Національний лісотехнічний університет України
(повна найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повна найменування інституту, назва факультету(відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: Розроблення мобільного застосунку для управління мікрокліматом в кампусі.

Виконав студент 2 курсу, групи КНС-21

Спеціальності:

122 "Комп'ютерні науки"
(шифр і назва напряму підготовки спеціальності)

Магир О.Б.
(прізвище, ініціали)

Керівники: Проць А.М.
Пірко І.Б.
(прізвище, ініціали)

Рецензент: Мокрицька О.В
(прізвище, ініціали)

Львів-2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій

Кафедра комп'ютерних наук

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри КН

_____ Борецька І. Б.

"10" червня 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Магир Ольга Богданівна
(прізвище, ім'я, по батькові)

1.Тема роботи: Розроблення мобільного застосунку для управління мікрокліматом в кампусі.

керівники роботи: Проць А. М., Пірко І. Б., к.ф-м.н., доцент
(прізвище, ім'я, (з батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від від "15" листопада 2024 року, №С-882

2.Термін подання студентом проекту (роботи) 10 червня 2024 року

3.Вихідні дані до проеекту (роботи) Мобільний застосунку для управління мікрокліматом в кампусі.

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

Розділ 1. Стан проблемної області.

Розділ 2. Інформаційне та математичне забезпечення.

Розділ 3. Програмне та технічне забезпечення.

Висновки. Список використаних джерел. Додатки.

5.Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Підготовка матеріалу до доповіді.

6. Дата видачі завдання 18 листопада 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз літературних джерел щодо управління мікрокліматом, мобільних технологій та протоколів бездротового зв'язку (BLE)	18.11.24 – 28.02.25	Виконано
2	Аналіз предметної області, визначення функціональних та нефункціональних вимог до мобільного застосунку	3.03.25 – 11.03.25	Виконано
3	Проектування архітектури системи (компоненти, взаємодія з BLE та сервером), розробка моделі бази даних та математичних алгоритмів (фільтрація, гістерезис)	12.03.25 – 22.03.25	Виконано
4	Вибір та обґрунтування засобів розробки (React Native, Figma, BLE-бібліотеки), проектування UI/UX дизайн-макетів застосунку	24.03.25 – 30.04.25	Виконано
5	Програмна реалізація мобільного застосунку (UI-компоненти, логіка BLE-зв'язку, навігація, керування станами, функціонал автентифікації)	1.04.25 – 23.05.25	Виконано
6	Оформлення пояснювальної записки, аналіз результатів та підготовка до захисту	26.05.25 – 09.06.25	Виконано

Студент _____ Магир О.Б.
(підпис) (прізвище та ініціали)

Керівники роботи _____ Проц А.М.
(підпис) (прізвище та ініціали)

_____ Пірко І.Б.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота містить 72 сторінок, 20 ілюстрацій, 1 додатки, 10 джерел.

Розроблено мобільний застосунок для моніторингу та керування мікрокліматом у кампусі. Застосунок зчитує дані з датчиків і керує актуаторами через BLE. Описано архітектуру: мобільний клієнт, сервер, база даних. Реалізовано фільтрацію та гістерезис. Створено у React Native, дизайн — у Figma, автентифікація — через Amazon Cognito.

Результати засвідчують ефективність застосунку як зручного інструмента централізованого контролю мікроклімату, що сприяє підвищенню комфорту та енергоефективності.

Ключові слова: мікроклімат, кампус, мобільний застосунок, React Native, Bluetooth Low Energy, BLE, IoT, гістерезис, управління кліматом.

ABSTRACT

The Bachelor's thesis consists of 72 pages, 20 illustrations, 1 appendix, and 10 sources.

A mobile app was developed to monitor and control the microclimate in a campus. It uses BLE to read sensor data and control actuators. The system includes a mobile client, server, and database. Filtering and hysteresis are implemented. Built with React Native, designed in Figma, and uses Amazon Cognito for authentication.

The results confirm the app’s effectiveness in providing centralized and user-friendly microclimate control, contributing to improved comfort and energy efficiency.

Keywords: microclimate, campus, mobile application, React Native, Bluetooth Low Energy, BLE, IoT, hysteresis, climate control.

ТЕХНІЧНЕ ЗАВДАННЯ

Розробляється мобільний застосунок для моніторингу та управління мікрокліматом у приміщеннях університетського кампусу. Застосунок взаємодіє з BLE-пристроями для зчитування даних з датчиків і надсилання команд актуаторам. Серверна частина забезпечує зберігання історії, синхронізацію та управління користувачами з використанням хмарних сервісів.

Застосунок створюється на React Native для Android та iOS з єдиним кодом. Інтерфейс проєктується у Figma. Комунікація з пристроями відбувається через BLE з використанням react-native-ble-plx.

Основний функціонал:.

1. Моніторинг показників мікроклімату в реальному часі.
2. Керування актуаторами (наприклад, зміна температури, увімкнення кондиціонера).
3. Реєстрація, вхід, автентифікація користувачів.
4. Додавання/редагування BLE-пристроїв (із підтримкою QR-сканування).
5. Організація пристроїв за приміщеннями
6. Перегляд історичних даних у вигляді графіків і таблиць.
7. Налаштування сценаріїв автоматизації.
8. Алгоритми фільтрації та гістерезису для надійної роботи обладнання.

Розробка здійснюється у Visual Studio Code з інтеграцією Android Studio/Xcode. Застосунок має забезпечити зручне та ефективне управління мікрокліматом, підвищуючи комфорт і енергоефективність кампусу.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Прикладний програмний інтерфейс (Application Programming Interface).

AWS – Amazon Web Services.

BLE – Bluetooth Low Energy (Bluetooth з низьким споживанням енергії).

BMS – Системи управління будівлями (Building Management Systems).

GATT – Загальний профіль атрибутів (Generic Attribute Profile) – стандарт у BLE, що визначає структуру даних.

HTTP – Протокол передачі гіпертексту (Hypertext Transfer Protocol).

HTTPS – Безпечний протокол передачі гіпертексту (Hypertext Transfer Protocol Secure).

HVAC – Опалення, вентиляція та кондиціонування повітря (Heating, Ventilation, and Air Conditioning).

IDE – Інтегроване середовище розробки (Integrated Development Environment).

IoT – Інтернет речей (Internet of Things).

JS – JavaScript.

JSON – JavaScript Object Notation.

MQTT – Протокол обміну повідомленнями у чергах телеметрії (Message Queuing Telemetry Transport).

SDK – Комплект для розробки програмного забезпечення (Software Development Kit).

UI – Користувацький інтерфейс (User Interface).

UML – Уніфікована мова моделювання (Unified Modeling Language).

ВСТУП

У сучасному світі, що динамічно розвивається, забезпечення комфортних та енергоефективних умов у великих освітніх комплексах, таких як університетські кампуси, набуває надзвичайної актуальності. Мікроклімат у приміщеннях безпосередньо впливає на продуктивність навчання, здоров'я та загальне самопочуття студентів та персоналу. Недостатній контроль над параметрами навколишнього середовища, такими як температура, вологість та якість повітря, може призводити до значних енергетичних втрат, нераціонального використання ресурсів та зниження загального рівня задоволеності інфраструктурою. Існуючі традиційні системи управління часто є застарілими, неефективними та не надають можливості оперативного контролю та персоналізації налаштувань, особливо для кінцевого користувача.

Актуальність розроблення мобільного застосунку для управління мікрокліматом у кампусі полягає у потребі в сучасному, гнучкому та доступному інструменті, який би забезпечував централізований моніторинг та оперативне керування кліматичними параметрами в реальному часі. Такий застосунок дозволить оптимізувати енергоспоживання, підвищити комфорт перебування в приміщеннях та надати користувачам можливість впливати на умови свого оточення.

Об'єктом дослідження є процес управління мікрокліматом у приміщеннях кампусу за допомогою мобільного застосунку.

Предметом дослідження виступають методи та засоби розроблення мобільних застосунків на крос-платформній основі (React Native), алгоритми взаємодії з апаратними пристроями через протокол Bluetooth Low Energy (BLE), а також моделі зберігання та обробки даних мікроклімату.

Метою дипломної роботи є розроблення мобільного застосунку, який забезпечує моніторинг та управління мікрокліматом у приміщеннях кампусу, інтегруючи бездротову взаємодію з BLE-пристроями та синхронізацію даних із серверною частиною для підвищення енергоефективності та комфорту користувачів.

Для досягнення цієї мети у роботі вирішувалися такі завдання:

1. Провести аналіз стану проблемної області та існуючих рішень для управління мікрокліматом.
2. Визначити функціональні та нефункціональні вимоги до мобільного застосунку.
3. Розробити архітектуру системи управління мікрокліматом, включаючи взаємодію з BLE-пристроями та серверною частиною.
4. Спроектувати структуру бази даних для зберігання інформації про користувачів, приміщення, пристрої та історичні дані мікроклімату.
5. Обґрунтувати вибір та описати засоби розробки, зокрема крос-платформну мобільну платформу React Native та інструменти дизайну Figma.
6. Деталізувати принципи роботи технології Bluetooth Low Energy (BLE) та обрані бібліотеки для її інтеграції в мобільний застосунок.
7. Реалізувати програмну частину мобільного застосунку, включаючи розробку UI-компонентів, логіку BLE-зв'язку (сканування, підключення, обмін даними), організацію навігації, керування станами та функціонал автентифікації/реєстрації користувачів.
8. Розробити алгоритми фільтрації даних з сенсорів та застосування гістерезису для ефективного управління актуаторами.
9. Провести тестування та перевірку функціональності розробленого застосунку.

Практична значущість роботи полягає у створенні працездатного прототипу мобільного застосунку, який може бути впроваджений у реальних умовах кампусу для автоматизації управління мікрокліматом. Розроблене рішення може слугувати основою для подальшого розширення функціоналу, інтеграції з іншими системами "розумного кампусу" та підвищення загального рівня комфорту та енергоефективності освітнього закладу.

Таким чином, розроблений мобільний застосунок є важливим кроком у напрямку створення інтелектуальних систем управління інфраструктурою, що забезпечують адаптивність, точність та оптимальне використання ресурсів у динамічному середовищі кампусу.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Актуальність та сучасні виклики

У сучасному освітньому просторі, зокрема в кампусах вищих навчальних закладів, підтримка оптимального мікроклімату є завданням, що набуває дедалі більшої актуальності. Комфортні умови перебування в аудиторіях, бібліотеках, гуртожитках та інших приміщеннях кампусу безпосередньо впливають на працездатність, концентрацію уваги та загальне самопочуття студентів та викладачів. Недостатній контроль температури, вологості та вентиляції може призводити до зниження академічної успішності, погіршення здоров'я та загального рівня задоволеності інфраструктурою. З огляду на тенденції до зростання енергоспоживання, пов'язані з опаленням та кондиціонуванням, оптимізація управління мікрокліматом стає також важливим аспектом екологічної відповідальності та економічної ефективності.

Традиційні системи управління мікрокліматом, як правило, характеризуються недостатньою гнучкістю та відсутністю централізованого контролю. Це часто виражається у відсутності детальної інформації про поточні параметри в кожному окремому приміщенні, неможливості оперативного реагування на зміни зовнішніх або внутрішніх умов, а також відсутністю персоналізованих налаштувань для різних зон кампусу. Ручне регулювання, що вимагає постійної уваги персоналу, є неефективним та призводить до значних втрат енергії. Крім того, складнощі з інтеграцією різнорідного обладнання та відсутність єдиної точки доступу до системи управління ускладнюють її масштабування та модернізацію.

Таким чином, розроблення сучасної системи управління мікрокліматом, що базується на мобільному застосунку, є актуальним завданням, яке дозволить забезпечити централізований моніторинг та оперативне регулювання параметрів у реальному часі. Використання мобільної платформи надасть користувачам зручний та інтуїтивно зрозумілий

інструмент для контролю над умовами у приміщеннях, сприятиме підвищенню комфорту та оптимізації енергоспоживання.

1.2 Огляд існуючих рішень та технологій

Аналіз існуючих підходів до управління мікрокліматом та відповідних технологій є критично важливим для виявлення прогалин у сучасних рішеннях та визначення оптимального вектора розробки. У даний час спостерігається тенденція до впровадження так званих "розумних" систем управління будівлями (Building Management Systems, BMS) у комерційних та освітніх об'єктах. Такі системи зазвичай пропонують централізований контроль над різними інженерними підсистемами, включаючи опалення, вентиляцію, кондиціонування повітря (HVAC), освітлення та безпеку. Комерційні рішення, як правило, є комплексними, проте часто характеризуються високою вартістю впровадження, складністю інтеграції з існуючою інфраструктурою та обмеженими можливостями кастомізації. Застосування цих систем переважно передбачає використання стаціонарних пультів управління або спеціалізованого програмного забезпечення на персональних комп'ютерах.

Паралельно з розвитком BMS активно розвиваються мобільні застосунки, призначені для взаємодії з окремими пристроями або невеликими системами "розумного будинку". Ці застосунки, як правило, забезпечують керування термостатами, розумними розетками, системами освітлення та безпеки через Wi-Fi або Bluetooth-з'єднання. Серед них можна виділити рішення від таких виробників, як Nest, Ecobee, Philips Hue. Їхньою перевагою є простота використання та інтуїтивно зрозумілий інтерфейс, проте вони часто орієнтовані на індивідуального користувача та мають обмежені можливості масштабування для великих об'єктів, таких як кампуси, де необхідно керувати великою кількістю приміщень та різнорідними пристроями.

Щодо технологій, що використовуються для бездротової взаємодії, то переважно застосовуються Wi-Fi та Bluetooth. Wi-Fi забезпечує високу

швидкість передачі даних та широке покриття, проте може бути енергоємним для пристроїв, що працюють від батарей, і вимагає наявності розвиненої мережевої інфраструктури. Технологія Bluetooth, зокрема Bluetooth Low Energy (BLE), вирізняється низьким енергоспоживанням, що є критично важливим для автономних сенсорів, а також спрощеним механізмом підключення. Водночас, BLE має обмежений радіус дії та нижчу швидкість передачі даних порівняно з Wi-Fi, що накладає певні обмеження на його застосування у великих мережах. Наукові розробки часто зосереджуються на вдосконаленні протоколів зв'язку, інтеграції сенсорних мереж та розробці адаптивних алгоритмів управління, проте рідко пропонують повноцінні мобільні інтерфейси, оптимізовані для кінцевого користувача.

Таким чином, огляд існуючих рішень виявляє потребу у створенні мобільного застосунку, який би поєднував у собі зручність та доступність персональних систем "розумного будинку" з масштабованістю та функціональністю корпоративних BMS, враховуючи при цьому специфіку великого освітнього кампусу та забезпечуючи оптимальне використання енергоефективних бездротових технологій, таких як BLE.

