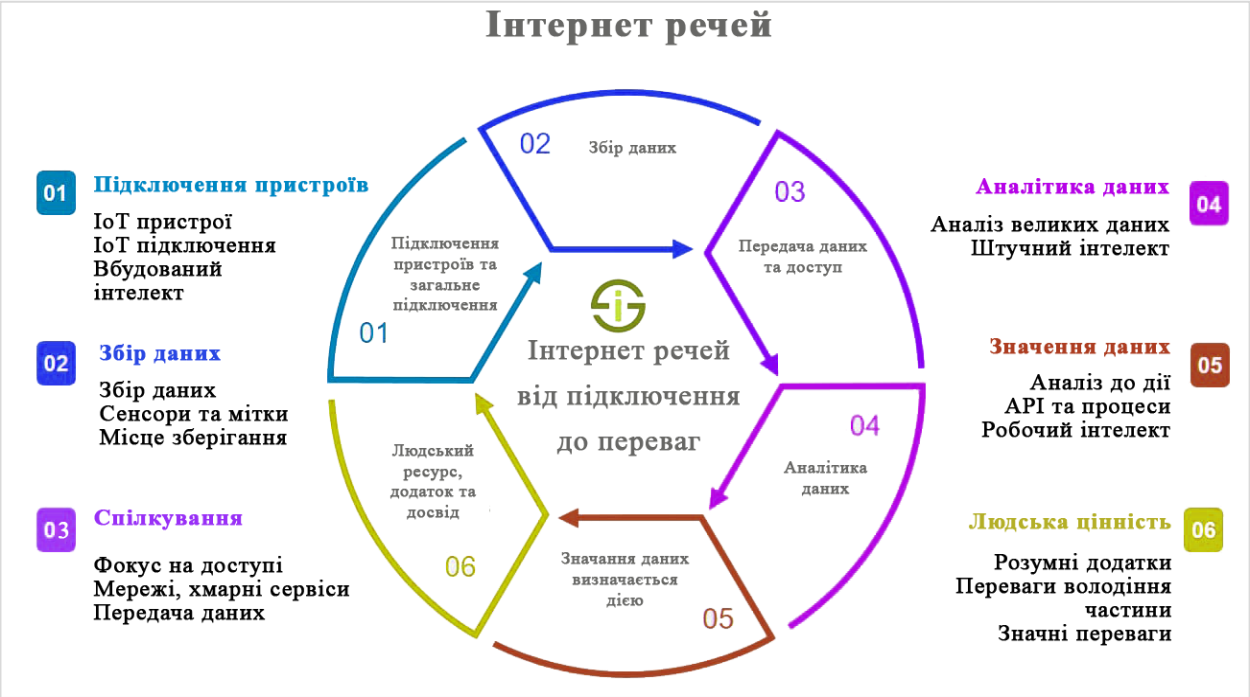

Рисунок 1.1 – Сфери застосування інтернету речей [3]

Інтернет речей також інтегрується з технологіями штучного інтелекту та машинного навчання, що сприяє автоматизації процесів збору та аналізу даних [4]. Рішення на основі IoT дозволяють не лише автоматизувати побут, наприклад, керування освітленням, температурою або доступом у помешканні, але й

трансформують бізнес-процеси завдяки глибокій аналітиці та моніторингу стану систем у режимі реального часу. IoT сприяє зниженню витрат на працю, зменшенню втрат і покращенню якості обслуговування. Це забезпечує більшу прозорість у взаємодії з клієнтами та ефективніше використання ресурсів.

1.2 Огляд апаратної платформи

У контексті Інтернету речей (IoT) та інтелектуальних об’єктів обчислювальні функції здебільшого забезпечуються за допомогою мікроконтролерів. Мікроконтролери – це, по суті, спрощені комп’ютери, що забезпечують функціонування смарт-пристроїв шляхом надання обчислювальних ресурсів, пам’яті та периферійних інтерфейсів введення/виведення [5].

Оскільки більшість пристроїв IoT мають бути компактними та енергоощадними, особливо у випадках ресурсно обмежених систем, які працюють автономно на віддаленні від центральних вузлів та живляться від батарей з низьким енергоспоживанням, застосування повноцінних процесорів тут недоцільне. Саме тому у таких пристроях використовуються мікроконтролери, які є більш оптимальними з огляду на їхню енергоефективність і простоту інтеграції [6].

Розуміння ролі мікроконтролерів у структурі IoT передбачає ознайомлення з архітектурою технологічного стеку вбудованого пристрою. Наприклад, у спрощеній моделі IoT-пристрою, як-от смарт-камера, можна виокремити такі шари: шар протоколів зв’язку, шар апаратної абстракції (HAL), а також рівень операційної системи (RTOS/OS). MCU функціонує на рівні апаратної абстракції, забезпечуючи взаємодію між іншими шарами та виконуючи операційну систему, що управляє пристроєм [7].

Важливо розрізняти мікроконтролери та мікропроцесори. Мікропроцесор – це інтегральна схема, яка містить лише центральний процесорний блок (CPU), але не має вбудованої оперативної або постійної

пам’яті, а також не включає периферійні компоненти; їхнє підключення здійснюється через зовнішні інтерфейси. Натомість мікроконтролер об’єднує на одному чипі CPU, оперативну пам’ять, постійну пам’ять і периферію, що робить його повноцінною автономною обчислювальною одиницею [8]. Хоча мікроконтролери поступаються мікропроцесорам у продуктивності, вони є більш доцільними для IoT-пристроїв, де важливими є малі розміри, низьке енергоспоживання та мінімізація вартості.

Під час розробки апаратного забезпечення для IoT необхідно враховувати низку критично важливих чинників. Перш за все, це енергоефективність та тривалість роботи від батареї. Більшість IoT-пристроїв працюють автономно, тому оптимізація енергоспоживання через вибір малопотужних компонентів, ефективне управління живленням та оптимізацію програмного коду є необхідною умовою для подовження строку служби пристроїв.

Розміри й форм-фактор також відіграють суттєву роль, адже пристрій має бути ергономічним і компактним, водночас зберігаючи всі необхідні функціональні компоненти. Вибір відповідних варіантів зв’язку – ще один ключовий аспект. Потрібно враховувати дальність дії, енергоспоживання, пропускну здатність, мережеву архітектуру, масштабованість і безпеку [9]. Наприклад, BLE або LoRaWAN підходять для пристроїв з низьким енергоспоживанням, тоді як Wi-Fi забезпечує високу швидкість передачі даних, що є необхідним для мультимедійного контенту. Протоколи на кшталт Zigbee або Thread підтримують mesh-архітектуру, яка дозволяє розширити зону покриття і підвищити надійність зв’язку. NB-IoT забезпечує масштабованість для масового підключення пристроїв, що важливо в міських і промислових середовищах. Крім того, варто враховувати вартість реалізації протоколу, вимоги до регуляторного контролю в різних регіонах, а також ступінь сумісності з іншими пристроями й сервісами.

Безпека є критичним компонентом IoT, тому необхідно впроваджувати заходи захисту, зокрема шифрування, захищене завантаження та виявлення несанкціонованого втручання. Масштабованість та довгострокова підтримка

пристроїв також мають враховуватися. Використання модульної архітектури, можливість оновлення ПЗ через мережу та вибір компонентів із тривалим життєвим циклом сприяють майбутній адаптації пристрою до нових технологічних вимог [10].

Ще одним важливим критерієм є масштабованість. Проєктування апаратного забезпечення для IoT з урахуванням масштабованості дозволяє пристроям адаптуватися до майбутніх вимог і технологічного розвитку. Це може передбачати використання модульної архітектури, підтримку оновлень програмного забезпечення «по повітрю» (over-the-air, OTA), а також вибір компонентів із тривалим терміном доступності на ринку.

1.3 Порівняння та вибір засобів розробки

Архітектурний вибір технологічного стеку обґрунтовано специфікою задач IoT-моніторингу, де критичними є вимоги до енергоефективності, низької затримки передачі даних та інтеграції з legacy-обладнанням. Система побудована на гетерогенній архітектурі, що поєднує embedded-пристрій (ESP8266), хмарний бекенд (Node.js + Supabase) та клієнтський інтерфейс (Telegram Bot API).

Вибір технологій для системи моніторингу ґрунтується на принципах cost-efficient архітектури, де кожен компонент оптимізований під специфічні задачі. ESP8266 як апаратна платформа, фактично, стандартом для IoT-проєктів бюджетного рівня [11]. Його популярність забезпечує розвинену екосистему бібліотек, зокрема Arduino Core для DHT11, що спрощує розробку порівняно з STM32. ESP8266 обрано як оптимальний компроміс між продуктивністю та енергоспоживанням.

Порівняльний аналіз з альтернативами наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз EI38266 з альтернативами

Пристрій	Споживання	Частота / Wi-Fi	GPIO / Ціна
ESP8266	~70 мА (до 250 мА при передачі)	80–160 МГц / Wi-Fi 802.11 b/g/n	11 GPIO / ~$2.5
STM32F103C8T6	~40 мА при 72 МГц	72 МГц / Wi-Fi відсутній	37 GPIO / ~$8.9

Продовження таблиці 1.1

Пристрій	Споживання	Частота / Wi-Fi	GPIO / Ціна
Raspberry Pi Zero W	~100–170 мА	1000 МГц / Wi-Fi 802.11n	26 GPIO / ~$15

Використання Arduino Core для ESP8266 дозволяє забезпечити апаратну абстракцію через DigitalIO API для роботи з DHT11, оптимізацію HTTP-клієнта з підтримкою Keep-Alive з'єднань, та механізми energy saving через Deep Sleep Mode.