Проведений аналіз стану проблемної області дозволяє стверджувати, що підтримка оптимального мікроклімату в кампусах вищих навчальних закладів є критично важливою для забезпечення комфортних умов навчання та праці, а також для досягнення енергоефективності. Існуючі традиційні системи управління часто страждають від відсутності гнучкості, централізованого контролю та можливості оперативного реагування на змінні умови, що призводить до неефективного використання ресурсів та зниження загальної задоволеності користувачів.

Огляд сучасних комерційних та наукових рішень у сфері керування мікрокліматом виявив певні обмеження. Комерційні BMS-системи є ефективними для великих об'єктів, проте їхня висока вартість та складність

інтеграції можуть бути значним бар'єром. Мобільні застосунки для "розумних будинків" забезпечують зручність та простоту використання, але, як правило, не масштабуються для потреб великих кампусів. Щодо бездротових технологій, Wi-Fi пропонує широке покриття, тоді як Bluetooth Low Energy (BLE) є енергоефективним, що робить його привабливим для автономних сенсорів, попри обмежений радіус дії.

Таким чином, існує нагальна потреба у розробленні інноваційного мобільного застосунку, який би поєднував переваги енергоефективних бездротових технологій, таких як BLE, з функціональністю, що дозволяє централізовано моніторити та керувати мікрокліматом у масштабах усього кампусу. Такий застосунок повинен забезпечувати інтуїтивно зрозумілий інтерфейс для кінцевого користувача, оптимізувати енергоспоживання та сприяти створенню комфортного освітнього середовища. Реалізація такого рішення є актуальною як з точки зору підвищення якості послуг, так і з позицій сталого розвитку та ефективного використання ресурсів.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Проектування архітектури застосунку

Проектування архітектури застосунку є фундаментальним етапом у розробці будь-якої складної програмної системи, оскільки воно визначає її структуру, поведінку та взаємодію компонентів. На цьому етапі формується загальна концепція системи, що дозволяє забезпечити її масштабованість, надійність, гнучкість та зручність подальшої розробки та супроводу. Метою проектування архітектури для мобільного застосунку управління мікрокліматом у кампусі є створення чіткої та ефективної моделі, яка враховуватиме всі функціональні та нефункціональні вимоги, а також особливості використовуваних технологій, зокрема Bluetooth Low Energy (BLE).

2.1.1 Визначення загальної архітектури системи

Загальна архітектура системи представляється як трирівнева модель, що складається з:

1. Рівня пристроїв (сенсорна мережа та актуатори): Включає в себе всі фізичні пристрої, що відповідають за моніторинг (датчики температури, вологості, якості повітря) та керування (вентилятори, кондиціонери, опалювальні елементи). Взаємодія на цьому рівні відбувається переважно за допомогою BLE-протоколу.

2. Рівня мобільного застосунку: Це основний інтерфейс взаємодії користувача з системою. Мобільний застосунок здійснює сканування та підключення до BLE-пристроїв, збір даних, надсилання команд керування, візуалізацію інформації та взаємодію з віддаленим сервером для синхронізації даних та налаштувань.

3. Рівня серверної частини та бази даних: Відповідає за централізоване зберігання історичних даних мікроклімату, налаштувань користувачів,

інформації про пристрої та управління доступом. Серверна частина також може реалізовувати складні алгоритми аналізу даних, прогнозування та оптимізації мікроклімату, а також інтеграцію з іншими системами кампусу. Взаємодія між мобільним застосунком та серверною частиною здійснюється за допомогою стандартних веб-протоколів (наприклад, HTTP/HTTPS).

Цей архітектурний підхід дозволяє розділити відповідальність між різними рівнями, підвищити модульність системи та забезпечити її масштабованість. Мобільний застосунок є тонким клієнтом для безпосередньої взаємодії з BLE-пристроями та водночас забезпечує зручний доступ до централізованих даних на сервері.

2.1.2 Розробка функціональних та нефункціональних вимог до мобільного застосунку

Ефективне проектування архітектури неможливе без чіткого визначення вимог до системи.

Функціональні вимоги:

1. Моніторинг параметрів мікроклімату: Застосунок повинен відображати поточні показники температури, вологості, якості повітря з підключених BLE-датчиків у реальному часі.
2. Керування пристроями: Повинна бути реалізована можливість надсилання команд керування (наприклад, увімкнення/вимкнення кондиціонера, встановлення бажаної температури) до BLE-актуаторів.
3. Реєстрація та автентифікація користувачів:Забезпечення безпечного процесу створення облікового запису та входу в систему з можливістю управління правами доступу.
4. Управління пристроями: Можливість додавання, видалення, редагування інформації про пристрої (назва, розташування, тип).
5. Перегляд історії даних: Надання користувачам доступу до історичних даних мікроклімату у вигляді графіків і таблиць для аналізу трендів.

6. Налаштування сценаріїв автоматизації: Можливість створення користувацьких сценаріїв (наприклад, автоматичне увімкнення вентиляції при перевищенні рівня CO2, зниження температури в неробочий час).
7. Сповіщення: Система повинна надсилати сповіщення користувачам про критичні зміни мікроклімату або стан пристроїв.
8. Навігація: Інтуїтивно зрозуміла навігація між різними розділами застосунку (головний екран, список пристроїв, налаштування, історія).

Нефункціональні вимоги:

1. Продуктивність: Застосунок повинен забезпечувати швидку та плавну роботу, мінімальний час відгуку при взаємодії з пристроями та сервером.
2. Надійність: Система має бути стійкою до збоїв, забезпечувати коректну роботу при втраті зв'язку та відновленні.
3. Безпека: Захист даних користувачів та конфіденційності, використання безпечних протоколів передачі даних (особливо при взаємодії з сервером та BLE).
4. Масштабованість: Архітектура повинна дозволяти легке додавання нових приміщень, пристроїв та користувачів без суттєвої модифікації коду.
5. Зручність використання (Usability): Інтуїтивно зрозумілий інтерфейс, легкість освоєння функціоналу для користувачів з різним рівнем підготовки.
6. Сумісність: Застосунок повинен коректно працювати на основних мобільних операційних системах (Android, iOS) та широкому діапазоні пристроїв.
7. Енергоефективність: Оптимізація використання ресурсів мобільного пристрою (особливо енергоспоживання при роботі з BLE).

2.1.3 Проектування основних компонентів та їх взаємодії

Для візуалізації архітектури системи та відображення взаємодії між її основними компонентами буде використана діаграма архітектури системи, яка демонструє потік даних та зв'язок між апаратним та програмним забезпеченням.

На цій діаграмі, основними вузлами та компонентами системи є:

1. Bluetooth Low Energy Devices (BLE пристрої): Це кінцеві пристрої, що відповідають за моніторинг параметрів мікроклімату (наприклад, датчики температури, вологості, освітлення, як символічно зображені піктограмами термометра та лампочки) та/або за керування (актуатори, що можуть відповідати за вентиляцію, опалення тощо). Ці пристрої використовують BLE-протокол для комунікації, забезпечуючи низьке енергоспоживання. Взаємодія між BLE-пристроями та мобільним застосунком відбувається безпосередньо через Bluetooth.
2. Мобільний пристрій (Smartphone): Це центральний елемент користувацького інтерфейсу, на якому працює мобільний застосунок. Він виступає посередником між BLE-пристроями та хмарною платформою.
3. Взаємодія з BLE-пристроями: Мобільний пристрій встановлює бездротове з'єднання через Bluetooth (піктограма Bluetooth) з BLE пристроями для отримання даних (сенсорні показники) та надсилання команд управління. Для цієї комунікації може використовуватись протокол MQTT, що дозволяє організувати ефективний обмін повідомленнями.
4. Взаємодія з хмарною платформою: Мобільний пристрій підключається до хмарних сервісів через стандартні інтернет-протоколи (Wi-Fi), використовуючи MQTT, HTTP або Websocket для синхронізації даних, зберігання історії, управління користувачами та доступом.

5. Android/iOS SDKs: Ці компоненти символізують набори для розробки програмного забезпечення, що використовуються для створення крос-платформного мобільного застосунку на базі React Native. Вони забезпечують можливість доступу до рідних функцій пристрою, включаючи Bluetooth-модулі.
6. Amazon Cognito: Це сервіс Amazon Web Services (AWS), що відповідає за управління ідентифікацією та доступом користувачів. Він забезпечує безпечну реєстрацію, вхід та авторизацію користувачів мобільного застосунку, надаючи токени доступу для взаємодії з іншими сервісами AWS.
7. AWS IoT: Ця хмарна платформа від AWS призначена для керування пристроями Інтернету речей (IoT). Вона виступає як центральний брокер повідомлень для даних, що надходять від мобільного застосунку. AWS IoT забезпечує надійну передачу даних, обробку подій та інтеграцію з іншими сервісами AWS для подальшого зберігання, аналізу та візуалізації.

Таким чином, запропонована архітектура демонструє взаємодію між трьома основними рівнями: фізичними BLE-пристроями, мобільним застосунком як шлюзом та інтерфейсом, та хмарною платформою AWS для централізованого управління та зберігання даних. Цей підхід забезпечує гнучкість, масштабованість та безпеку системи управління мікрокліматом у кампусі (рис. 2.1).

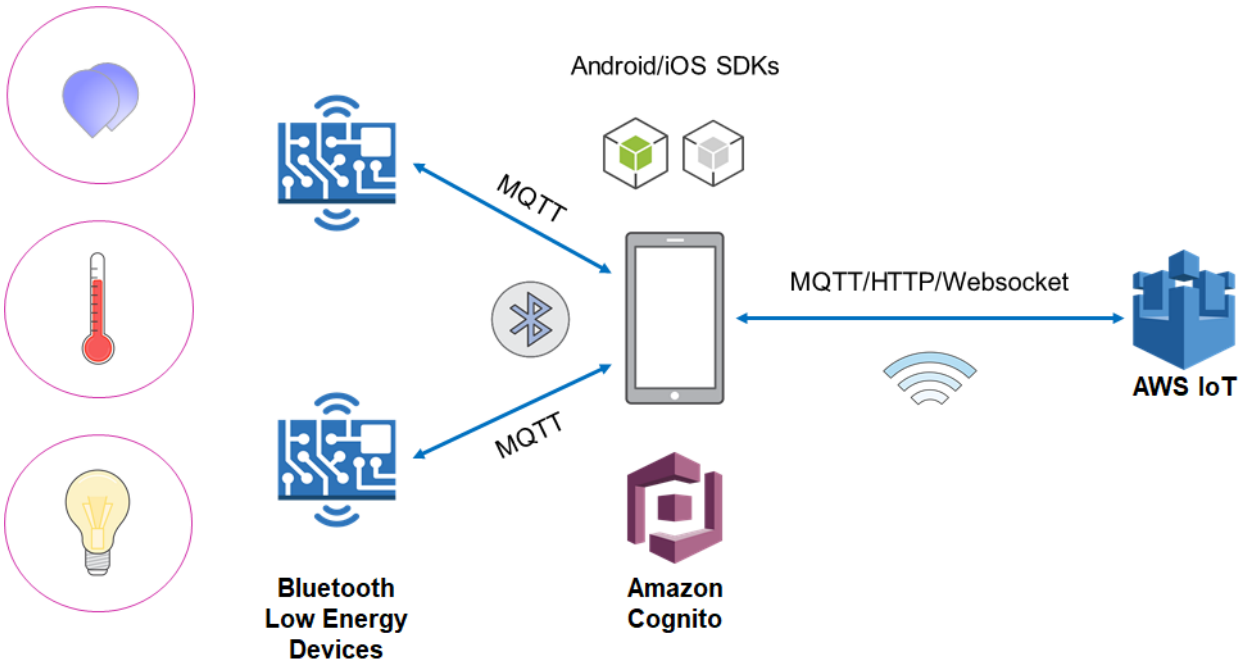


Рисунок 2.1 – Діаграма архітектури системи

2.2 Проектування бази даних

Ефективне функціонування системи управління мікрокліматом вимагає надійної та структурованої системи зберігання даних, яка забезпечить цілісність, доступність та масштабованість інформації. Проектування бази даних є фундаментальним етапом, що дозволяє організувати дані про користувачів, приміщення, пристрої, їхні налаштування та історичні показники мікроклімату. Обрана реляційна модель даних забезпечує логічні зв'язки між сутностями, мінімізує дублювання інформації та оптимізує продуктивність запитів.

Схема бази даних розроблена з урахуванням потреби у зберіганні інформації про:

1. Користувачів системи та їхні облікові дані.
2. Фізичні приміщення, що знаходяться під управлінням.
3. Пристрої моніторингу та керування (датчики та актуатори) з їхніми характеристиками.
4. Типи розкладів для пристроїв.

5. Детальні розклади роботи пристроїв.
6. Агреговану статистику щодо приміщень та окремих пристроїв.

Логічна структура бази даних системи управління мікрокліматом демонструє сутності, їхні атрибути та встановлені зв'язки (рис. 2.2).

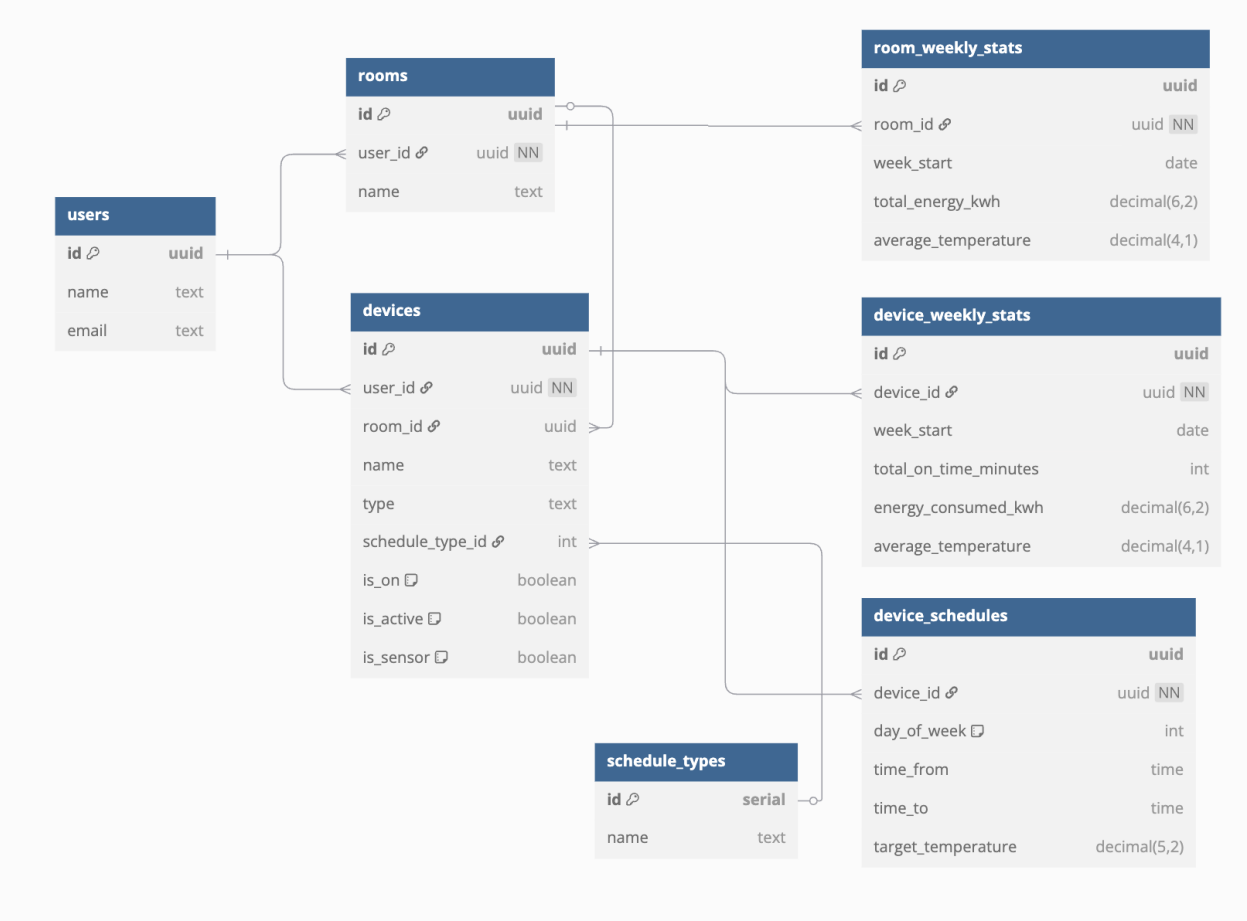


Рисунок 2.2 – Схема бази даних системи управління мікрокліматом

Основними сутностями бази даних є:

1. Користувачі (users): Ця таблиця зберігає основну інформацію про зареєстрованих користувачів системи.

1.1 *id* (uuid, primary key): Унікальний ідентифікатор користувача.

1.2 *name* (text): Ім'я користувача.

1.3 *email* (text, unique): Електронна пошта користувача, яка також є унікальним ідентифікатором для входу.

2. Приміщення (rooms): Таблиця, що містить інформацію про кожне приміщення в кампусі, за яким здійснюється моніторинг або управління.

2.1 *Id* (uuid, primary key): Унікальний ідентифікатор приміщення.

2.2 *user_id* (uuid, not null, foreign key): Ідентифікатор користувача, який володіє або відповідає за це приміщення. Встановлює зв'язок "один-до-багатьох" з таблицею `users`.

2.3 *name* (text): Назва приміщення (наприклад, "Аудиторія 205").

3. Типи розкладів (schedule_types): Допоміжна таблиця для визначення типів розкладів, що можуть бути застосовані до пристроїв.

3.1 *id* (serial, primary key): Унікальний автоматично інкрементований ідентифікатор типу розкладу.

3.2 *name* (text, unique): Назва типу розкладу (наприклад, "Щоденний", "Тижневий", "Ручний").

4. Пристрої (devices): Основна таблиця для зберігання інформації про всі датчики та актуатори, інтегровані в систему.

4.1 *id* (uuid, primary key): Унікальний ідентифікатор пристрою.

4.2 *user_id* (uuid, not null, foreign key): Ідентифікатор користувача, який володіє пристроєм. Встановлює зв'язок "один-до-багатьох" з таблицею `users`.

4.3 *room_id* (uuid, foreign key): Ідентифікатор приміщення, в якому встановлено пристрій. Встановлює зв'язок "один-до-багатьох" з таблицею `rooms`.

4.4 *name* (text): Зрозуміла назва пристрою.

4.5 *type* (text): Тип пристрою (наприклад, "температурний датчик", "кондиціонер", "вентилятор").

4.6 *schedule_type_id* (int, foreign key): Ідентифікатор типу розкладу, що застосовується до цього пристрою. Встановлює зв'язок "один-до-багатьох" з таблицею `schedule_types`.

4.7 *is_on* (boolean, default: false): Поточний стан увімкнення/вимкнення пристрою (для актуаторів).

4.8 *is_active* (boolean, default: false): Поточний статус активності пристрою в системі.

4.9 *is_sensor* (boolean, default: false): Флаг, що вказує, чи є пристрій лише сенсором (true), чи також актуатором (false).

5. Розклади пристроїв (device_schedules): Ця таблиця деталізує розклади роботи для керуючих пристроїв.

5.1 *id* (uuid, primary key): Унікальний ідентифікатор запису розкладу.

5.2 *device_id* (uuid, not null, foreign key): Ідентифікатор пристрою, до якого застосовується розклад. Встановлює зв'язок "один-до-багатьох" з таблицею `devices`.

5.3 *day_of_week* (int): День тижня, на який поширюється розклад (0 = неділя, 6 = субота).

5.4 *time_from* (time): Час початку дії розкладу.

5.5 *time_to* (time): Час закінчення дії розкладу.

5.6 *target_temperature* (decimal(5,2)): Бажана температура для даного часового проміжку.

6. Щотижнева статистика пристроїв (device_weekly_stats): Таблиця для зберігання агрегованої щотижневої статистики по кожному пристрою.

6.1 *id* (uuid, primary key): Унікальний ідентифікатор запису статистики.

6.2 *device_id* (uuid, not null, foreign key): Ідентифікатор пристрою. Встановлює зв'язок "один-до-багатьох" з таблицею `devices`.

6.3 *week_start* (date): Дата початку тижня, за який зібрана статистика.

6.4 *total_on_time_minutes* (int): Загальний час роботи пристрою за тиждень у хвилинах.

6.5 *energy_consumed_kwh* (decimal(6,2)): Спожита енергія за тиждень у кВт/год.

6.6 *average_temperature* (decimal(4,1)): Середня температура, зафіксована пристроєм за тиждень.

7. Щотижнева статистика приміщень (room_weekly_stats): Таблиця для зберігання агрегованої щотижневої статистики по кожному приміщенню.

7.2 id (uuid, primary key): Унікальний ідентифікатор запису статистики.

7.3 room_id` (uuid, not null, foreign key): Ідентифікатор приміщення. Встановлює зв'язок "один-до-багатьох" з таблицею `rooms`.

7.4 week_start (date): Дата початку тижня, за який зібрана статистика.

7.5 total_energy_kwh (decimal(6,2)): Загальне споживання енергії в приміщенні за тиждень у кВт/год.

7.6 average_temperature (decimal(4,1)): Середня температура в приміщенні за тиждень.

Зазначені зв'язки між таблицями (ForeignKey) забезпечують реляційну цілісність даних, гарантуючи, що записи в дочірніх таблицях завжди посилаються на існуючі записи в батьківських таблицях. Використання UUID для первинних ключів забезпечує глобальну унікальність ідентифікаторів, що зручно для розподілених систем та масштабування. Ця структура бази даних є

міцною основою для ефективного збору, зберігання та аналізу даних, що є критично важливим для функціонування системи управління мікрокліматом.

2.3 Математичне та алгоритмічне забезпечення

Ефективне функціонування системи управління мікрокліматом вимагає не лише надійного збору даних, а й їхньої інтелектуальної обробки. Математичне та алгоритмічне забезпечення системи призначене для перетворення сирих показань датчиків на корисну, достовірну інформацію, виявлення закономірностей та підтримки прийняття рішень щодо оптимізації параметрів мікроклімату. Взаємодія між фізичними датчиками, мобільним застосунком та серверною API передбачає багатоетапну обробку даних.

Збір даних з BLE-датчиків відбувається циклічно з заданою періодичністю. Кожне отримане значення температури, вологості, рівня CO2 тощо є потенційно вразливим до шуму, випадкових похибок або короткочасних відхилень, які можуть спотворити реальну картину мікроклімату. Для забезпечення достовірності та стабільності даних, що надсилаються до серверної API, а потім відображаються у мобільному застосунку, застосовуються алгоритми фільтрації.

1. Фільтрація за порогом: На первинному етапі збору даних застосовується порогова фільтрація. Якщо отримане значення виходить за межі заздалегідь визначеного діапазону фізично можливих або очікуваних значень для даного типу датчика, воно може бути відхилене або позначене як аномалія. Цей метод допомагає ігнорувати дані, які є явно помилковими через технічні збої або екстремальні зовнішні впливи. Наприклад, якщо датчик температури передає показник у 150°C, таке значення буде відфільтровано.

Ці алгоритми можуть реалізуватися на рівні мобільного застосунку (для відображення даних у реальному часі) або на серверній стороні (для зберігання очищених історичних даних).

Після первинної фільтрації дані стають придатними для подальшого аналізу та застосування математичних моделей, що дозволяють не лише відображати поточний стан, а й надавати інтелектуальну підтримку для управління.

2. Застосування гістерезису для управління актуаторами:

Гістерезис є ключовим алгоритмічним рішенням, що забезпечує стабільність та енергоефективність системи управління температурою, запобігаючи надмірному перемиканню опалювальних або охолоджувальних пристроїв. Він застосовується при заданні бажаної температури в кімнаті або для панелі опалення та інших актуаторів [10].

Суть гістерезису полягає у створенні двох різних порогових значень для вмикання та вимикання пристрою, замість одного жорсткого. Це дозволяє уникнути циклічного "тремтіння" системи, коли пристрій постійно вмикається та вимикається через незначні коливання температури навколо заданої точки. Без гістерезису, якщо бажана температура, наприклад, 22°C, то пристрій опалення вимикався б на 22.0°C і одразу вмикався б на 21.9°C, що призводило б до його швидкого зносу та високого енергоспоживання.

Користувач задає бажану цільову температуру $Тцільова$ та значення гістерезису ΔT, наприклад, 1°C. Це значення ΔT формує діапазон навколо цільової температури.

2.1 Для системи опалення:

Опалення вмикається, коли поточна температура $Тпоточна$ падає нижче порогу (2.1).

$$Твмикання = Тцільова - \Delta T \quad (2.1)$$

Опалення вимикається, коли $Тпоточна$ досягає або перевищує порогову (2.2).

$$Твимкнення = Тцільова \quad (2.2)$$

2.2 Для системи охолодження (кондиціонера):

Охолодження вмикається, коли $Tпоточна$ піднімається вище порогу (2.3).

$$Tвмикання = Tцільова + \Delta T \quad (2.3)$$

Охолодження вимикається, коли $Tпоточна$ падає до або нижче порогу (2.4).

$$Tвимкнення = Tцільова \quad (2.4)$$

Приклад: Якщо $Tцільова = 22°C$ і $\Delta T = 1°C$: Для опалення: пристрій ввімкнеться, якщо температура опуститься до 21°C або нижче, і вимкнеться, коли досягне 22°C. Для охолодження: пристрій ввімкнеться, якщо температура підніметься до 23°C або вище, і вимкнеться, коли опуститься до 22°C.

Призначення значення гістерезису буде відбуватись при налаштуванні параметрів панелі опалення, кондиціонера, або інших кліматичних пристроїв через мобільний застосунок. Користувач вказує бажану температуру, а система автоматично застосовує визначений діапазон гістерезису для ефективного управління. Ця логіка може бути реалізована як на серверній API (для централізованого управління та синхронізації), так і безпосередньо у прошивці BLE-актуаторів для забезпечення швидкого реагування, навіть при тимчасовій втраті зв'язку з сервером або мобільним застосунком. Завдяки гістерезису, система забезпечує більш стабільний мікроклімат, зменшує знос обладнання та оптимізує енергоспоживання.

Дані, зібрані з датчиків та відфільтровані, надсилаються до Серверної API (Application Programming Interface). Серверна API є посередником між мобільним застосунком та базою даних, а також містить логіку для застосування правил управління (наприклад, гістерезис). Після обробки на сервері, актуальні показники мікроклімату, історичні дані або підтвердження команд управління надсилаються назад до мобільного застосунку для візуалізації та взаємодії з користувачем. Цей двосторонній потік даних забезпечує актуальність інформації та оперативність керування системою [8].

У рамках розділу було детально розглянуто процеси математичного моделювання та проектування основних структурних елементів системи управління мікрокліматом. Запропонована трирівнева архітектура системи, що охоплює рівні фізичних пристроїв (сенсорна мережа та актуатори), мобільного застосунку та серверної частини з базою даних, забезпечує чітке розділення відповідальності, модульність та масштабованість рішення. Кожен рівень відіграє ключову роль у забезпеченні функціональності: BLE-пристрої для збору та виконання команд, мобільний застосунок як інтуїтивний інтерфейс, а хмарна платформа AWS (з AWS IoT та Amazon Cognito) для централізованого управління, зберігання даних та забезпечення безпеки. Було визначено як функціональні (моніторинг, керування, реєстрація, історія, сценарії, сповіщення, навігація), так і нефункціональні вимоги (продуктивність, надійність, безпека, масштабованість, зручність, сумісність, енергоефективність), що лягли в основу подальшого проектування.

Проектування бази даних здійснено з урахуванням специфіки даних мікроклімату та потреб користувачів. Визначено сім основних сутностей – users, rooms, schedule_types, devices, device_schedules, device_weekly_stats, та room_weekly_stats – з відповідними атрибутами та реляційними зв'язками. Ця структура дозволяє ефективно зберігати інформацію про користувачів, об'єкти моніторингу (приміщення), керуючі пристрої, а також агреговані та історичні дані для аналізу та звітності. Використання UUID для первинних ключів підкреслює гнучкість системи до масштабування.

Нарешті, математичне та алгоритмічне забезпечення системи зосереджене на підвищенні достовірності даних та ефективності управління. Запропоновано алгоритм порогової фільтрації для первинного очищення даних від аномальних значень. Особлива увага приділена застосуванню гістерезису в алгоритмах управління актуаторами (системами опалення та охолодження). Це дозволяє уникнути "тремтіння" системи, продовжити термін служби обладнання та значно підвищити енергоефективність за рахунок

зменшення частоти циклів увімкнення/вимкнення. Механізм гістерезису, що задається користувачем через мобільний застосунок, інтегрується в логіку управління як на серверному рівні (API), так і потенційно безпосередньо в BLE-актуаторах.

Усі ці елементи – архітектура, модель даних та алгоритмічні підходи – створюють міцну основу для розробки стабільного, ефективного та зручного мобільного застосунку для управління мікрокліматом у кампусі.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вибір та обґрунтування засобів розробки

3.1.1 Середовище розробки: Visual Studio Code

Вибір інтегрованого середовища розробки (IDE) є одним з першочергових рішень, що впливає на продуктивність, ефективність та зручність процесу створення програмного забезпечення. Для розробки мобільного застосунку управління мікрокліматом було обрано Visual Studio Code (VS Code) – безкоштовний, легкий та потужний редактор коду з відкритим вихідним кодом від Microsoft. Це рішення обґрунтовується рядом ключових переваг, які роблять його ідеальним вибором для крос-платформної мобільної розробки на React Native [7].

На відміну від повноцінних IDE, таких як Android Studio або Xcode, які є ресурсномісткими та орієнтовані на нативну розробку, VS Code надає оптимальний баланс функціональності, швидкості та гнучкості. Він забезпечує широку підтримку JavaScript та TypeScript, які є основними мовами для React Native, пропонуючи розширені можливості автодоповнення коду (IntelliSense), підсвічування синтаксису, рефакторингу та виявлення помилок у реальному часі. Це значно прискорює написання коду та зменшує ймовірність синтаксичних помилок на ранніх етапах.