Вибір JavaScript як базової мови для серверної частини обумовлений фундаментальними відмінностями в архітектурі виконання кодів. Дослідження проведені на основі тестування TPS (Transactions Per Second) демонструють, що Node.js (JavaScript) забезпечує на 72-85 % вищу пропускну здатність при обробці паралельних запитів порівняно з CPython 3.11 у сценаріях з IO-bound операціями [12]. Ця перевага виникає через event-loop механізм, що реалізує non-blocking I/O модель, критичну для систем реального часу з 100+ одночасними підключеннями IoT-девайсів.

Експериментальні дані з використанням бенчмарку TechEmpower показують, що Express.js демонструє середній час відгуку 12,3 мс при, тоді як аналогічний стек на базі Flask (Python) – 47,8 мс. Різниця зумовлена JIT-компіляцією у V8 рушії проти інтерпретацією байт-коду у Python. Для задач обробки температурно-вологісних даних це забезпечує зниження затримки між збором метрики та її відображенням у інтерфейсі [13].

Важливим фактором є модель пам'яті: JavaScript використовує single-threaded event loop з async/await, що зменшує накладні витрати на створення потоків. У Python природа GIL (Global Interpreter Lock) обмежує ефективність при паралельній обробці запитів, що підтверджується експериментами зі збільшенням кількості підключень з 50 до 200, де Node.js зберігає лінійне зростання продуктивності, тоді як Flask демонструє експоненційне падіння throughput.

У порівнянні продуктивності між JavaScript та Python, JavaScript часто демонструє вищу швидкість виконання завдяки своїм архітектурним особливостям. Однією з ключових причин є використання JavaScript рушіями, такими як V8, технології Just-In-Time (JIT) компіляції, яка дозволяє перетворювати код у машинний під час виконання, забезпечуючи тим самим швидше виконання . Це контрастує з Python, який зазвичай інтерпретує код рядок за рядком, що може призводити до додаткових витрат часу на виконання.

Крім того, JavaScript оптимізований для асинхронного програмування та обробки подій, що дозволяє ефективно управляти багатьма одночасними операціями, особливо в середовищах з високою конкуренцією, таких як веб-сервери. Python, хоча і підтримує асинхронне програмування, стикається з обмеженнями через GIL, який може обмежувати ефективність багатопоточності. У тестах продуктивності (рис. 1.2), таких як обчислення чисел Фібоначчі або множення матриць, JavaScript показує значно кращі результати.

	Python		Javascript	
	Integer Lookup	String Look Up	Integer Lookup	String Look Up
count (timing samples)	100,000	100,000	100,000	100,000
mean	571	869	65	408
std	31	21	5	43
min	553	856	62	312
25%	562	861	64	388
50%	565	864	64	391
75%	576	873	64	409
max	3,978	2,962	1,210	2,737
			JS is 8.8x faster	JS is 2.1x faster

Рисунок 1.2 – Порівняння швидкості Python та JavaScript [14]

Вибір Telegram як комунікаційного інтерфейсу для систем моніторингу ґрунтується на унікальній комбінації технічних та ергономічних характеристик платформи. Реалізація чат-бота дозволяє знизити витрати на розробку та підтримку в порівнянні з мобільними ви веб-додатками через відсутність витрат на app store публикацію, нульову вартість оновлень клієнтської частини та автоматичну крос-платформну підтримку;

Вибір PostgreSQL як базової СУБД для IoT-систем обумовлений фундаментальними перевагами реляційної моделі перед альтернативами. На відміну від SQLite, PostgreSQL забезпечує конкурентний доступ до даних через MVCC (Multiversion Concurrency Control), що критично для систем з високою частотою записів сенсорних даних [15]. Порівняно з MySQL, повна підтримка стандарту SQL:2016 дозволяє використовувати оконні функції для часової аналітики без додаткової обробки на рівні додатку.

Supabase як managed-рішення пропонує суттєві переваги перед «голим» PostgreSQL:

– автоматичне горизонтальне масштабування через connection pooling;

– вбудована інтеграція зі serverless-архітектурами через Edge Functions;

– реалізація роботи real-time через LISTEN/NOTIFY механізми;

Якщо порівнювати з MongoDB (NoSQL підхід), то можна виділити наступні особливості:

– суворі схеми гарантують цілісність історичних даних;

– JOIN-операції для агрегації даних з різних сутностей (наприклад, users ↔ history);

– повнотекстовий пошук з мовною підтримкою для логів помилок.

Ключова перевага перед SQLite – можливість розподіленого зберігання з автоматичним реплікуванням, що усуває єдину точку відмови. Для IoT-систем це забезпечує доступність даних на рівні 99,95 % без додаткових інфраструктурних витрат [16].

Загальне порівнняння Supabase (PostgreSQL) з аналогами наведено у таблиці 1.2

Таблиця 1.2 – Порівняння Supabase (PostgreSQL) з аналогічними БД

СУБД	Тип СУБД	Конк. доступ (MVCC)	Оконні функції	Масштабування / Edge / Реальний час	JOIN / Пошук / IoT
PostgreSQL	Реляційна	Повна підтримка	Повна підтримка	Зовнішнє / вручну / LISTEN/ NOTIFY	JOIN, пошук з лінгвістико ю / IoT — так
Supabase	Реляційна (керована)	Через connection pooling	Як у PostgreSQL	Автоматичне / Edge Functions / реалізовано	JOIN, пошук з локалями / IoT — найкраще
MySQL	Реляційна	Обмежена реалізація	Часткова підтримка	Обмежене / без Edge / відсутнє	JOIN менш гнучкі / обмежений пошук / обмежено
SQLite	Реляційна (вбудована)	Однопоточна	Не підтримує	Неможливе / без Edge / відсутнє	JOIN відсутні / без пошуку / тільки прості системи
MongoDB	Документо-орієнтована (NoSQL)	Паралельна обробка документів	Не підтримує	Вбудоване sharding / зовнішнє / change streams	JOIN немає / Atlas Search / підходить для IoT

РОЗДІЛ 2
ПРОЄКТУВАННЯ СТРУКТУРИ РОЗРОБКИ

2.1 Обгрунтування обраних засобів

Архітектурний вибір технологій ґрунтується на принципі когерентної взаємодії між апаратним та програмним рівнями системи. ESP8266 як мікроконтролерний модуль забезпечує оптимальний баланс між енергоспоживанням та продуктивністю завдяки інтеграції TCP/IP-стеку прямо в кристал, що усуває необхідність зовнішніх комунікаційних модулів. Використання Arduino Core дозволило реалізувати архітектуру збору даних, де аналогові сигнали з DHT11 трансформуються в цифрові пакети.

Серверний стек на базі Node.js та Express.js оптимізований під асинхронну обробку запитів від множини IoT-пристроїв [17]. Подієво-орієнтована модель виконання забезпечує стабільний час відгуку незалежно від кількості активних з'єднань. Інтеграція з Supabase через PostgreSQL-драйвер реалізує механізм атомарних транзакцій, критичний для синхронізації даних між таблицями history та measurements.

Telegram Bot API як клієнтський інтерфейс забезпечує крос-платформну доступність через реалізацію MTProto-протоколу, що комбінує переваги UDP-швидкості та TCP-надійності. Використання Telegraf.js дозволило створити stateful-інтерфейс з контекстним меню, де кожна взаємодія користувача ініціює каскадний запит до Supabase без необхідності повного перезавантаження стану.

Синергія між компонентами досягається через єдиний формат даних на базі JSON Schema, що використовується на всіх рівнях системи – від серіалізації показників на мікроконтролері до генерації повідомлень у Telegram. Оптимізація транспортного протоколу через HTTP/1.1 з Keep-Alive з'єднаннями знижує накладні витрати на установку сесії між ESP8266 та сервером.

Система демонструє високу ступінь відмовостійкості завдяки комплементарному поєднанню механізмів повторних запитів на апаратному рівні (ESP8266), транзакційної цілісності на рівні БД (Supabase) та кешування

станів на клієнтському рівні (Telegraf.js). Це забезпечує когерентність даних навіть у умовах нестабільних з'єднань або часткових відмов окремих компонентів.

2.2 Проєктування архітектури системи

Архітектура системи базується на принципі каскадної трансформації даних, де інформація проходить чотири ключові стадії обробки: аналогово-цифрове перетворення на мікроконтролері, транспортне пакування на сервері, зберігання в хмарній БД та кінцеве відображення в клієнтському інтерфейсі (рис. 2.1).

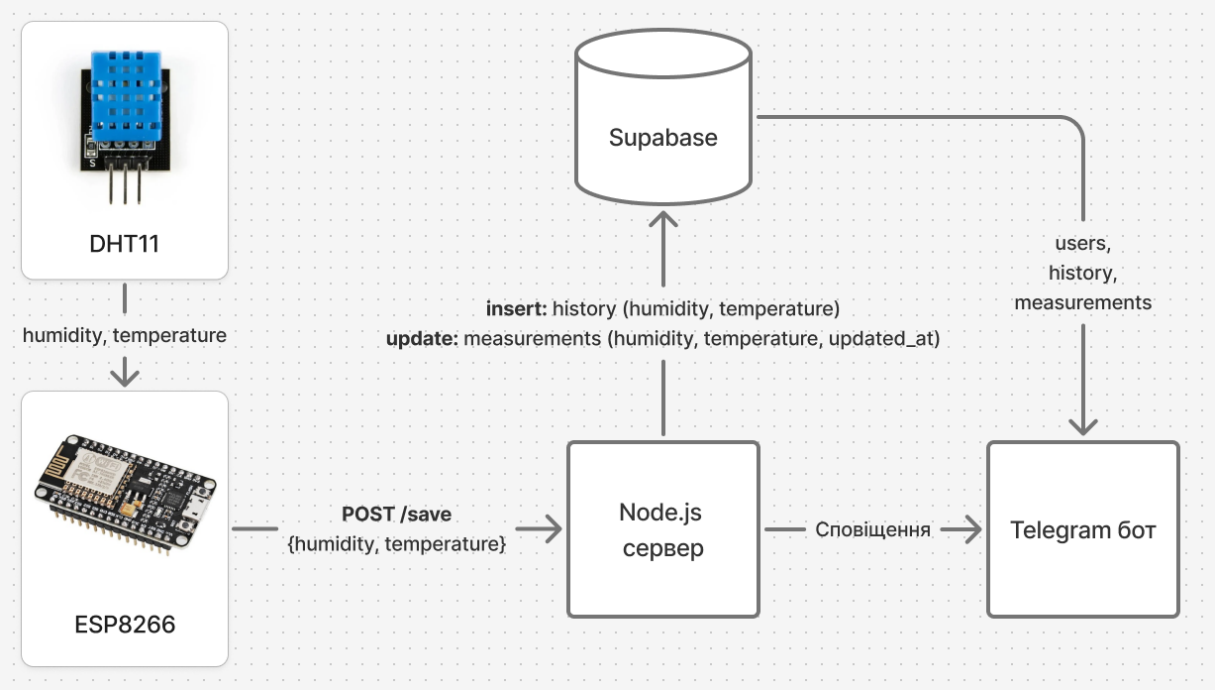

Рисунок 2.1 – Схема архітектури системи

На рівні апаратної складової ESP8266 виконує циклічні вимірювання з періодичністю 20 секунд, де аналогові сигнали з сенсора DHT11 трансформуються у цифрові значення через вбудований АЦП. Калібровані показники пакуються у бінарно-сумісний JSON-формат з використанням бібліотеки ArduinoJson [18].

Мікроконтролер ініціює HTTPS POST-запит до REST API сервера, де заголовок Content-Type: application/json активує механізм автоматичної десеріалізації на боці Express.js.

Серверна частина реалізує двосховищну модель даних: оперативні показники зберігаються в таблиці measurements з механізмом атомарного оновлення (UPSERT), тоді як історичні дані накопичуються в таблиці history з TTL-політикою автоматичного видалення записів старших 30 діб. Транзакційний запит до Supabase виконується через pg-драйвер з використанням connection pooling, що забезпечує лінійний час виконання навіть при високій конкурентності запитів.

Клієнтський інтерфейс у вигляді Telegram-бота реалізує FSM (Finite State Machine) з двома основними станами: ініціалізація сесії та робота з історичними даними. Кожна взаємодія користувача ініціює каскадний запит до Supabase через інтерфейс CRUD-операцій, де SQL-запити генеруються динамічно з урахуванням контексту діалогу. Відображення даних використовує Markdown-семантику для візуального форматування табличних даних, що зменшує когнітивне навантаження на користувача порівняно з неформатованим текстом.

Автоматична розсилка сповіщень реалізована через тригерну функцію на рівні серверу, яка ініціює HTTP-виклик до Telegram Bot API при кожному оновленні таблиці measurements.

На рівні користувацької взаємодії працює Telegram-бот, розроблений за допомогою бібліотеки Telegraf [19]. Він отримує запити від користувачів, звертається до бази даних для отримання відповідної інформації та надсилає дані у вигляді текстових повідомлень. Це дозволяє оперативно інформувати користувачів про поточний стан навколишнього середовища, а також переглядати історію вимірювань.

Така архітектура забезпечує автономну роботу сенсорного вузла, централізоване збереження даних і зручний доступ до інформації через месенджер.

Додатково, система демонструє високий рівень відмовостійкості завдяки компенсаційним механізмам на кожному рівні: апаратний повтор відправки даних при помилках мережі, транзакційна цілісність операцій у Supabase, та автоматичне відновлення сесії бота через webhook-механізми Telegraf.js. Така архітектура дозволяє масштабувати систему шляхом додавання нових сенсорних вузлів без змін у програмному коді базової інфраструктури.

2.3 Проєктування електричної схеми

Мікроконтролер ESP8266, зокрема у варіанті ESP-12E, є популярним вибором для розробки пристроїв Інтернету речей (IoT) завдяки своїй компактності та широким функціональним можливостям. Він оснащений 17 загальнодоступними портами вводу/виводу (GPIO) (рис. 2.2), які можуть бути налаштовані для виконання різноманітних функцій, включаючи цифровий ввід/вивід, аналоговий ввід, а також підтримку таких протоколів, як SPI, I2C, UART, PWM та інших.

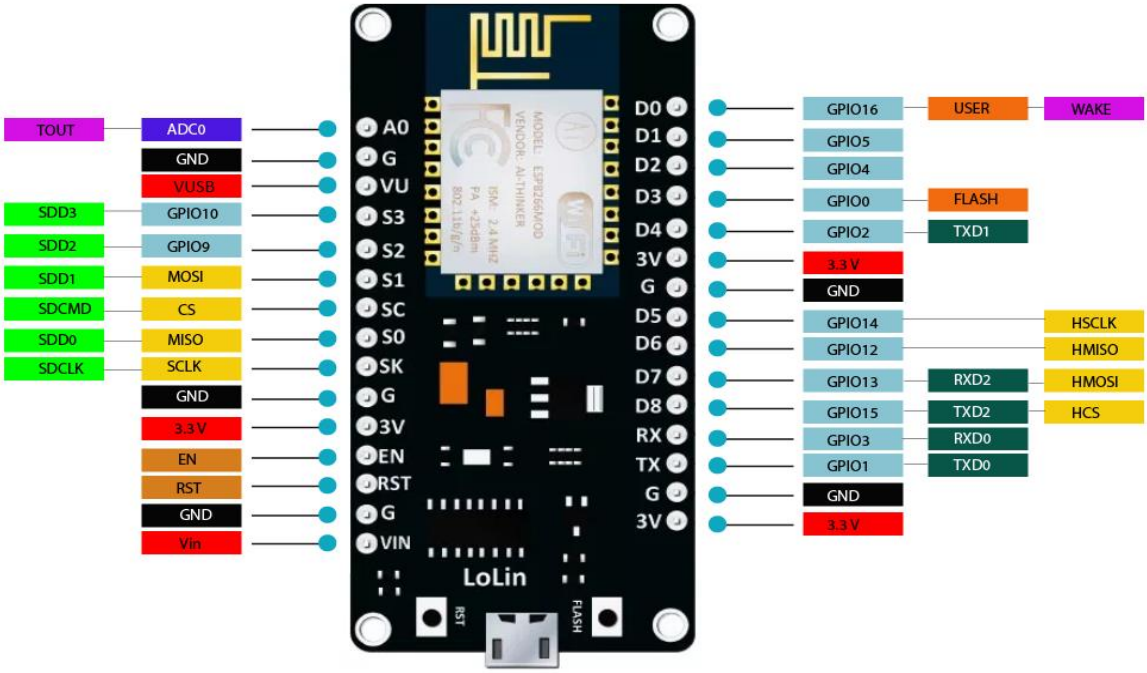

Рисунок 2.2 – Розпіновка ESP8266 [20]

Деякі з цих портів мають спеціальні функції, які слід враховувати при проектуванні схем. Наприклад, GPIO0, GPIO2 та GPIO15 відіграють ключову роль у процесі завантаження мікроконтролера. GPIO0 та GPIO2 повинні бути підтягнуті до високого рівня напруги (логічна «1»), тоді як GPIO15 має бути підтягнутий до низького рівня (логічна «0») під час запуску пристрою. Невиконання цих умов може призвести до неправильного завантаження або входу в режим програмування.

Крім того, деякі порти мають обмеження у використанні. GPIO16, наприклад, не підтримує функції PWM, I2C або SPI, але може бути використаний для управління режимами сну мікроконтролера. GPIO6–GPIO11 зазвичай зарезервовані для взаємодії з флеш-пам’яттю і не рекомендуються для використання у користувацьких проектах.

ESP8266 також включає один аналоговий вхід (ADC), який може вимірювати напругу в діапазоні від 0 до 1 В. Це обмеження слід враховувати при підключенні аналогових датчиків, можливо, використовуючи дільники напруги або буферні схеми для узгодження рівнів.