Однією з головних переваг VS Code для React Native розробки є його глибока інтеграція з екосистемою мобільної розробки через систему розширень. Завдяки встановленню відповідних плагінів, таких як "React Native Tools" або "ES7+ React/Redux/GraphQL/React-Native snippets", VS Code трансформується у повноцінне середовище для React Native. Ці розширення надають доступ до спеціалізованих команд, що дозволяють:

1. Запускати Metro Bundler: Інструмент, що компілює JavaScript-код у формат, зрозумілий для мобільних пристроїв. Це відбувається автоматично при запуску застосунку з VS Code.
2. Інтегруватися з Android Studio та Xcode: VS Code не замінює ці нативні IDE, але гармонійно з ними взаємодіє. Для запуску застосунку на Android-емуляторі або реальному пристрої, VS Code використовує інструменти командного рядка Android SDK (які встановлюються разом з Android Studio), а саме `adb` (Android Debug Bridge). Аналогічно, для iOS-платформи, VS Code взаємодіє з `xcodebuild` та іншими інструментами Xcode, дозволяючи запускати симулятор iOS або тестувати на реальному пристрої Apple. Таким чином, розробник може ініціювати запуск застосунку на обраному емуляторі/симуляторі безпосередньо з терміналу VS Code або за допомогою спеціальних команд розширень, які делегують завдання відповідним нативним SDK.
3. Налагоджувати код:VS Code надає потужні засоби для налагодження React Native застосунків, дозволяючи встановлювати точки зупинки, покроково виконувати код, переглядати значення змінних та відстежувати стек викликів. Це інтегрується з можливостями Chrome DevTools для налагодження JavaScript-частини застосунку.
4. Керувати залежностями: Вбудований термінал VS Code дозволяє зручно виконувати команди `npm` або `yarn` для встановлення та оновлення бібліотек та залежностей проєкту, а також виконувати інші операції командного рядка, що є стандартними для React Native розробки.
5. Контроль версій: VS Code має вбудовану підтримку Git, що дозволяє легко керувати версіями коду, робити коміти, працювати з гілками та синхронізувати зміни з віддаленими репозиторіями, забезпечуючи ефективну командну роботу.

Таким чином, Visual Studio Code, у поєднанні з його розширеною екосистемою плагінів та інтеграцією з нативними SDK (Android Studio,

Xcode), є оптимальним середовищем для розробки крос-платформних мобільних застосунків на React Native, забезпечуючи високу продуктивність та зручність на всіх етапах життєвого циклу проекту.

3.1.2 Крос-платформна мобільна платформа: React Native

Вибір основної технології розробки мобільного застосунку є фундаментальним рішенням, що визначає ефективність, якість та терміни реалізації проекту. Для системи управління мікрокліматом було обрано React Native – відкритий фреймворк від Meta (раніше Facebook), що дозволяє розробляти мобільні застосунки для платформ iOS та Android, використовуючи JavaScript та React. Це рішення зумовлене унікальним поєднанням крос-платформних можливостей та доступу до нативних функцій пристроїв, що є критично важливим для взаємодії з BLE-обладнанням та забезпечення високої продуктивності інтерфейсу [1].

React Native працює на основі так званого "JavaScript-мосту" (JavaScript Bridge), який є ключовим елементом його архітектури. На відміну від гібридних фреймворків, що рендерять веб-перегляди (WebView), React Native компілює JavaScript-код у рідні компоненти користувацького інтерфейсу, що забезпечує високу продуктивність та "рідний" вигляд застосунку. Архітектура включає наступні основні компоненти:

1. Поток JavaScript (JavaScript Thread): Тут виконується весь код React Native, написаний на JavaScript/TypeScript. Він відповідає за логіку застосунку, управління станом та обробку подій.
2. Рідний потік (Native Thread): Цей потік виконує код, написаний на Objective-C/Swift (для iOS) або Java/Kotlin (для Android). Він відповідає за рендеринг рідних UI-компонентів, взаємодію з апаратним забезпеченням та системними API.
3. Міст (Bridge): Це механізм, що забезпечує асинхронну комунікацію між JavaScript та рідними потоками. Він серіалізує дані, що передаються між

потоками, і дозволяє JavaScript-частині "спілкуватися" з нативною платформою та викликати її функції. Завдяки мосту, розробник React Native може використовувати нативні модулі, що дає доступ до Bluetooth, камери, геолокації та інших системних можливостей, які не доступні напряму з JavaScript.

4. Тіньовий потік (Shadow Thread): Цей потік, що працює на C++, відповідає за обчислення макета (layout) для рідних UI-компонентів, використовуючи Flexbox. Результати цих обчислень передаються до рідного потоку для рендерингу.

З 2023 року React Native активно переходить на нову архітектуру, що включає **Fabric** (новий механізм рендерингу) та **TurboModules** (оптимізований механізм рідних модулів) та **New Architecture Renderer.** Ці зміни спрямовані на покращення продуктивності, зменшення накладних витрат на мості та спрощення взаємодії між JavaScript та рідним кодом.

Ключові переваги React Native:

1. Крос-платформність: Це основна перевага, що дозволяє писати єдиний codebase на JavaScript/TypeScript, який може бути розгорнутий як на Android, так і на iOS. Це значно скорочує час та вартість розробки, оскільки немає потреби в підтримці двох окремих команд розробників для кожної платформи. Для проекту управління мікрокліматом це означає, що єдиний застосунок зможе працювати на більшості мобільних пристроїв користувачів кампусу.
2. Використання рідних компонентів: На відміну від деяких гібридних фреймворків, React Native рендерить справжні, рідні компоненти інтерфейсу. Це забезпечує застосунку високу продуктивність, плавність анімацій та відповідність рекомендаціям UI/UX кожної платформи. Користувачі відчувають, що працюють з нативним застосунком, а не з веб-сторінкою.

3. Fast Refresh (Швидке оновлення): Ця функція дозволяє розробникам миттєво бачити зміни в коді без необхідності повної перезавантаження застосунку або втрати поточного стану. Це суттєво прискорює ітераційний процес розробки та налагодження.
4. Гнучкість та розширюваність: React Native підтримує безліч сторонніх бібліотек, які розширюють його можливості. Для взаємодії з BLE-пристроями існують спеціалізовані бібліотеки (наприклад, `react-native-ble-plx` або `react-native-bluetooth-le`), які надають високорівневий API для сканування, підключення та обміну даними з BLE-пристроями. У випадках, коли стандартного функціоналу недостатньо, існує можливість написання власних рідних модулів на Java/Kotlin (для Android) або Objective-C/Swift (для iOS), що забезпечує максимальну гнучкість.
5. Велика спільнота та екосистема: Завдяки широкій популярності React та React Native, існує велика спільнота розробників, що активно підтримує фреймворк, створює нові інструменти, бібліотеки та надає підтримку. Це значно спрощує пошук рішень для потенційних проблем та прискорює процес розробки.

Можливості для реалізації інтерфейсу та функціоналу: React Native дозволяє реалізувати весь необхідний функціонал для управління мікрокліматом:

1. Інтуїтивний інтерфейс: Створення сучасних та адаптивних UI-компонентів для відображення даних з датчиків (температура, вологість, CO2), керування актуаторами (кнопки, слайдери, перемикачі).
2. Взаємодія з BLE: Можливість сканування та виявлення BLE-пристроїв у кампусі, підключення до них, читання їхніх характеристик (показників датчиків) та запис у них (команди керування).

3. Управління станом: Ефективне керування глобальним та локальним станом застосунку для синхронізації даних між UI, BLE-модулем та серверною частиною.
4. Навігація: Впровадження складної навігації між різними екранами (список приміщень, деталі пристрою, історія, налаштування, автоматизація).
5. Інтеграція з хмарними сервісами: Забезпечення взаємодії з серверною частиною (AWS IoT, база даних) для зберігання історичних даних, завантаження налаштувань, автентифікації користувачів та управління правами доступу.

Таким чином, React Native є оптимальним вибором для розробки мобільного застосунку управління мікрокліматом, забезпечуючи високу продуктивність, ефективність розробки та можливість доступу до всіх необхідних нативних та хмарних функцій.

3.1.3 Інструменти дизайну: Figma

Для етапу проектування дизайну мобільного застосунку управління мікрокліматом було обрано Figma – потужний хмарний інструмент для дизайну інтерфейсів та прототипування. Вибір Figma обґрунтований її можливостями, що є критичними для сучасного процесу дизайну [3].

1. Хмарна природа та спільна робота: Figma дозволяє всій команді працювати над проектом одночасно в реальному часі, що забезпечує швидкий зворотний зв'язок та ефективну взаємодію. Це також надає гнучкість доступу до файлів з будь-якого пристрою.
2. Прототипування та тестування UX: Інструмент надає розширені можливості для створення інтерактивних прототипів. Це дозволяє імітувати потік взаємодії користувача з застосунком (переходи між екранами, реакції на дії), що є критично важливим для раннього тестування зручності використання та виявлення проблем до початку

кодування. Наприклад, було змодельовано логіку зміни температури та перемикання режимів пристроїв (рис. 3.1).

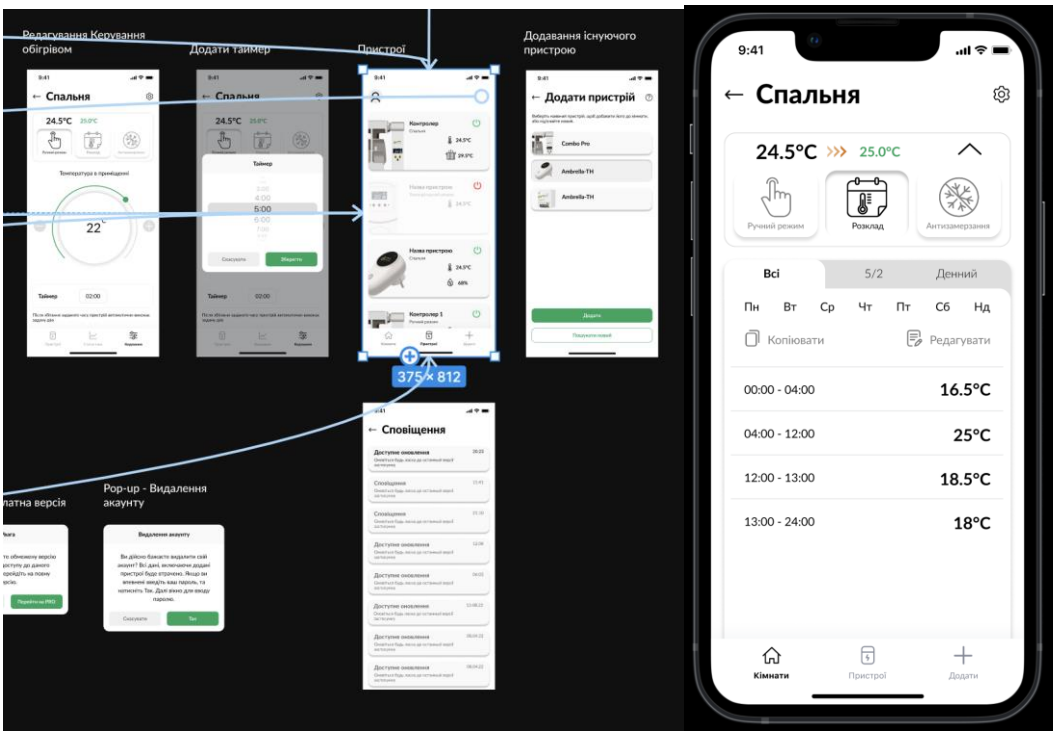


Рисунок 3.1 – Прототипування інтерфейсу

3. Системи компонентів та стилів: Figma підтримує створення багаторазових UI-компонентів та стилів (кольори, типографіка), що значно прискорює процес дизайну та забезпечує консистентність інтерфейсу. Це дозволяє швидко вносити зміни та стандартизувати вигляд елементів (рис. 3.2, 3.3).

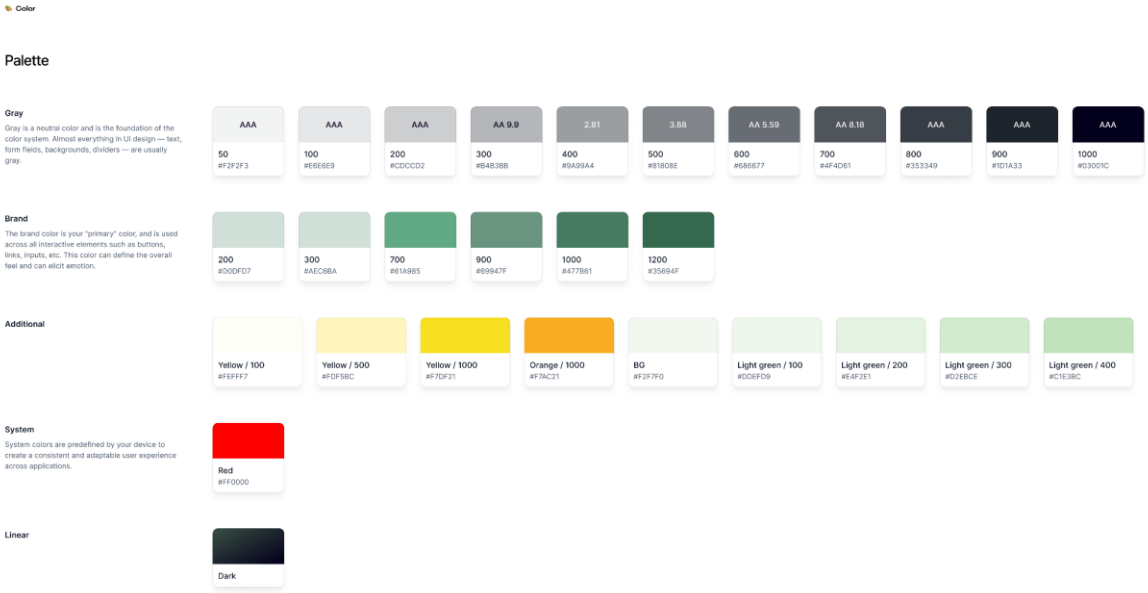


Рисунок 3.2 – Система кольорів

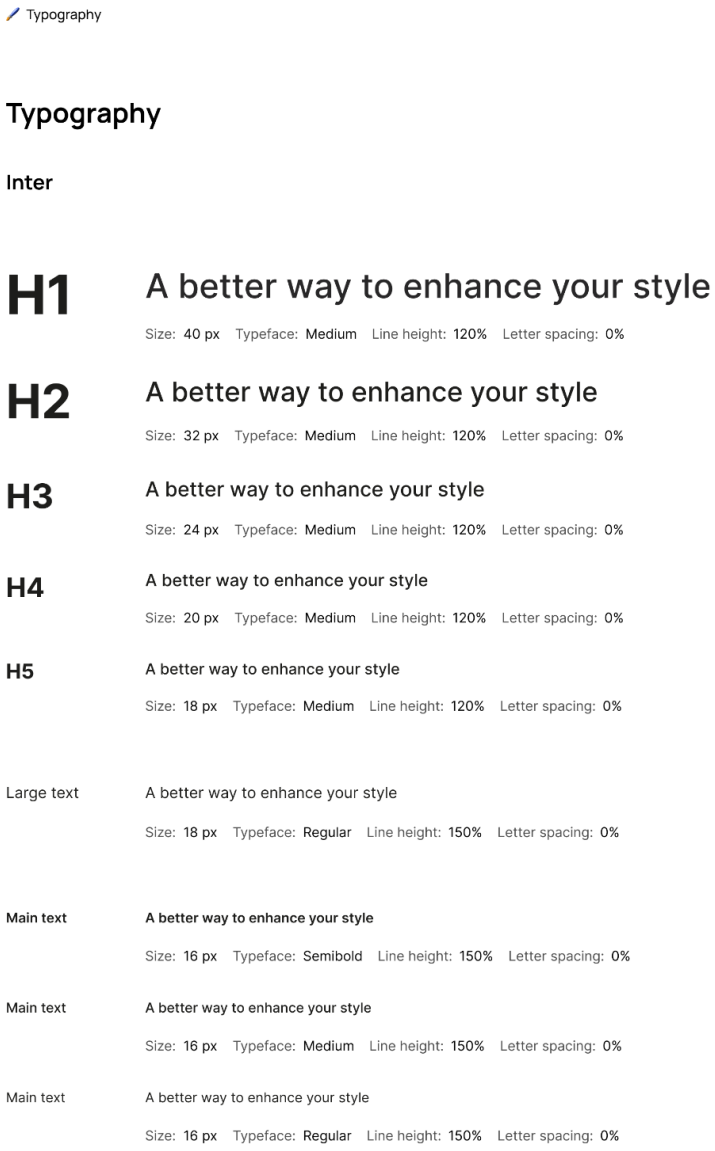


Рисунок 3.3 – Типографічна схема

Таким чином, Figma дозволила створити візуально привабливий та функціональний дизайн, який став надійною основою для подальшої програмної реалізації застосунку на React Native.

3.2 Технологія Bluetooth Low Energy (BLE)

Технологія Bluetooth Low Energy (BLE) є ключовим компонентом архітектури системи управління мікрокліматом, забезпечуючи бездротовий зв'язок між мобільним застосунком та фізичними датчиками/актуаторами. Її

вибір обумовлений енергоефективністю, що критично важливо для автономних пристроїв, що працюють від батарей [2].

3.2.1 Принципи роботи BLE та його роль у системі

BLE – це бездротова технологія короткого радіусу дії, оптимізована для низького енергоспоживання, що дозволяє пристроям працювати роками від однієї батареї. Вона відрізняється від класичного Bluetooth спрощеним стеком протоколів, призначеним для періодичного або потокового обміну невеликими обсягами даних.

Основними концепціями BLE є:

1. GATT-профіль (Generic Attribute Profile): Визначає структуру даних та спосіб їхнього обміну між BLE-пристроями. GATT базується на концепції "сервер-клієнт", де BLE-пристрій (сервер) надає певні дані, а мобільний пристрій (клієнт) їх запитує або записує (рис. 3.4).

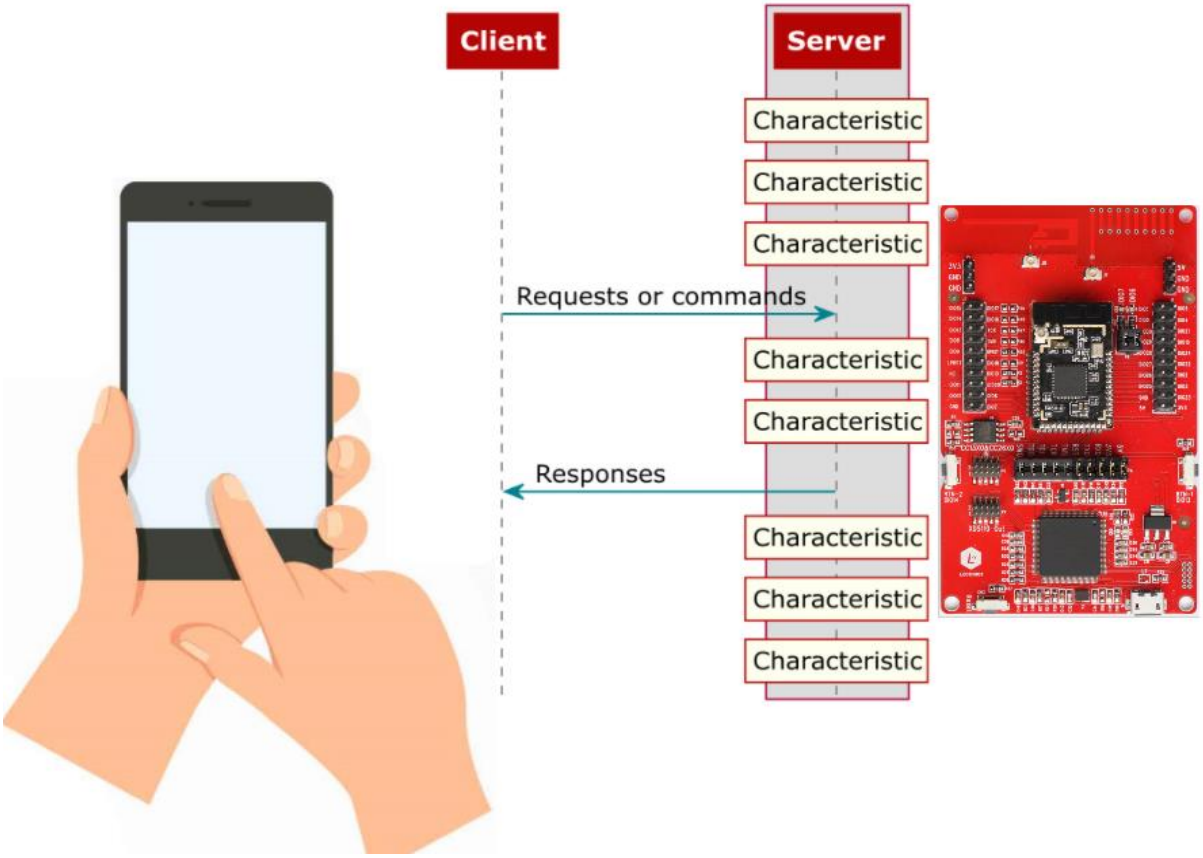


Рисунок 3.4 – Діаграма GATT-профілю

2. Сервіси (Services): Логічні групи характеристик, що об'єднують пов'язані дані та функціонал пристрою. Кожен сервіс має унікальний UUID (Universally Unique Identifier). Наприклад, сервіс "Health Thermometer Service" містить характеристики, що стосуються температури.
3. Характеристики (Characteristics): Основні одиниці даних у GATT. Кожна характеристика має UUID, значення (Value) та властивості (Properties), які визначають, чи можна її читати (Read), записувати (Write), чи підписуватися на її зміни (Notify/Indicate). Читання (Read): Дозволяє клієнту отримати поточне значення характеристики (наприклад, поточну температуру). Запис (Write): Дозволяє клієнту змінити значення характеристики (наприклад, надіслати команду керування). Нотифікації/Індикації (Notify/Indicate): Дозволяють серверу спонтанно надсилати оновлення значення характеристики до клієнта, без запиту з боку клієнта. Це ідеально підходить для потокового отримання даних з датчиків.

Роль BLE у системі:

У системі управління мікрокліматом, BLE виконує роль первинного бездротового інтерфейсу для:

1. Моніторингу: Безпосереднього збору даних з BLE-датчиків (температура, вологість, CO2).
2. Керування: Надсилання команд до BLE-актуаторів (наприклад, увімкнення/вимкнення, встановлення режимів).
3. Конфігурації: Передачі облікових даних Wi-Fi від мобільного застосунку до пристроїв, що підтримують Wi-Fi, для їхнього підключення до мережі та подальшої взаємодії з хмарною платформою.

3.2.2 Огляд BLE-бібліотек для React Native

Для інтеграції BLE-функціоналу в крос-платформний React Native застосунок використовуються сторонні бібліотеки, які надають JavaScript-інтерфейс до нативних BLE API (Core Bluetooth на iOS та Android Bluetooth API).

react-native-ble-plx: Це одна з найпопулярніших та найнадійніших бібліотек для роботи з BLE у React Native. Вона надає повний спектр функцій для сканування, підключення, виявлення сервісів/характеристик, читання, запису та підписки на нотифікації. Її архітектура добре продумана, забезпечуючи стабільну роботу та широкі можливості налагодження. Вибір цієї бібліотеки для проекту зумовлений її активною підтримкою, гнучкістю та наявністю детальної документації.

Інші бібліотеки (приклад): Існують також альтернативи, такі як **react-native-bluetooth-le** або **react-native-ble-manager**, кожна з яких має свої особливості. Однак **react-native-ble-plx** була обрана за її зрілість, високу продуктивність та активну спільноту, що є важливим для довгострокової підтримки проекту.

Ці бібліотеки спрощують складність роботи з нативними BLE API, дозволяючи розробникам зосередитися на бізнес-логіці застосунку, а не на низькорівневих аспектах Bluetooth-протоколу. (рис. 3.5)

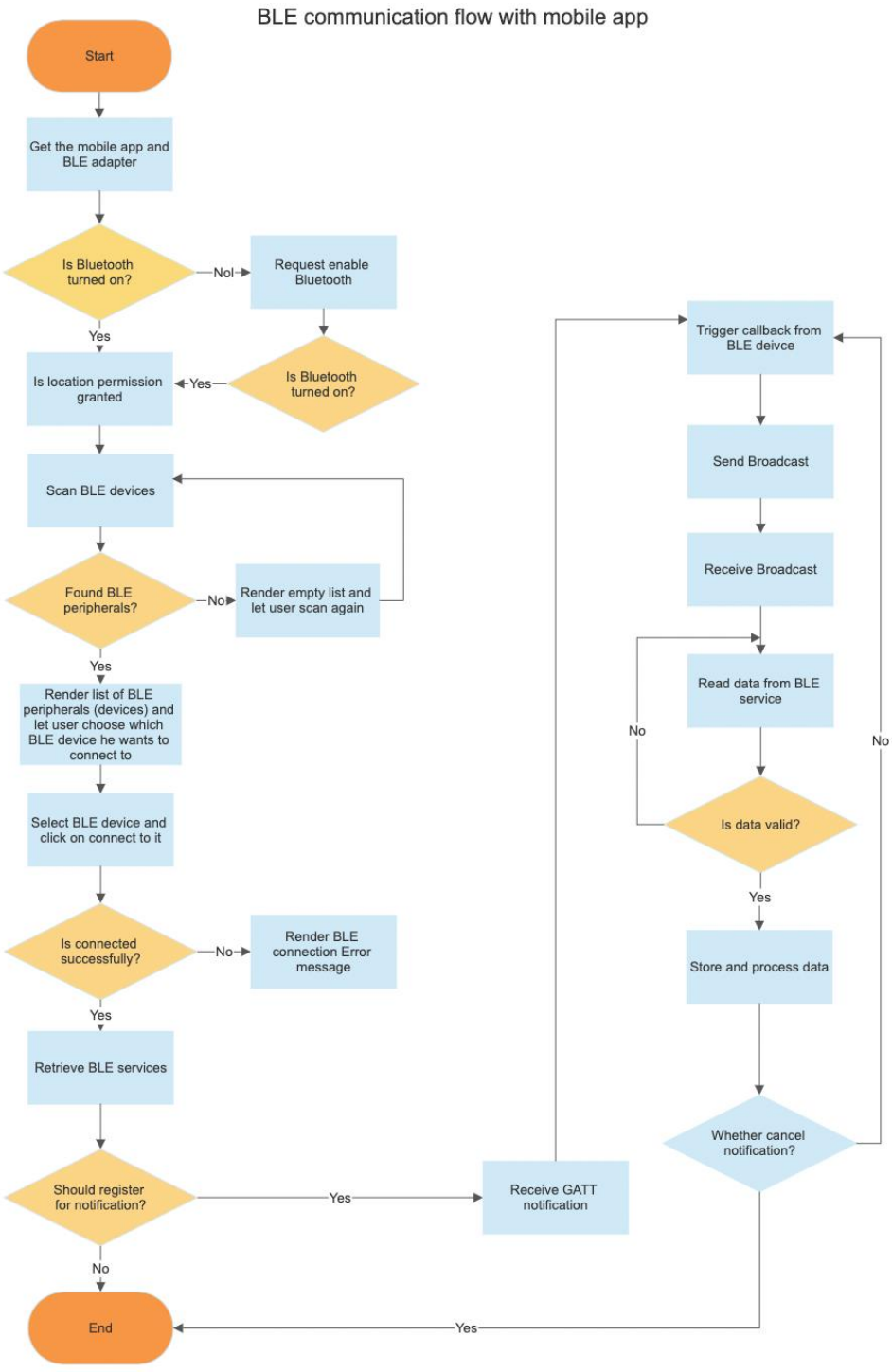


Рисунок 3.5 – Схема взаємодії React Native з BLE через бібліотеку react-native-ble-plx

3.3 Програмна реалізація мобільного застосунку

Програмна реалізація мобільного застосунку є кульмінаційним етапом розробки, де архітектурні рішення, вимоги та дизайнерські концепції перетворюються на функціонуючий програмний продукт. Цей етап охоплює створення користувацького інтерфейсу, інтеграцію з Bluetooth Low Energy (BLE) пристроями, налагодження навігації, управління станами та реалізацію функціоналу реєстрації/автентифікації користувачів. Для забезпечення крос-платформності та ефективності було використано фреймворк React Native [7].