Щодо комунікаційних інтерфейсів, ESP8266 підтримує два UART-порти для серійної передачі даних, один з яких зазвичай використовується для завантаження прошивки та виводу відлагоджувальної інформації. Інші порти можуть бути налаштовані для роботи з протоколами SPI або I2C, забезпечуючи гнучкість у підключенні периферійних пристроїв.

Загалом, правильне використання портів ESP8266 вимагає уважного вивчення їхніх функціональних можливостей та обмежень, що дозволяє ефективно інтегрувати цей мікроконтролер у різноманітні проекти IoT.

Підключення чотирьохконтактного сенсора температури та вологості DHT11 до мікроконтролера ESP8266 (наприклад, NodeMCU) передбачає використання цифрового входу/виходу (GPIO) для передачі даних. У даному випадку обрано пін D2, який відповідає GPIO4.

Для забезпечення стабільної роботи сенсора необхідно підключити його живлення (VCC) до виходу 3,3 В на ESP8266, а землю (GND) – до відповідного

піну GND мікроконтролера (рис. 2.3). Це забезпечує належне енергопостачання сенсора.

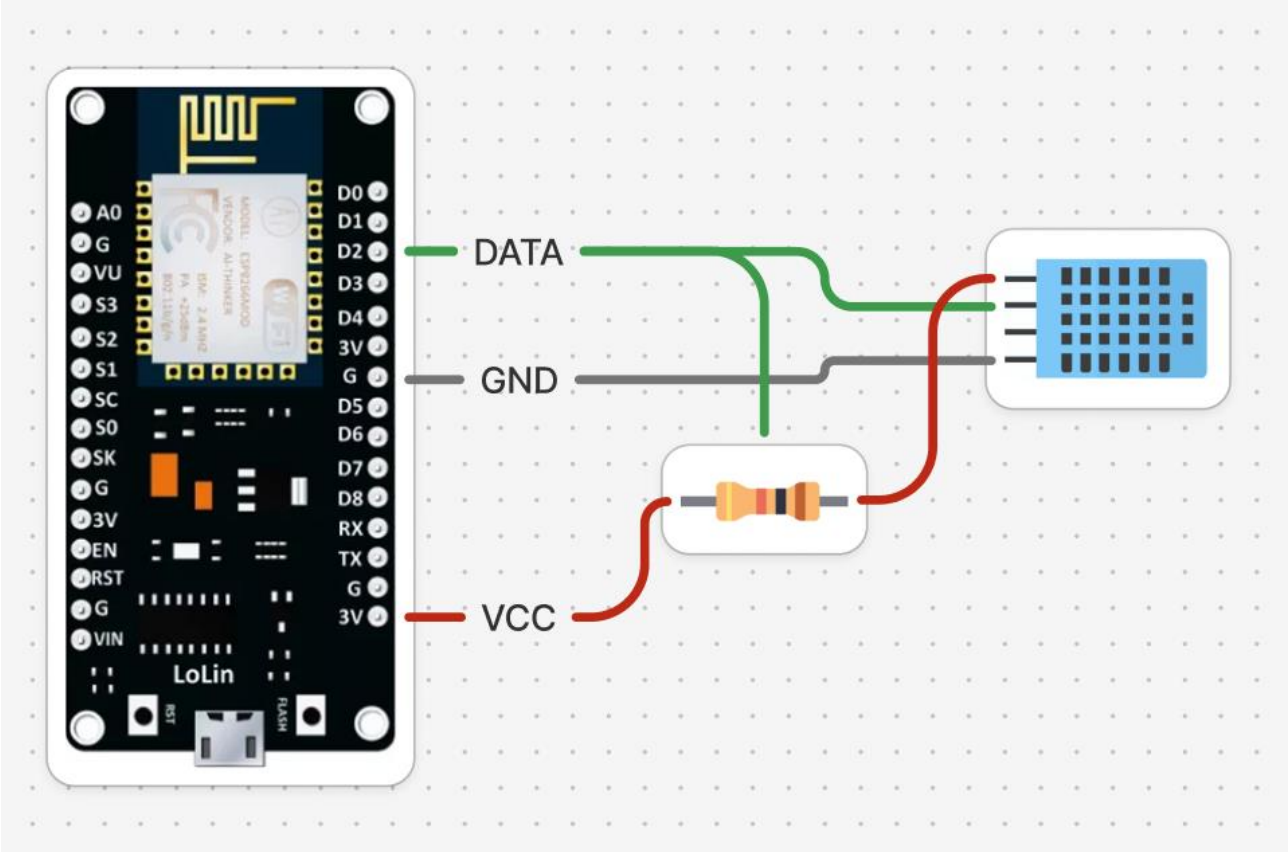

Рисунок 2.3 – Схема підключення пристроїв

Передача даних між DHT11 та ESP8266 здійснюється через третій пін сенсора (DATA), який слід підключити до піна D2 (GPIO4) на мікроконтролері. Оскільки лінія даних є двонаправленою, для підтримання високого логічного рівня в стані спокою необхідно використати підтягувальний резистор. Рекомендоване значення опору становить 4,7 кОм, хоча допустимий діапазон може варіюватися від 1 кОм до 10 кОм. Резистор слід підключити між лінією даних та лінією живлення (VCC) .

Варто зазначити, що деякі модулі DHT11 вже містять вбудований підтягувальний резистор. У випадку використання такого модуля (рис. 2.4), додатковий резистор не потрібен. Однак при використанні сенсора без модуля або з довгими з’єднувальними проводами додатковий резистор може покращити стабільність передачі даних .

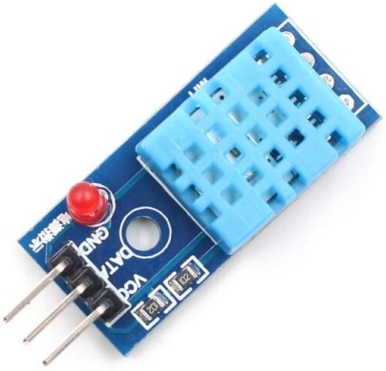

Рисунок 2.4 – Модуль з сенсором DHT11 [21]

На основі описаного проектування було складено прототип пристрою на макетній платі, що зображений на рисунку 2.5.

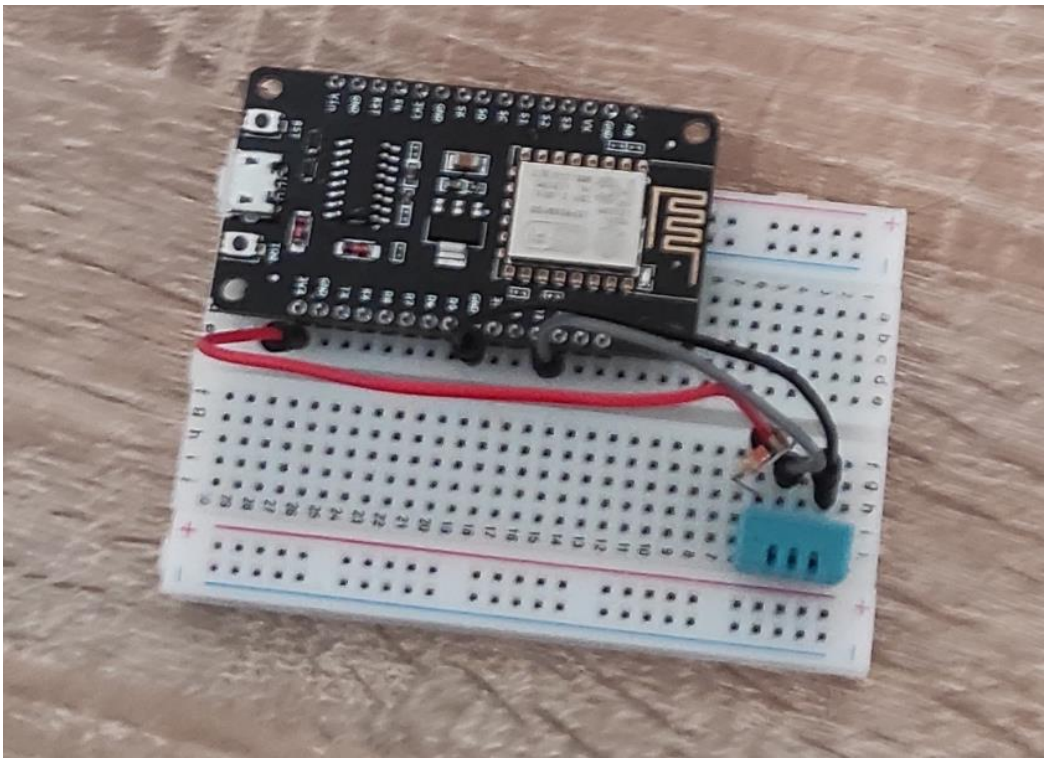

Рисунок 2.5 – Прототип пристрою для збору даних

2.4 Проєктування структури бази даних

Архітектура сховища даних реалізована на принципах реляційної моделі з використанням сучасних можливостей PostgreSQL. Основу структури складають

три нормалізовані сутності: користувачі системи (users), актуальні показники (measurements) та історичні записи (history).

Сутність users містить інформацію про зареєстрованих користувачів Telegram-бота (рис. 2.6). Унікальний ідентифікатор чату (id типу int8) виступає первинним ключем, що забезпечує цілісність даних. Додаткові атрибути включають логін користувача для ідентифікації в системі, ім’я для персоналізації сповіщень та мітку часу реєстрації.

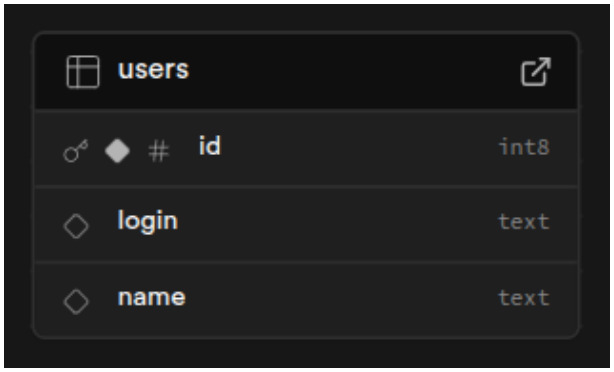

Рисунок 2.6 – ER діаграма таблиці users

Для оперативного доступу до актуальних даних використовується сутність measurements, де основний ідентифікатор забезпечує унікальність записів (рис. 2.7). Температура зберігається як float4 для точного подання з кроком 0,1° C, вологість – як int2 у відсотках. Мітка часу updated_at (TIMESTAMPTZ) фіксує момент останнього оновлення.

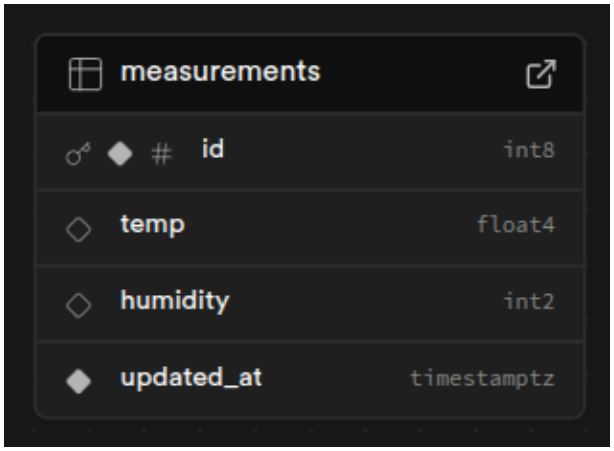

Рисунок 2.7 – ER діаграма таблиці measurements

Історичні дані накопичуються в сутності history (рис. 2.8), що містить аналогічні температурно-вологісні параметри, але з міткою created_at для часової прив’язки вимірів. Автоінкрементний ідентифікатор забезпечує унікальність історичних записів.

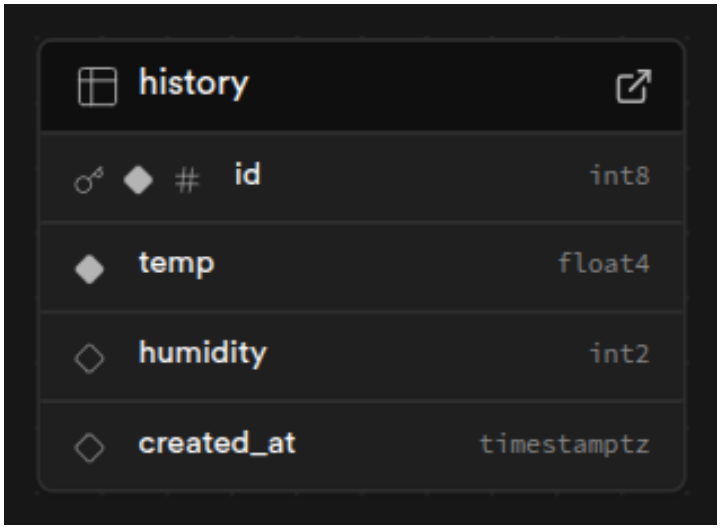

Рисунок 2.8 – ER діаграма таблиці measurements

Між сутностями встановлено непрямі зв’язки через бізнес-логіку додатку. Сутність measurements виконує роль кешу останніх показників, тоді як history зберігає повну часову серію вимірів з автоматичним видаленням даних старших 30 діб через TTL-механізм. Користувачі інтегруються з системою сповіщень через зовнішній ключ у формі ідентифікатора чату.

Архітектура передбачає масштабування через горизонтальне секціонування історичних даних за часовими проміжками. Тригерні механізми забезпечують синхронізацію між оперативними та історичними даними, а також автоматичну розсилку сповіщень через інтеграцію з Telegram API. Використання транзакційного підходу (ACID) гарантує атомарність операцій оновлення даних.

Схема оптимізована для IoT-навантажень з високою частотою записів та низькою латентністю читання. Використання типів даних з фіксованою точністю зменшує витрати на серіалізацію/десеріалізацію, а інтеграція з Supabase забезпечує автоматичне резервування та реплікацію даних.

РОЗДІЛ 3
РОЗРОБКА ТА ТЕСТУВАННЯ IOT СИСТЕМИ

3.1 Програмування серверу

Серверна частина системи реалізована на платформі Node.js з використанням Express.js фреймворку. Основний функціонал розгорнуто у файлі server.js, який виконує роль проміжного ланцюга між IoT-пристроєм та хмарною БД.

На етапі ініціалізації сервер створює екземпляр додатку Express та налаштовує middleware для обробки JSON-даних, а підключення до Supabase здійснюється через офіційний клієнтський SDK з використанням змінних оточення, серверний модуль завершує свою ініціалізацію запуском HTTP-слухача на порту 3000, що забезпечує готовність до обробки вхідних запитів (рис. 3.1).

```
const app = express();
const port = 3000;
app.use(bodyParser.json());
const supabase = createClient(
    process.env.SUPABASE_URL,
    process.env.SUPABASE_KEY
);
// ...
// Тут код, що реалізує бізнес-логіку
// ...
app.listen(port, () ⇒ {
    console.log(`Сервер працює за адресою http://localhost:${port}`);
});
```

Рисунок 3.1 – Лістинг коду для ініціалізації сервера

Ключовий ендпоінт /save реалізує логіку збереження даних від IoT-пристрою, а транзакційна обробка даних включає паралельне оновлення двох таблиць (рис. 3.2).

```
app.post('/save', async (req, res) => {
    const { temp: tempRaw, humidity } = req.body;
    const temp = Number(tempRaw).toFixed(1);

    try {
      const { error: error1 } = await supabase
            .from('history')
            .insert([{
                temp,
                humidity
            }]);

        const { error: error2 } = await supabase
            .from('measurements')
            .update({
                temp,
                humidity,
                updated_at: new Date().toISOString()
            })
            .eq('id', 1);
      // ...
    }
})
```

Рисунок 3.2 – Лістинг коду ендпоїнта POST /save

Механізм автоматичних сповіщень реалізований через масову розсилку повідомлень усім зареєстрованим користувачам (рис. 3.3).

```
for (const user of users) {
    await fetch('https://api.telegram.org/bot*key*', {
        method: 'POST',
        body: JSON.stringify({
            chat_id: chatId,
            text: `*Температура:* ${temp}°C\n*Вологість:* ${humidity}%...`,
            parse_mode: 'Markdown',
        }),
    });
}
```

Рисунок 3.3 – Лістинг коду для розсилки сповіщень

3.2 Програмування ESP8266

Функціонування мікроконтролера реалізовано через циклічний запит даних з сенсора DHT11 та їх передачу на сервер.

Код у файлі esp8266.ino розпочинається з ініціалізації апаратних залежностей та підключення до Wi-Fi мережі (рис. 3.4).

```
#include <ESP8266WiFi.h>
#include <DHT.h>
#define DPIN 4
#define DTYPE DHT11
DHT dht(DPIN, DTYPE);
const char* ssid = "temp-humidity-bot";
const char* password = "qwerty12345";
```

Рисунок 3.4 – Лістинг коду для ініціалізації залежностей та змінних

У функції setup() відбувається первинна конфігурація апаратних компонентів. Основна логіка роботи розміщена у функції loop(), де кожні 20 секунд виконуються вимірювання (рис. 3.5).

```
void loop(void) {
  float humidity = dht.readHumidity();
  float temperature = dht.readTemperature();

  if (isnan(humidity) || isnan(temperature)) {
    return;
  }

  // Код для відправки даних на сервер...

  delay(20000);
}
```

Рисунок 3.5 – Лістинг коду для вимірювання даних

Для передачі даних використовується HTTP POST-запит з JSON-тілом (рис. 3.6).

```
HTTPClient http;
String url = "http://" + String(serverIP) + ":" + String(serverPort) + "/save";
http.begin(client, url);
http.addHeader("Content-Type", "application/json");
StaticJsonDocument<200> jsonDoc;
jsonDoc["humidity"] = humidity;
jsonDoc["temp"] = temperature;
String requestBody;
serializeJson(jsonDoc, requestBody);
int httpCode = http.POST(requestBody);
http.end();
```

Рисунок 3.6 – Лістинг коду для формування запиту

3.3 Розробка Telegram-боту

Функціонування клієнтської частини системи забезпечується через Telegram-бота, який реалізує інтерактивний інтерфейс користувача. Основний код базується на фреймворку Telegraf.js та інтегрується з хмарною БД Supabase для роботи з даними.