3.3.1 Розробка UI-компонентів та основних сторінок

Розробка інтерфейсу розпочалася зі створення бібліотеки базових, атомарних компонентів, що дозволило забезпечити консистентність та швидкість верстки. Ці компоненти є будівельними блоками для всіх екранів застосунку:

Кнопки (<TouchableOpacity/>): Для реалізації інтерактивних кнопок використовувався базовий компонент React Native – TouchableOpacity. Він забезпечує візуальний відгук при натисканні (зменшення прозорості), що покращує користувацький досвід. Було створено кілька варіантів кнопок (основні, допоміжні, з іконками), уніфікованих за стилем (рис. 3.6).

src > components > views > button > AppButton.tsx > AppButton

```tsx
import React from 'react';
import { StyleProp, Text, TouchableOpacity, ViewStyle } from 'react-native';
import { useStyles } from './AppButtonStyles';
import { AppButtonType } from './AppButtonType';

export const AppButton = ({
    title,
    type = AppButtonType.Main,
    enabled = true,
    onPress = () => { },
    style = undefined,
}: {
    title?: string,
    type?: AppButtonType,
    enabled?: boolean;
    onPress?: () => void,
    style?: StyleProp<ViewStyle>,
}) => {
    const { styles } = useStyles(type || AppButtonType.Main);

    return (
        <TouchableOpacity
            style={[styles.container, enabled ? styles.enabled : styles.disabled, style]}
            onPress={onPress}
            disabled={!enabled}
            activeOpacity={0.7}
        >
            <Text style={styles.title}>{title}</Text>
        </TouchableOpacity>
    );
};
```

Рисунок 3.6 – Приклад реалізації компоненту кнопки

Текстові поля (<TextInput/>): Для введення даних (наприклад, ім'я користувача, назва пристрою, бажана температура) використовувалися стандартні компоненти TextInput, які були стилізовані відповідно до дизайн-системи. Забезпечено підтримку різних типів клавіатур та валідацію введених даних (рис. 3.7).

src > components > views > TextField > email > AppTextFieldEmail.tsx > ...

```tsx
import React from 'react';
import { ViewStyle } from 'react-native';
import { useStyles } from './AppTextFieldEmailStyles';
import { StyleProp } from 'react-native';
import { AppTextField } from '../Base/AppTextField';
import { LText, translate } from '../../../../locales/Translation';

export const AppTextFieldEmail = ({
    title = translate(LText.Email),
    placeholder = translate(LText.Email),
    editable = true,
    onTextChange = () => { },
    value = undefined,
    style = undefined,
}: {
    title?: string,
    placeholder?: string,
    editable?: boolean,
    onTextChange?: (text: string) => void,
    value?: string,
    style?: StyleProp<ViewStyle>,
}) => {
    const { styles } = useStyles();

    return (
        <AppTextField
            title={title}
            editable={editable}
            placeholder={placeholder}
            keyboardType='email-address'
            onTextChange={onTextChange}
            value={value}
            style={style}
        />
    );
};
```

Рисунок 3.7 – Приклад реалізації компоненту поля вводу

Модальні вікна (<Modal/>): Для відображення додаткової інформації, підтвердження дій або збору невеликих обсягів даних (наприклад, підтвердження виходу, діалоги налаштувань) застосовувалися вбудовані модальні компоненти React Native. Це дозволило створювати контекстні діалоги, не перевантажуючи основний екран.

Вибір зі списку (<Picker/>): Для вибору значень зі заздалегідь визначених списків (наприклад, тип пристрою, день тижня для розкладу) використовувався компонент Picker з бібліотеки @react-native-picker/picker. Це забезпечує зручний та стандартизований спосіб вибору опцій (рис. 3.8).

```
18  export const AppTextFieldPicker = ({
61
62      return (
63          <View style={styles.container}>
64              <View style={styles.titleContainer}>
65                  <Text style={styles.titleText}>{title}</Text>
66                  {infoText ? (
67                      <TouchableOpacity onPress={() => setInfoModalVisible(true)} style={styles.infoIcon}>
68                          <InfoIcon />
69                      </TouchableOpacity>
70                  ) : null}
71              </View>
72
73              <AppTextField
74                  editable={false}
75                  placeholder={placeholder}
76                  value={textInputValue}
77                  rightIcon={<PickerIcon />}
78                  style={style}
79                  onPress={() => setPickerVisible(true)}
80              />
81
82              <Modal visible={isPickerVisible} transparent={true} animationType="fade">
83                  <View style={styles.modalContainer} >
84                      <View style={styles.pickerContainer}>
85                          <Text style={styles.modalTitle}>
86                              {modalTitle}
87                          </Text>
88
89                          <BasePicker
90                              isWheelPicker={isWheelPicker}
91                              options={options}
92                              value={selectedValue}
93                              onSelectionChange={handlePickerSelect}
94                          />
95
96                          <View style={styles.controlButtonsContainer}>
97                              <AppButton type={AppButtonType.FilledLight} title={translate(LText.Cancel)} onPress={() =
98                              <AppButton title={translate(LText.Done)} onPress={handleDonePress} style={styles.buttonC
99                          </View>
100                     </View>
101                 </View>
102             </Modal>
103
104             <Modal visible={infoModalVisible} transparent={true} animationType="slide">
```

Рисунок 3.8 – Приклад реалізації компоненту селектора

Списки та відображення даних (<FlatList/>): Для ефективного відображення динамічних колекцій даних (наприклад, список приміщень, перелік пристроїв, записи історії мікроклімату) застосовувався компонент FlatList. Він оптимізований для роботи з великими обсягами даних,

забезпечуючи плавну прокрутку та високу продуктивність за рахунок рендерингу лише видимих елементів.

Мобільний застосунок складається з декількох ключових екранів, кожен з яких виконує свою специфічну функцію:

Головний екран (Dashboard): Є центральним хабом застосунку, надаючи користувачеві огляд поточного стану всіх підключених пристроїв та основних параметрів мікроклімату в кампусі. На цьому екрані відображаються картки пристроїв (температура, вологість, стан увімкнення/вимкнення), забезпечуючи швидкий доступ до їхнього керування (рис. 3.9).

src > screens > devices > view > DevicesScreen.tsx > ...

```tsx
import React from 'react';
import LoadingView from '../../../components/views/AppActivityLoadingView';
import { FlatList, View } from 'react-native';
import { DeviceCard } from '../../../components/views/deviceCard/DeviceCard';
import { useDevicesNavigator } from '../../../navigation/devices/DevicesNavigator';
import { useDevicesScreenModel } from '../viewModel/DevicesScreenModel';
import { useStyles } from './DevicesScreenStyles';

const DevicesScreen = () => {
  const navigator = useDevicesNavigator();
  const viewModel = useDevicesScreenModel();
  const { styles } = useStyles();

  // TODO: handle onStateButtonPress
  return (
    <View style={styles.container}>
      <FlatList
        data={viewModel.devices}
        renderItem={({ item }) => (
          <DeviceCard
            item={item}
            onItemPress={() => navigator.goToDevice(item)}
            onStateButtonPress={viewModel.toggleIsOn}
          />
        )}
        keyExtractor={item => item.deviceId}
        style={styles.flatList}
      />

      {viewModel.isLoading && (
        <LoadingView />
      )}
    </View>
  );
};

export default DevicesScreen;

```

Рисунок 3.9 – Скріншот реалізації головного екрану

Сторінка керування пристроєм: Детальний екран для кожного окремого пристрою, що дозволяє користувачеві переглядати його поточні показники, змінювати налаштування (наприклад, бажану температуру для термостата за допомогою слайдера), перемикати режими роботи та отримувати доступ до історії даних. Тут також реалізовано механізми керування сценаріями автоматизації для конкретного пристрою (рис. 3.10).

```
export const DeviceControlBaseView = (props: DeviceControlBaseViewProps) => {
  const styles = useStyles();
  const viewModel = props.viewModel;

  return (
    <View style={styles.container}>
      <ScheduleTabsBar
        onModeChange={(newMode) => viewModel.updateDataSource('controlType', newMode)}
        selectedTab={viewModel.dataSource.controlType ?? 'generic'}
      />

      {viewModel.dataSource.controlType === 'generic' &&
        <ScheduleGenericView
          settings={viewModel.dataSource.schedule}
          controlType={viewModel.dataSource.scheduleControlType}
          onControlTypeChange={(value) => viewModel.updateDataSource('scheduleControlType', value)}
          onScheduleChange={(value) => viewModel.updateDataSource('schedule', value)}
        />
      }

      {viewModel.dataSource.controlType === 'manual' &&
        <ScheduleManualView
          temperature={viewModel.dataSource.manualTemperature ?? 23}
          onTemperatureChange={(value) => viewModel.updateDataSource('manualTemperature', value)}
          timer={viewModel.dataSource.timer}
          onTimerChange={(value) => viewModel.updateDataSource('timer', value)}
        />
      }

      {viewModel.dataSource.controlType === 'antiFreeze' &&
        <ScheduleAntiFreezeView />
      }

      {viewModel.isSaveButtonVisible &&
        <View style={{ position: 'absolute', left: 20, bottom: 10, right: 20 }}>
          <AppButton
            title={translate(LText.Save)}
            onPress={props.onSubmitTap}
          />
        </View>
      }
    </View>
  );
}
```

Рисунок 3.10 – Скріншот сторінки керування пристроєм

Сторінка налаштувань: Цей екран надає можливість конфігурувати загальні параметри застосунку (перемикання між світлою та темною темами інтерфейсу, мовні налаштування, часовий пояс), а також управляти обліковим записом користувача.

Сторінки сценаріїв автоматизації: Окремий розділ, де користувачі можуть створювати, редагувати та активувати власні сценарії для автоматичного керування пристроями. Створювати сценарій можна для окремих пристроїв, та для кімнат.

Сторінки реєстрації та входу: Забезпечують безпечний доступ до системи, дозволяючи новим користувачам зареєструватися, а існуючим – авторизуватися. Реалізовано поля для введення облікових даних та кнопки для відправлення запитів до серверної частини (рис. 3.11).

Рисунок 3.11 – Скріншот сторінки керування пристроєм

Застосування компонентного підходу React Native дозволило створити гнучкий, масштабований та легкий у супроводі інтерфейс, що забезпечує високий рівень користувацького досвіду та ефективне управління мікрокліматом у кампусі.

3.3.2 Реалізація BLE-зв'язку та керування пристроями

Взаємодія мобільного застосунку з фізичними пристроями мікроклімату, що підтримують технологію Bluetooth Low Energy (BLE), є однією з найскладніших та найважливіших частин програмної реалізації. Цей

підрозділ деталізує алгоритм сканування, підключення, обміну даними та інтеграцію BLE-логіки з користувацьким інтерфейсом застосунку [8].

1. Алгоритм підключення до BLE-пристроїв:

Процес додавання нового пристрою до системи починається з ідентифікації та встановлення з'єднання з BLE-обладнанням. Це реалізується за допомогою компонента BLEDeviceScanner та бібліотеки `react-native-ble-plx.

1. 1 Ініціалізація та сканування: Після ініціалізації менеджера BLE (bleManager = new BleManager()), застосунок запускає сканування пристроїв. Для ідентифікації цільового пристрою використовується механізм QR-сканування. Компонент `QRScanner` (який використовує `react-native-vision-camera`) зчитує QR-код, що містить унікальні дані про пристрій, такі як `id`, `name`, `type` та `key`. Це дозволяє швидко знайти конкретний пристрій серед інших BLE-сигналів (рис. 3.12).

```tsx
const QRScanner: React.FC<QRScannerProps> = props => {
  const [hasPermission, setHasPermission] =
    useState<CameraPermissionStatus>('not-determined');
  const [torchEnabled, setTorchEnabled] = useState<boolean>(false);
  const device = useCameraDevice('back');
  const navigation = useNavigation();
  const styles = useStyles();
  const colors = useThemeColors();

  const codeScanner = useCodeScanner({
    codeTypes: ['qr'],
    onCodeScanned: codes => {
      const codeValue = codes[0]?.value || null;
      const codeFrame = codes[0]?.frame || null;

      Logger.logDebug(`QR Code Value: ${codeValue}`);

      if (codeFrame) {
        Logger.logDebug(`QR Code Frame:, ${codeFrame}`);
      }

      props.onRead(codeValue);
    },
  });

  useEffect(() => {
    const requestCameraPermission = async () => {
      const permission = await Camera.requestCameraPermission();
      setHasPermission(permission);
    };

    requestCameraPermission();

    const timeoutId = setTimeout(() => {
      props.onRead(null);
      Logger.logDebug(`${props.onRead}`);
    }, 20 * 1000);

    return () => clearTimeout(timeoutId);
  }, []);
```

Рисунок 3.12 – Скріншот компонента QRScanner

1.2 Встановлення безпечного підключення: Після виявлення цільового пристрою здійснюється спроба підключення до нього. Важливим кроком є перевірка поточного стану з'єднання (device.isConnected()) для уникнення повторного підключення. Після успішного з'єднання, відбувається виявлення всіх сервісів та характеристик пристрою (discoverAllServicesAndCharacteristics()). Це необхідно для подальшої взаємодії, оскільки дані передаються через характеристики, згруповані у сервіси (рис. 3.13).

```
const connectToDevice = device => {
  return device
    .isConnected()
    .then(connected => {
      if (!connected) {
        return device.connect();
      }
      return device;
    })
    .then(connectedDevice => {
      Logger.logDebug(`${objectName}: Connected to device: ${JSON.stringify(connectedDevice)}`);
      return connectedDevice.discoverAllServicesAndCharacteristics();
    });
};
```

Рисунок 3.13 – Скріншот функції під'єднання

1. 3 Авторизація пристрою: Для забезпечення безпеки, пристрій вимагає "ключ" для розблокування функціоналу. Цей ключ (scannedData.key), отриманий з QR-коду, записується у спеціальну BLE-характеристику (keyCharacteristicId). У разі невдалої спроби запису, реалізовано механізм повторних спроб (до 3 разів).

2. Механізми читання показників та надсилання команд:

Після успішного підключення та авторизації, застосунок може взаємодіяти з пристроєм для отримання даних та керування.

2.1 Отримання даних Wi-Fi мереж: Пристрій може надавати список доступних Wi-Fi мереж. Застосунок надсилає запит, а пристрій відповідає, надсилаючи дані через нотифікації BLE-характеристики

(notificationCharacteristicId). Мобільний застосунок підписується на ці нотифікації (monitorCharacteristicForService) для асинхронного отримання списку мереж. Отримані дані парсяться та відображаються у випадаючому списку (за допомогою `AppTextFieldPicker`) (рис. 3.14).

```
if (decodedValue.includes('wait')) {
  Logger.logDebug(`${objectName}: Value contains 'wait', subscribing to notifications on ${notificationCharacteristicId}`);
  return device.monitorCharacteristicForService(
    serviceId,
    notificationCharacteristicId,
    (error, characteristic) => {
      if (error) {
        Logger.logError(`${objectName}: Error subscribing to notifications: ${JSON.stringify(error)}`);
        return;
      }
      const notificationValue = characteristic?.value
        ? atob(characteristic.value)
        : '';
      Logger.logDebug(`${objectName}: Notification received: ${JSON.stringify(notificationValue)}`);

      const parsedNetworks = JSON.parse(notificationValue);
      const networkObjects = parsedNetworks.map(network => ({
        label: network[0],
        value: network[0],
      }));
      Logger.logDebug(`${objectName}: Parsed Networks: ${JSON.stringify(networkObjects)}`);

      setNetworks(networkObjects);
      setIsLoadingNetworks(false); // Закінчення завантаження
    },
  );
```

Рисунок 3.14 – Скріншот функції підписки на сповіщення

2.2 Надсилання даних для підключення до Wi-Fi: Користувач вибирає Wi-Fi мережу та вводить пароль (за допомогою `AppTextFieldPicker` та `AppTextFieldPassword`). Ці дані (SSID та пароль) формуються у JSON-об'єкт, кодуються у base64 та надсилаються до спеціальної характеристики (connectedCharacteristicId). Після запису, застосунок очікує відповіді від пристрою щодо статусу підключення до Wi-Fi. Це відбувається шляхом періодичного читання характеристики статусу з'єднання (characteristicForStatusConnect) до тих пір, поки не буде отримана відповідь "conn" (підключено), "check" (невірний пароль) або "wait" (помилка розробника) (рис. 3.15).