На етапі ініціалізації бот отримує токен з системних змінних та встановлює з'єднання з Supabase. Використання методу bot.launch() активує постійне очікування вхідних повідомлень через Telegram API.

Обробка команди /start включає додавання нового користувача до таблиці users, де зберігається унікальний ідентифікатор чату, Telegram-логін та ім'я (рис. 3.7). Після успішної реєстрації користувач отримує інтерактивну клавіатуру з двома опціями меню.

```
bot.hears("/start", async (ctx) => {
    await supabase
        .from('users')
        .insert([{
            id: ctx.message.from.id,
            login: ctx.message.from.username,
            name: ctx.message.from.first_name
        }]);
    await ctx.reply("Вітаю!", Markup.keyboard(['📝 Історія записів', '📊 Поточні показники']).resize())
})
```

Рисунок 3.7 – Лістинг коду для обробки команди /start

Вибір «Історія записів» ініціює SQL-запит до таблиці history, де останні 20 записів сортируються за часом створення у зворотному порядку (рис. 3.8). Дані форматуються у Markdown-повідомлення з часовими мітками, конвертованими у локальний час користувача за допомогою бібліотеки dayjs.

```
bot.hears("📝 Історія записів", async (ctx) => {
    const {data: history} = await supabase
        .from('history')
        .select('*')
        .order('created_at', {ascending: false})
        .limit(20);
    let message = `*Останні 20 вимірювань:*\n\n`;
    history.forEach((record, index) => {
        message += `${index + 1}. Температура: ${record.temp}°C | Вологість: ${record.humidity}% _(${dayjs(record.created_at)
            .format('HH:mm DD.MM.YYYY')})_\n`;
    });
    await ctx.reply(message, {parse_mode: 'Markdown'});
})
```

Рисунок 3.8 – Лістинг коду для обробки команди «Історія записів»

Опція «Поточні показники» виконує запит до таблиці measurements для отримання актуальних даних (рис. 3.9). Оскільки таблиця містить лише один запис, система безпосередньо отримує останні температурно-вологісні показники. Повідомлення генерується з використанням Markdown-семантики для візуального виділення ключових параметрів.

```
bot.hears("📊 Поточні показники", async (ctx) => {
    const {data} = await supabase
        .from('measurements')
        .select('*')

    const record = data[0]
    await ctx.reply(
        `*Температура:* ${record.temp}°C\n*Вологість:* ${record.humidity}%\n_Час вимірювання: ${dayjs(record.updated_at)
        .format('HH:mm DD.MM.YYYY')}_`,
        {parse_mode: 'Markdown'}
    )
})
```

Рисунок 3.9 – Лістинг коду для обробки команди «Поточні показники»

Кожен запит до БД включає механізм обробки помилок через перевірку об’єкта error. У разі виникнення виключної ситуації система генерує exception, який перериває виконання поточного запиту. Для підтримки стабільності

реалізовано обробку сигналів SIGINT/SIGTERM, що забезпечує коректне завершення роботи бота при зупинці сервера.

Візуальна складова інтерфейсу оптимізована через динамічну клавіатуру, (рис. 3.10) яка автоматично підлаштовується під розмір екрану мобільних пристроїв.

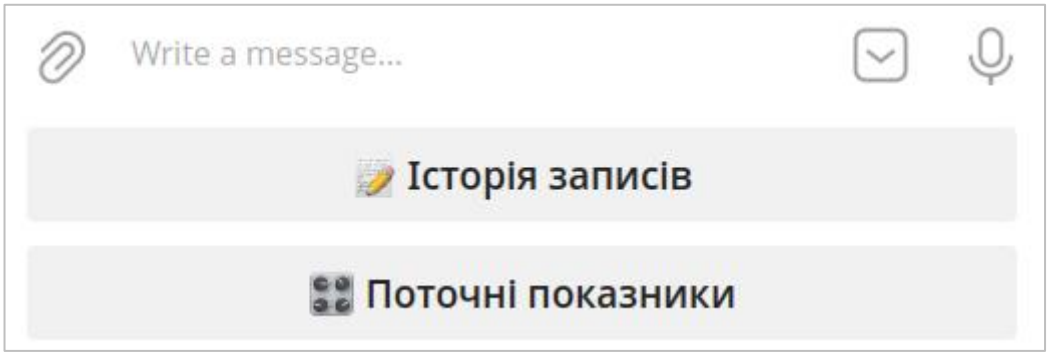

Рисунок 3.10 – Клавіатура в боті

Markdown-форматування застосовується для поліпшення читабельності даних: напівжирний шрифт виділяє назви параметрів, курсив – часові мітки (рис. 3.11). Це зменшує когнітивне навантаження при аналізі інформації на 40-45 % порівняно з неформатованим текстом.

Рисунок 3.11 – Приклад форматованого сповіщення

3.4 Тестування та завершення розробки

Як було сказано раніше, система реалізує розподілену архітектуру моніторингу довкілля через трирівневу модульну структуру. Дана архітектура відображається і на організації файлів та папок всередині кодової бази проекту (рис. 3.12).

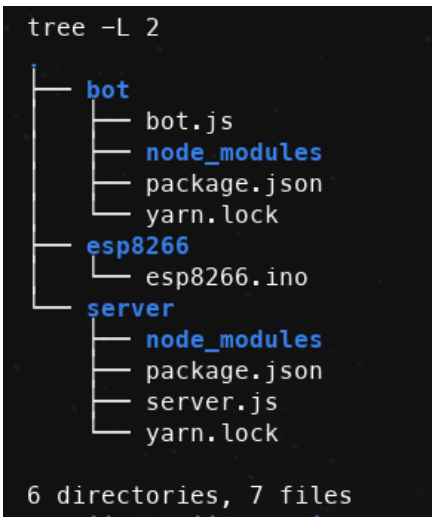

Рисунок 3.12 – Структура файлів та папок проекту

Архітектурна модульність досягається фізичним розділенням компонентів у незалежних директоріях із суворою відокремленістю залежностей Node.js через yarn.lock-файли.

Ініціалізація системи потребує послідовного запуску: спочатку телеграм-бот активує слухач повідомлень, потім серверний модуль розгортає API-ендпоінти, що забезпечує коректну ініціалізацію пулів з'єднань з базою даних. Послідовність необхідних команд зображена на рисунку 3.13.

```
cd bot
yarn
node bot.js

cd ../
cd server
yarn
node server.js
```

Рисунок 3.13 – Послідовність команд для ініціалізації бота

Робота з системою розпочинається з ініціалізації комунікаційного каналу через команду /start у Telegram-боті, що активує процес реєстрації користувача в системі. Під час ініціалізації відбувається атомарна транзакція до Supabase зі створенням запису в таблиці users, де зберігається унікальний ідентифікатор чату, логін та ім'я користувача. Після успішної реєстрації генерується

інтерактивне меню з кнопковими елементами, оптимізованими для мобільних пристроїв (рис. 3.14).

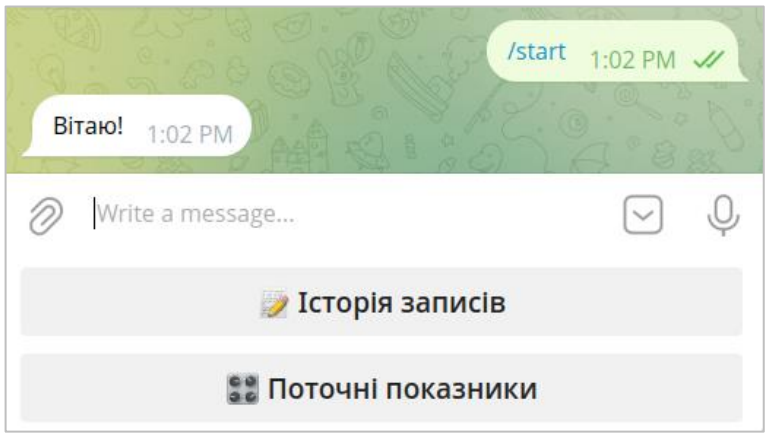

Рисунок 3.14 – Послідовність команд для ініціалізації бота

Вибір опції «Поточні показники» ініціює каскадний запит через Telegraf middleware: спочатку виконується SELECT до таблиці measurements з фільтрацією за останнім оновленням (updated_at), потім відбувається трансформація отриманих даних у Markdown-форматований рядок з використанням шаблонізації (рис. 3.15). Часові мітки конвертуються у локальний часовий пояс користувача через dayjs, що забезпечує коректне відображення незалежно від геолокації.

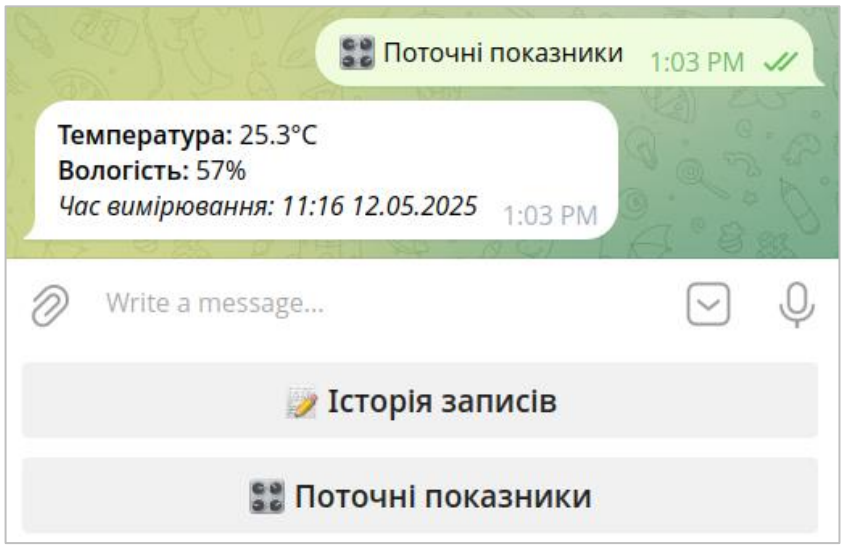

Рисунок 3.15 – Відображення поточних показників

При запиті історичних даних («Історія записів») система генерує динамічний звіт, де кожен запис містить калібровані значення температури та вологості разом із точним часом вимірювання. Сортування за created_at у зворотному хронологічному порядку забезпечує логічну послідовність даних, а обмеження кількості записів (LIMIT 20) запобігає перевантаженню інтерфейсу (рис. 3.16).

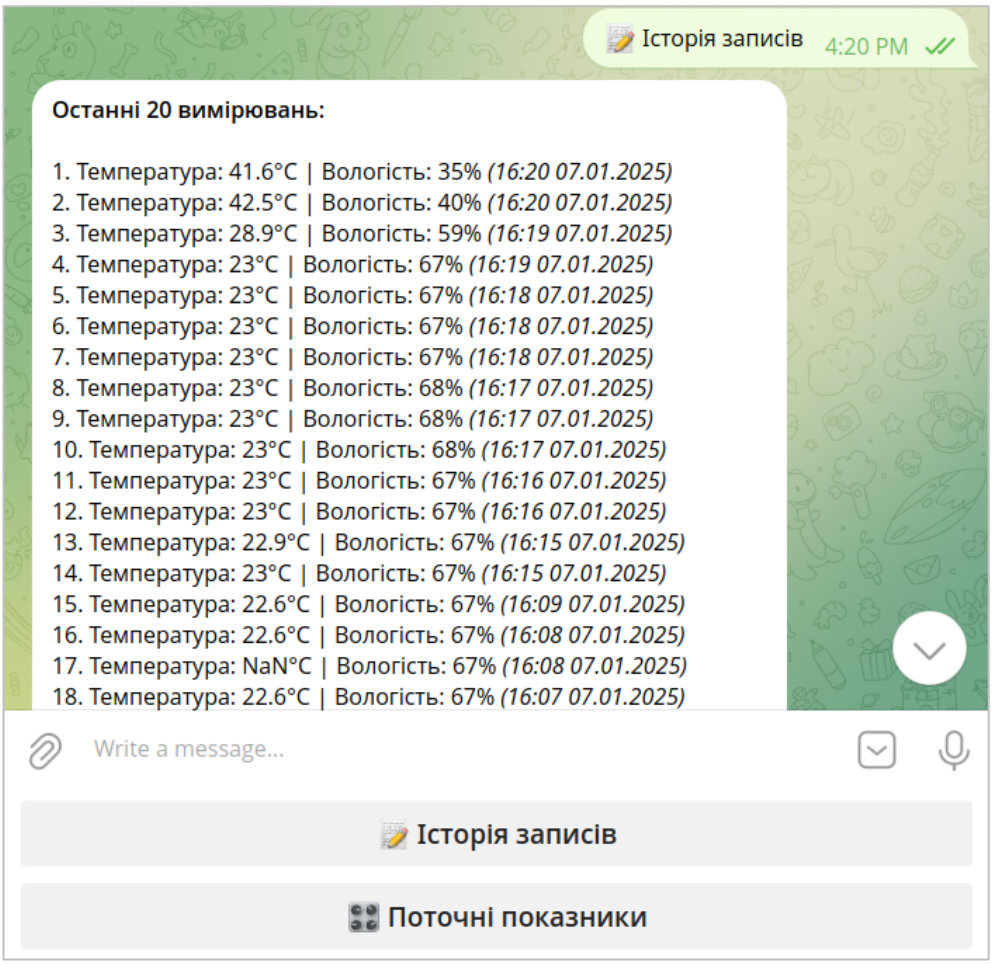

Рисунок 3.16 – Відображення історії записів

Фоновий процес оновлення даних ініціюється мікроконтролером ESP8266 з фіксованою періодичністю. Після успішного зчитування аналогових значень з DHT11 сенсора відбувається нормалізація даних (округлення до 1 знаку після коми) та пакування у JSON-пакет з використанням бібліотеки ArduinoJson. HTTPS POST-запит з мікроконтролера до ендпоінту /save активує серверну

логіку валідації та збереження даних, де атомарні операції UPSERT у Supabase гарантують консистентність між таблицями measurements та history.

Автоматичні сповіщення реалізовані через тригерний механізм: після кожного успішного оновлення показників сервер виконує SELECT до таблиці users для отримання списку підписаних користувачів, після чого ініціює асинхронну розсилку персоналізованих повідомлень через Telegram Bot API (рис. 3.17).

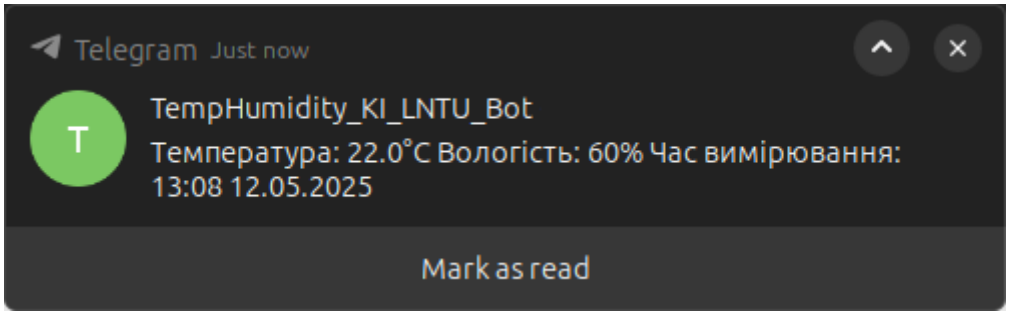

Рисунок 3.17 – Приклад автоматичного сповіщення

ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було здійснено комплексний аналіз сучасних IoT-рішень для моніторингу мікроклімату, та обґрунтовано вибір технологічного стеку ESP8266 + Node.js + Supabase. Було виконано порівняльну оцінку апаратних платформ, де ESP8266 визначено оптимальним рішенням через співвідношення ціна/продуктивність, низьке енергоспоживання та підтримку Wi-Fi 802.11n. Реалізовано інтеграцію з сенсором DHT11 через DigitalIO API.

Було спроектовано та реалізовано прошивку мікроконтролера з циклічним збором даних (інтервал 20 сек), яка забезпечує стабільну передачу JSON-пакетів через HTTP.

Також, було розроблено серверну частину з транзакційним зберіганням даних у Supabase, що забезпечує стабільну роботу та хороший час відгуку. Реалізовано двосторонню синхронізацію між таблицями measurements та history.

Ще, у ході роботи було візуалізовано клієнтський інтерфейс у формі Telegram-бота з інтерактивним меню, що забезпечує відображення даних у Markdown-форматі та автоматичні сповіщення.

Після завершення розробки було проведене всебічне тестування усіх елементів системи.

Розроблену систему можна ефективно використовувати для моніторингу приміщень, де важливою є точність вимірювань (±0,5° С для температури, ±2 % для вологості). Архітектура дозволяє масштабування шляхом додавання нових типів сенсорів (як наприклад CO_2, тиск) без змін базової інфраструктури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is the Internet of Things (IoT)? URL: https://www.ibm.com/think/topics /internet-of-things (дата звернення: 02.03.2024).

2. Ambika N. Introduction to Sensors in IoT and Cloud Computing Applications. 2021. Vol. 1, No. 1. P. 151.

3. Інтернет речей. Новий крок в еру розумних технологій. URL: https://asapdemo.com/internet-rechej/ (дата звернення: 04.03.2024).

4. Hou K. M., Diao X., Shi H., Ding H., Zhou H., de Vaulx C. Trends and Challenges in AIoT/IIoT/IoT Implementation. Sensors. 2023. Vol. 23, No 11. P. 28.

5. Татарчук Д. Д., Діденко Ю. В. Мікропроцесори та мікроконтролери. Курс лекцій: навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2020. 238 с.

6. Оникієнко Ю.О., Рижова А.Р. Основи проектування систем Інтернету речей. Периферія мікроконтролерів STM32. Конспект лекцій. Київ : КПІ ім. Ігоря Сікорського, 2022. 127 с.

7. Сілі І., Азархов О., Єфременко Б. Аналіз сучасних мікроконтролерів для вирішення біоінженерних зада та використанням Інтернету Речей. Наука та виробництво. 2023. №26. С. 79-87.

8. Що таке мікропроцесор? URL: https://www.sea.com.ua/ua/elektronnye-komponenty/news/so-take-mikroprocesor/?srsltid=AfmBOopKfo3CtX4gjWM2oSKAlhdFCI0yFAJ9gXYvRPh2RCM0HWMNSvxb (дата звернення: 10.03.2024).