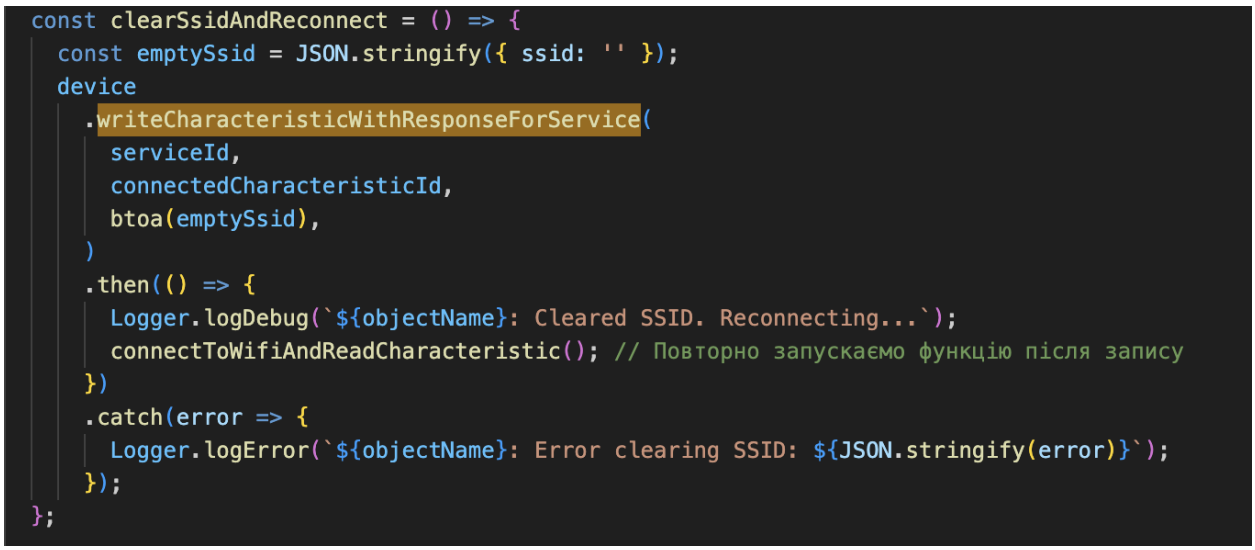


Рисунок 3.15 – Скріншот функції запису та читання статусу

2.3 Керування пристроями через BLE: Після успішного підключення пристрою до Wi-Fi, застосунок може надсилати команди для керування мікрокліматом (наприклад, увімкнення/вимкнення пристрою, встановлення температури) до відповідних характеристик. Логіка цих команд залежить від типу пристрою.

3. Інтеграція BLE-логіки з UI-компонентами застосунку:

Компонент BLEDeviceScanner інкапсулює всю логіку взаємодії з BLE, інтегруючись з базовими UI-компонентами.

3.1 Відображення стану: Залежно від стадії процесу (сканування QR-коду, пошук пристрою, підключення, очікування Wi-Fi мереж), користувачу відображаються відповідні елементи інтерфейсу та текстові повідомлення (isScanning, isConnected, isLoadingNetworks, connectedNetworkMessage).

3.2 Введення даних: Використання стандартизованих компонентів вводу (текстові поля, випадаючі списки) забезпечує зручність та консистентність введення інформації про Wi-Fi мережі та налаштування пристрою.

3.3 Обробка подій: Кнопка "Додати" (handleAddButtonPress) ініціює запис Wi-Fi даних до пристрою та подальшу логіку його реєстрації в системі.

Кнопка "Reconnect to new Wifi" (`clearSsidAndReconnect`) дозволяє скинути налаштування Wi-Fi та повторно підключитися.

Управління життєвим циклом BLE: Функція stopBleProcesses забезпечує коректне завершення сканування, відписку від нотифікацій та закриття BLE-з'єднання після успішного додавання пристрою або у випадку помилки, що є критично важливим для оптимізації енергоспоживання та уникнення витоків пам'яті.

Реалізація BLE-зв'язку та керування пристроями є центральною частиною функціоналу застосунку, забезпечуючи надійну та ефективну взаємодію з апаратним забезпеченням системи мікроклімату (дивитись Додаток 1).

3.3.3 Організація навігації та керування станами

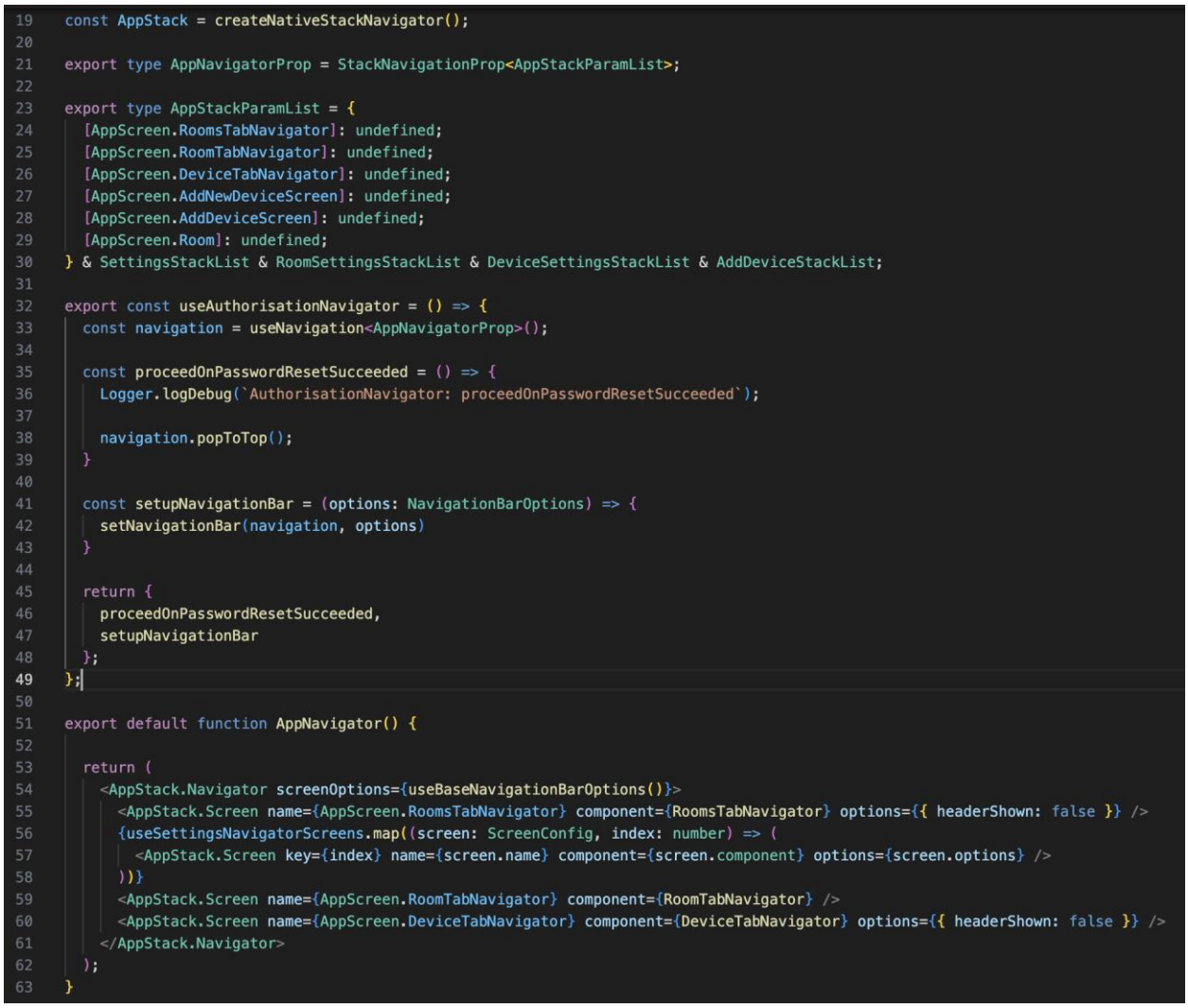
Для забезпечення плавного переходу між різними екранами застосунку та підтримки консистентності даних на всіх рівнях, критично важливою є правильно спроектована система навігації та ефективне керування станами. У даному проекті ці аспекти реалізовано з використанням бібліотеки React Navigation для навігації та комбінації локальних станів React з потенційним використанням централізованого сховища даних для глобального стану (наприклад, Redux або React Context API).

1. Система навігації між екранами (React Navigation):

React Navigation є стандартним рішенням для організації навігації у React Native застосунках. Вона дозволяє створювати складні навігаційні структури, такі як стекові навігатори, навігатори з вкладками (tabs) та Drawer-навігатори (висувні бічні меню).

Навігатори стекового типу (<createNativeStackNavigator/>): Для більшості переходів між екранами використовується стек-навігатор. Кожен новий екран "накладається" на попередній, створюючи стек, що дозволяє

легко повертатися до попереднього стану. Це реалізовано в `AppNavigator`, який є кореневим стеком застосунку (рис. 3.16).



```
const AppStack = createNativeStackNavigator();

export type AppNavigatorProp = StackNavigationProp<AppStackParamList>;

export type AppStackParamList = {
  [AppScreen.RoomsTabNavigator]: undefined;
  [AppScreen.RoomTabNavigator]: undefined;
  [AppScreen.DeviceTabNavigator]: undefined;
  [AppScreen.AddNewDeviceScreen]: undefined;
  [AppScreen.AddDeviceScreen]: undefined;
  [AppScreen.Room]: undefined;
} & SettingsStackList & RoomSettingsStackList & DeviceSettingsStackList & AddDeviceStackList;

export const useAuthorisationNavigator = () => {
  const navigation = useNavigation<AppNavigatorProp>();

  const proceedOnPasswordResetSucceeded = () => {
    Logger.logDebug(`AuthorisationNavigator: proceedOnPasswordResetSucceeded`);

    navigation.popToTop();
  }

  const setupNavigationBar = (options: NavigationBarOptions) => {
    setNavigationBar(navigation, options)
  }

  return {
    proceedOnPasswordResetSucceeded,
    setupNavigationBar
  };
};

export default function AppNavigator() {

  return (
    <AppStack.Navigator screenOptions={useBaseNavigationBarOptions()}>
      <AppStack.Screen name={AppScreen.RoomsTabNavigator} component={RoomsTabNavigator} options={{ headerShown: false }} />
      {useSettingsNavigatorScreens.map((screen: ScreenConfig, index: number) => (
        <AppStack.Screen key={index} name={screen.name} component={screen.component} options={screen.options} />
      ))}
      <AppStack.Screen name={AppScreen.RoomTabNavigator} component={RoomTabNavigator} />
      <AppStack.Screen name={AppScreen.DeviceTabNavigator} component={DeviceTabNavigator} options={{ headerShown: false }} />
    </AppStack.Navigator>
  );
}
```

Рисунок 3.16 – Приклад коду AppNavigator

Вкладкові навігатори (Tab Navigators): Для організації основних розділів застосунку використовуються вкладкові навігатори. Це забезпечує швидкий доступ до ключових функцій застосунку без глибокого переходу по стеку. RoomsTabNavigator та DeviceTabNavigator є прикладами такої реалізації.

Вкладені навігатори: Складніші структури (наприклад, налаштування пристрою або кімнати) реалізовані за допомогою вкладених стек-навігаторів (DeviceSettingsStackList, RoomSettingsStackList). Це дозволяє створювати автономні навігаційні потоки всередині більшого застосунку, забезпечуючи модульність та керованість.

Управління навігацією: Застосунок використовує хук useNavigation() з React Navigation для доступу до об'єкта навігації. Це дозволяє програмно здійснювати переходи між екранами (navigation.push(), navigation.pop(), navigation.goBack(), navigation.popToTop()), а також налаштовувати верхню панель навігації (setupNavigationBar).

2. Керування станами застосунку:

Керування станами даних є фундаментальним для підтримки їхньої консистентності, особливо в асинхронних операціях (BLE-зв'язок, взаємодія з API) та при оновленні UI.

Локальні стани компонентів (useState): Для управління станом окремих компонентів (scannedData, device, isScanning, isConnected у BLEDeviceScanner) використовується хук useState з React. Це дозволяє ефективно керувати даними, які безпосередньо впливають на рендеринг конкретного компонента.

Глобальні стани та їх поширення: Для даних, які повинні бути доступні в різних частинах застосунку (наприклад, інформація про підключені пристрої, дані користувача, налаштування застосунку), використовується централізований підхід. Хоча в наданих фрагментах коду прямих прикладів Redux чи Context API не видно, їх застосування є типовим для таких систем. Це дозволяє уникнути "прокидання пропсів" (prop drilling) та забезпечити легкий доступ до глобальних даних з будь-якого компонента.

Context API: Дозволяє створювати контекст для даних, які повинні бути доступні в дереві компонентів без явного передавання пропсів на кожному рівні. Це підходить для менших глобальних станів або для ієрархічних даних.

Redux : Для великих та складних застосунків Redux пропонує передбачуваний контейнер стану, що полегшує налагодження, тестування та управління складними взаємодіями.

Ефекти життєвого циклу (useEffect): Хук useEffect використовується для виконання побічних ефектів у компонентах, таких як запити до API, підписки на події BLE, управління таймерами. Він дозволяє синхронізувати компонент з зовнішніми системами та очищати ресурси при відключенні компонента.

Логування: Для відстеження поведінки застосунку та налагодження проблем активно використовується модуль `Logger.tsx`, який записує інформацію про події та стани застосунку.

Загалом, інтегрована система навігації та керування станами забезпечує плавний користувацький досвід та підтримує ефективну розробку, дозволяючи обробляти складні асинхронні операції та підтримувати актуальність даних у всьому застосунку (рис. 3.17).

```
type SettingsNavigationType = StackNavigationProp<RoomSettingsStackList>;
const SettingsStack = createNativeStackNavigator();

export type RoomSettingsStackList = {
  [AppScreen.Settings]: undefined;
  [AppScreen.RoomSettings]: undefined;
};
export const useSettingsNavigator = () => {
  const navigation = useNavigation<SettingsNavigationType>();
  function parentAsAppNavigator() {
    const parenAppNavigator = navigation.getParent()?.getParent<AppNavigatorProp>();
    if (parenAppNavigator) {
      return parenAppNavigator
    } else {
      Logger.logError("SettingsNavigator: Failed to fetch parent navigator as AppNavigator")
      return null
    }
  }

  const goToPasswordChange = () => {
    Logger.logDebug(`SettingsNavigator: go to goToPasswordChange`);
    parentAsAppNavigator()?.push(AppScreen.PasswordChange)
  }

  const procceedOnPasswordChangeSucceeded = () => {
    Logger.logDebug(`SettingsNavigator: go to rooms`);
    parentAsAppNavigator()?.pop();
  };

  const setupNavigationBar = (options: NavigationBarOptions) => {
    setNavigationBar(navigation, options)
  }
  return {
    goToPasswordChange,
    procceedOnPasswordChangeSucceeded,
    setupNavigationBar
  };
};

export const useRoomSettingsNavigatorScreens: ScreenConfig[] = [
  { name: AppScreen.Settings, component: SettingsScreen },
  { name: AppScreen.RoomSettings, component: RoomSettingsScreen },
];
```

Рисунок 3.17 – Приклад коду RoomSettingsNavigator

3.3.4 Реалізація функціоналу реєстрації та входу користувачі

Функціонал реєстрації та входу користувачів є критично важливим елементом будь-якої системи, що вимагає персоналізованого доступу та управління даними. У системі управління мікрокліматом у кампусі, це забезпечує, що лише авторизовані користувачі можуть переглядати дані приміщень, керувати пристроями та змінювати налаштування. Реалізація

цього функціоналу охоплює взаємодію мобільного застосунку із серверною частиною для перевірки облікових даних та управління сесіями [9].