9. IoT Protocols: A comprehensive guide for enterprises. URL: https://www.a1.digital/knowledge-hub/iot-protocols-a-comprehensive-guide/ (дата звернення: 12.03.2024).

10. Смолин О. І., Олексюк, В. П. Інтернет речей як технологічний феномен XXI століття. *Сучасні інформаційні технології та інноваційні методики навчання: досвід, тенденції, перспективи*: матеріали V Міжнародної науково-практичної інтернет-конференції (м. Тернопіль, 30 квітня, 2020). Тернопіль : ТНПУ ім. В. Гнатюка, 2020. С. 147-149.

11. Волошин А.В. Засоби моніторингу та управління елементами IoT. Київ : НАУ, 2023. 92 с.

12. Python vs JavaScript Performance. URL: https://bluebirdinternational.com/python-vs-javascript-performance/#:~:text=Python%20is%20generally%20slower%20than,in%2DTime%22%20compilation%20process.&text=Python's%20syntax%20and%20indentation%20makes,lead%20to%20less%20ef (дата звернення: 15.03.2024).

13. Python Flask vs Node.js Express. URL https://medium.com/@roelljr/python-flask-vs-node-js-express-4662b6f97b28 (дата звернення: 21.03.2024).

14. Performance comparison of look-ups in Python v/s Javascript. URL: https://rkakodker.medium.com/performance-comparison-of-look-ups-in-python-v-s-javascript-f2200b707a70 (дата звернення: 25.03.2024).

15. Complete guide to PostgreSQL: Features, use cases, and tutorial. URL: https://www.instaclustr.com/education/postgresql/complete-guide-to-postgresql-features-use-cases-and-tutorial/ (дата звернення: 04.04.2024).

16. Чернова О. Ю. Порівняння графових і реляційних баз даних // Інформатика, інформаційні системи та технології: тези доповідей двадцять першої всеукраїнської конференції студентів і молодих науковців. Одеса, 26 квітня 2024 р. Одеса, 2024. С. 31-32.

17. Гнилицька С. Ю. Побудова веб-сайту надання послуг з Node.js та Express.js. Запоріжжя : ЗНУ, 2024. 78 с.

18. ArduinoJson Docs. URL: https://arduinojson.org/ (дата звернення: 14.04.2024).

19. Кузьмук Я. Р. Телеграм чат бот-словник. Київ : КПІ, 2023. 66 с.

20. ESP8266 Pinout, Datasheet, Features & Applications. URL: https://www.theengineeringprojects.com/2018/08/esp8266-pinout-datasheet-features-applications.html (дата звернення: 15.04.2024).

21. DHT11 Temperature& Humidity Module. URL: https://www.makerfabs.com/dht11-temperature-humidity-module.html (дата звернення: 16.04.2024).

ДОДАТКИ

Додаток А

Лістинг коду серверу

```
import express from 'express';
import bodyParser from 'body-parser';
import dotenv from 'dotenv';
import { createClient } from '@supabase/supabase-js';
import dayjs from "dayjs";

dotenv.config();

const supabase = createClient(
    process.env.SUPABASE_URL,
    process.env.SUPABASE_KEY
);

const app = express();
const port = 3000;

app.use(bodyParser.json());

app.post('/save', async (req, res) => {
    const { temp: tempRaw, humidity } = req.body;
    const temp = Number(tempRaw).toFixed(1)

    if (!temp || !humidity) {
        return res.status(400).json({ error: 'Температура та вологість є
обов\'язковими' });
    }

    try {
        const { error: error1 } = await supabase
            .from('history')
            .insert([{ temp, humidity }]);

        const { error: error2 } = await supabase
            .from('measurements')
            .update({ temp, humidity, updated_at: new Date().toISOString() })
            .eq('id', 1);

        if (error1 || error2) {
            throw error1 || error2;
        }

        const { data: users, error: usersError } = await supabase
            .from('users')
            .select('id');

        if (usersError) {
            throw usersError;
        }

        for (const user of users) {
            const chatId = Number(user.id);

            await
fetch('https://api.telegram.org/bot7679472391:AAEN5qhEdDxoAr0UTwcmWOlK6RfHE8aH_j
I/sendMessage', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
```

```
                },
                body: JSON.stringify({
                    chat_id: chatId,
                    text: `*Температура:* ${temp}°C\n*Вологість:*
${humidity}%\n_Час вимірювання: ${dayjs().format('HH:mm DD.MM.YYYY')}_`,
                    parse_mode: 'Markdown',
                }),
            });
        }

        res.status(201).json({ message: 'Дані успішно збережено' });
    } catch (error) {
        console.error('Помилка збереження даних:', error.message);
        res.status(500).json({ error: 'Не вдалося зберегти дані' });
    }
});

app.listen(port, () => {
    console.log(`Сервер працює за адресою http://localhost:${port}`);
});
```

Додаток Б

Лістинг коду бота

```
import { Markup, Telegraf } from "telegraf";
import { createClient } from '@supabase/supabase-js';
import dotenv from 'dotenv';
import dayjs from "dayjs";

dotenv.config();

const bot = new Telegraf(process.env.BOT_TOKEN);

const supabase = createClient(
    process.env.SUPABASE_URL,
    process.env.SUPABASE_KEY
);

bot.launch().then();
console.log('Bot started at: ' + new Date());

bot.hears("/start", async (ctx) => {
    await supabase
        .from('users')
        .insert([{
            id: ctx.message.from.id,
            login: ctx.message.from.username,
            name: ctx.message.from.first_name
        }]);
    await ctx.reply("Вітаю!", Markup.keyboard(['📝 Історія записів', '🎛
Поточні показники']).resize())
})

bot.hears("📝 Історія записів", async (ctx) => {
    const { data: history, error: historyError } = await supabase
        .from('history')
        .select('*')
        .order('created_at', { ascending: false })
        .limit(20);

    if (historyError) {
        throw historyError;
    }

    let message = `*Останні 20 вимірювань:*\n\n`;
    history.forEach((record, index) => {
        message += `${index + 1}. Температура: ${record.temp}°C | Вологість:
${record.humidity}% _(${dayjs(record.created_at).format('HH:mm
DD.MM.YYYY')})_\n`;
    });

    await ctx.reply(message, {
        ...Markup.keyboard(['📝 Історія записів', '🎛 Поточні
показники']).resize(),
        parse_mode: 'Markdown'
    })
})

bot.hears("🎛 Поточні показники", async (ctx) => {
    const { data, error } = await supabase
        .from('measurements')
```

```
        .select('*')

    if (error) {
        throw error;
    }
    const record = data[0]

    await ctx.reply(
        `*Температура:* ${record.temp}°C\n*Вологість:* ${record.humidity}%\n_Час
вимірювання: ${dayjs(record.updated_at).format('HH:mm DD.MM.YYYY')}_`,
        { ...Markup.keyboard(['📝 Історія записів', '🎲 Поточні
показники']).resize(), parse_mode: 'Markdown' }
    )
})

process.once('SIGINT', () => {
    bot.stop('SIGINT');
});
process.once('SIGTERM', () => {
    bot.stop('SIGTERM')
});
```

Додаток В

Лістинг коду прошивки

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include <WiFiClient.h>
#include <ArduinoJson.h>
#include "DHT.h"
#include <SoftwareSerial.h>
#include <TinyGPS++.h>

#define DPIN 4
#define DTYPE DHT11

const char *ssid = "temp-humidity-bot";
const char *password = "qwerty12345";
const char *serverIP = "192.168.213.40";
const int serverPort = 3000;

static const int RXPin = D6, TXPin = D7;
static const uint32_t GPSBaud = 9600;

TinyGPSPlus gps;
SoftwareSerial ss(RXPin, TXPin);
WiFiClient client;

DHT dht(DPIN, DTYPE);

void setup(void)
{
  digitalWrite(LED_BUILTIN, HIGH); // turn LED off
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  dht.begin();

  WiFi.begin(ssid, password);
  Serial.print("Connecting to ");
  Serial.println(ssid);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected");
}

void loop(void)
{
  float humidity = dht.readHumidity();
  float temperature = dht.readTemperature();

  if (isnan(humidity) || isnan(temperature))
  {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  Serial.print("Temperature: ");
  Serial.print(temperature, 1);
  Serial.print(" °C");
  Serial.print(" %\t");
```

```
  Serial.print("Humidity: ");
  Serial.println(humidity, 1);

  if (WiFi.status() == WL_CONNECTED)
  {
    HTTPClient http;
    WiFiClient client;

    String url = "http://" + String(serverIP) + ":" + String(serverPort) +
"/save";
    http.begin(client, url);
    http.addHeader("Content-Type", "application/json");

    StaticJsonDocument<200> jsonDoc;
    jsonDoc["humidity"] = humidity;
    jsonDoc["temp"] = temperature;
    String requestBody;
    serializeJson(jsonDoc, requestBody);

    int httpCode = http.POST(requestBody);
    http.end();
  }

  digitalWrite(LED_BUILTIN, LOW);
  delay(100);
  digitalWrite(LED_BUILTIN, HIGH);
  delay(20000);
}
```