1. Процес реєстрації нових користувачів:

Процес реєстрації дозволяє новим користувачам створити обліковий запис у системі.

Збір даних: Користувач вводить необхідні дані, такі як ім'я, адреса електронної пошти та пароль, через відповідні текстові поля в інтерфейсі мобільного застосунку.

Валідація на стороні клієнта: Перед відправкою даних на сервер, застосунок виконує первинну валідацію введених даних (наприклад, формат електронної пошти, складність пароля, перевірка на порожні поля). Це покращує користувацький досвід та зменшує навантаження на сервер.

Відправка запиту на сервер: Після успішної клієнтської валідації, застосунок відправляє POST-запит до серверної API з обліковими даними нового користувача. Для цього використовується створений модуль ApiService (зокрема метод `post`). Запити направляються до відповідного endpoint реєстрації на API_BASE_URL.

Обробка відповіді сервера: Сервер обробляє запит, хешує пароль, зберігає нового користувача в базі даних та надсилає відповідь застосунку (успіх/невдача, повідомлення про помилку). У разі успішної реєстрації, користувачу може бути запропоновано увійти в систему.

2. Процес входу користувачів у систему:

Процес входу дозволяє зареєстрованим користувачам отримати доступ до функціоналу застосунку.

Збір облікових даних: Користувач вводить свої електронну пошту та пароль.

Відправка запиту на автентифікацію: Застосунок надсилає POST-запит до серверної API (endpoint входу) з введеними обліковими даними.

3. Механізми автентифікації та авторизації:

Amazon Cognito (або подібний сервіс): Як було зазначено в архітектурі (Розділ 2.1.3), Amazon Cognito використовується для управління ідентифікацією. Після отримання запиту на вхід, серверна частина (або безпосередньо мобільний застосунок, якщо це дозволено архітектурою Cognito) взаємодіє з Cognito для перевірки облікових даних [4, 6].

JWT-токени (JSON Web Tokens): У разі успішної автентифікації, Cognito або сервер генерує та повертає JWT-токен (або пару токенів – access token та refresh token). Цей токен є цифровим підписом, що підтверджує особу користувача та містить інформацію про його права доступу.

Зберігання токену: Отриманий токен зберігається на стороні клієнта (наприклад, у SecureStore або Keychain для iOS та Android) для безпечного та постійного доступу без необхідності повторного входу.

ApiService для авторизованих запитів: Модуль ApiService реалізує функціонал перехоплювачів запитів (`interceptors.request.use`). Це означає, що перед кожним HTTP-запитом до сервера, якщо токен існує, він автоматично додається до заголовку `Authorization` у форматі `Bearer {token}`. Це забезпечує авторизацію всіх подальших запитів до захищених ресурсів API.

Обробка відповідей та помилок: ApiService також має перехоплювачі відповідей (`interceptors.response.use`), які централізовано обробляють помилки HTTP, зокрема статус `401 Unauthorized`. Це дозволяє автоматично перенаправляти користувача на екран входу або оновлювати токен (якщо передбачено механізм Refresh Token) у випадку закінчення терміну дії або недійсності токена.

4. Інтеграція з інтерфейсом застосунку:

Екрани входу/реєстрації: В UI передбачені окремі екрани для реєстрації та входу з відповідними полями вводу та кнопками, що ініціюють запити до API.

Обробка успіху/невдачі: Після успішного входу користувач перенаправляється на головний екран застосунку. У разі невдачі (неправильний логін/пароль, помилка сервера), відображається відповідне повідомлення користувачу.

Реалізація цього функціоналу є основою для забезпечення безпеки, персоналізації та доступу до всіх можливостей системи управління мікрокліматом.

3.3.5 Інтеграція з серверною частиною

Інтеграція мобільного застосунку із серверною частиною є критично важливою для забезпечення централізованого управління, зберігання історичних даних, синхронізації налаштувань та реалізації складних функціональних можливостей, які не можуть бути повністю оброблені на клієнтській стороні. У даній системі управління мікрокліматом, серверна частина виступає як хаб, що агрегує дані від різних пристроїв та надає єдину точку доступу до них для мобільних клієнтів [10].

Взаємодія між мобільним застосунком та серверною API відбувається за допомогою HTTP/HTTPS-запитів, використовуючи створений модуль `ApiService` (як було описано у розділі 3.3.4), що забезпечує стандартизований та безпечний обмін даними. Враховуючи архітектуру системи (Розділ 2.1.3), серверна частина може включати такі компоненти, як AWS IoT для пристроїв Інтернету речей та базу даних для постійного зберігання інформації.

1. Механізми синхронізації даних:

Синхронізація даних є фундаментальною для забезпечення актуальності інформації, що відображається в мобільному застосунку, та консистентності станів пристроїв.

Первинна синхронізація при вході: При вході користувача в систему, мобільний застосунок надсилає запит до серверної API для отримання повного списку приміщень, пристроїв, їхніх поточних станів та налаштувань, які асоційовані з цим користувачем. Це забезпечує відображення актуальної інформації відразу після авторизації.

Фонова синхронізація (Push/Pull): Для підтримки актуальності даних у реальному часі може використовуватись комбінація механізмів:

Polling (періодичні запити): Застосунок може періодично надсилати запити до сервера (наприклад, кожні кілька секунд або хвилин) для отримання оновлень станів пристроїв або нових даних мікроклімату. Цей метод є простим у реалізації, але може створювати зайве навантаження на сервер.

WebSockets (push-нотифікації): Для більш ефективної синхронізації у реальному часі може бути використано WebSockets. Це дозволяє серверу "надсилати" оновлення даних до мобільного застосунку, як тільки вони з'являються, без необхідності постійних запитів з боку клієнта. Це особливо актуально для відстеження показників датчиків та оперативного реагування на команди управління. AWS IoT підтримує протокол MQTT, який ефективно працює з WebSockets, дозволяючи інтегрувати push-повідомлення про зміни стану пристроїв.

2. Отримання історичних показників:

База даних на сервері (`ClimateData`, `device_weekly_stats`, `room_weekly_stats` згідно Розділу 2.2) є основним сховищем історичних даних.

Запити до API: Мобільний застосунок надсилає GET-запити до серверної API, вказуючи необхідні параметри (наприклад, `DeviceID`, `RoomID`, часовий діапазон) (рис. 3.18).

```
const getDeviceHistory = async (deviceId, startDate, endDate) => {
  try {
    const result = await api.get(`/devices/${deviceId}/history`, {
      params: { startDate, endDate }
    });
    if (result.success) {
      console.log('Device history:', result.data);
      return result.data;
    } else {
      console.error('Failed to fetch device history:', result.message);
      return null;
    }
  } catch (error) {
    console.error('Network error fetching device history:', error);
    return null;
  }
};
```

Рисунок 3.18 – Приклад функції на отримання статистики

Агрегація та фільтрація на сервері: Серверна частина може виконувати агрегацію (наприклад, розрахунок середніх значень за день/тиждень/місяць) та фільтрацію даних перед їх надсиланням до клієнта, щоб зменшити обсяг переданої інформації та оптимізувати продуктивність застосунку.

Візуалізація: Отримані історичні дані відображаються в мобільному застосунку у вигляді графіків або таблиць, дозволяючи користувачам аналізувати динаміку зміни мікроклімату та споживання ресурсів.

3. Зберігання налаштувань та автоматизації сценаріїв:

Налаштування пристроїв: Будь-які зміни в налаштуваннях пристроїв (наприклад, бажана температура, режим роботи, параметри гістерезису), що встановлюються через мобільний застосунок, надсилаються до серверної API (POST/PUT-запити) та зберігаються в базі даних (`DeviceSettings`). Це

забезпечує постійність налаштувань, навіть якщо мобільний пристрій вимкнено або замінено.

Сценарії автоматизації: Створення, редагування та активація користувацьких сценаріїв (Розділ 2.2.1, сутність `AutomationScenarios`) також відбувається через взаємодію з серверною API. Серверна частина відповідає за виконання цих сценаріїв, моніторячи показники датчиків та надсилаючи відповідні команди до пристроїв. Це забезпечує централізоване та надійне виконання автоматизації.

4. Взаємодія з AWS IoT:

У рамках хмарної інфраструктури, AWS IoT відіграє роль брокера повідомлень [5].

Мобільний застосунок, підключений до інтернету, може взаємодіяти з AWS IoT для відправки команд на пристрої (якщо пристрої підключені до Wi-Fi і можуть отримувати команди від AWS IoT) та для отримання сповіщень про події з пристроїв.

AWS IoT може інтегруватися з іншими сервісами AWS (наприклад, Lambda для обробки даних, S3 для зберігання логів, DynamoDB/RDS для бази даних), що надає потужні можливості для масштабування, аналізу даних та розширення функціоналу системи управління мікрокліматом.

Інтеграція із серверною частиною є фундаментальною для забезпечення повноцінного та стабільного функціонування системи управління мікрокліматом, дозволяючи мобільному застосунку виступати як ефективний інструмент контролю та моніторингу, що спирається на потужні хмарні обчислення та сховища даних.

У рамках третього розділу було детально розглянуто програмне та технічне забезпечення розробки мобільного застосунку для управління мікрокліматом у кампусі. Обґрунтовано вибір ключових інструментів та

технологій, які забезпечили ефективність, крос-платформність та надійність системи.

Основними засобами розробки виступили Visual Studio Code як гнучке та потужне інтегроване середовище, React Native як крос-платформна мобільна платформа, що дозволяє створювати нативні застосунки для iOS та Android з єдиної кодової бази, а також Figma для інтуїтивного та ефективного проектування користувацького інтерфейсу та досвіду.

Ключовим аспектом реалізації стала інтеграція з Bluetooth Low Energy (BLE). Детально описано принципи роботи BLE, включаючи GATT-профіль, сервіси та характеристики, що є основою для обміну даними між мобільним застосунком та фізичними пристроями. Вибір бібліотеки `react-native-ble-plx` обґрунтовано її можливостями для сканування, підключення та ефективного обміну даними з BLE-пристроями, що критично важливо для моніторингу та керування мікрокліматом.

Програмна реалізація мобільного застосунку охоплює розробку стандартизованих UI-компонентів та основних екранів, таких як "Dashboard", екрани керування пристроями та налаштувань, які забезпечують інтуїтивно зрозумілу взаємодію. Деталізовано алгоритм BLE-зв'язку, починаючи від сканування та безпечного підключення, до читання показань датчиків та надсилання команд керування актуаторам, включаючи логіку підключення пристроїв до Wi-Fi. Ефективна організація навігації за допомогою React Navigation та управління станами за допомогою локальних станів (з потенціалом розширення до глобальних сховищ) забезпечують плавність роботи застосунку та консистентність даних. Реалізований функціонал реєстрації та входу користувачів, а також інтеграція із серверною частиною, що використовує `ApiService` та Amazon Cognito, гарантує безпечний та централізований доступ до системи та її даних.

Таким чином, у даному розділі було продемонстровано комплексний підхід до програмної та технічної реалізації системи управління мікрокліматом, який поєднує сучасні мобільні технології, ефективні протоколи бездротового зв'язку та принципи безпеки, закладаючи міцну основу для функціонування розробленого застосунку.

ВИСНОВКИ

У дипломній роботі було розроблено мобільний застосунок, призначений для моніторингу та управління мікрокліматом у приміщеннях університетського кампусу. Система забезпечує інтеграцію з фізичними пристроями через протокол Bluetooth Low Energy (BLE) та централізовану обробку даних на серверній частині, спрямовану на підвищення комфорту та енергоефективності.

Проведено ґрунтовний аналіз літературних джерел та стану проблемної області, що дозволило виявити актуальність задачі управління мікрокліматом у великих освітніх установах. Проаналізовано існуючі рішення та технології, включаючи традиційні BMS-системи та сучасні мобільні застосунки для "розумного будинку", що дозволило обґрунтувати переваги обраного підходу та виокремити ключові вимоги до розроблюваної системи.

У рамках математичного моделювання було визначено загальну трирівневу архітектуру системи, що включає рівні пристроїв, мобільного застосунку та серверної частини. Детально спроектовано структуру бази даних для зберігання інформації про користувачів, приміщення, пристрої, а також історичних даних мікроклімату та налаштувань. Розроблено та описано алгоритми фільтрації даних з сенсорів для підвищення їх достовірності, а також математичну модель застосування гістерезису для ефективного управління актуаторами, що дозволяє оптимізувати енергоспоживання та забезпечити стабільний мікроклімат.

Програмне забезпечення мобільного застосунку реалізовано на крос-платформному фреймворку React Native, що забезпечило можливість розгортання як на Android, так і на iOS платформах. Дизайн користувацького інтерфейсу розроблено в Figma, що гарантувало його інтуїтивність та естетичну привабливість. Детально описано реалізацію базових UI-компонентів, структуру основних екранів, а також механізми навігації та

керування станами застосунку. Ключовим елементом реалізації є модуль взаємодії з BLE-пристроями, що охоплює сканування, підключення, обмін даними та надсилання команд. Інтеграція із серверною частиною, що використовує AWS IoT та Amazon Cognito для автентифікації, забезпечує централізоване зберігання даних та доступ до розширеного функціоналу.

В результаті розробки створено працездатний прототип мобільного застосунку, який підтвердив свою ефективність у забезпеченні зручного та централізованого контролю над мікрокліматом. Реалізовані алгоритми та механізми дозволяють підвищити комфорт перебування користувачів у приміщеннях та сприяють оптимізації енергоспоживання кампусу.

Практична цінність розробленої системи полягає в можливості її використання як інструменту для моніторингу та управління мікрокліматом у реальних умовах університетського кампусу, а також як основи для подальшого розширення функціоналу, інтеграції з іншими системами "розумного кампусу" та проведення наукових досліджень у цій галузі.

Таким чином, мету дипломної роботи досягнуто, усі поставлені задачі виконано, а запропоноване рішення є важливим кроком у напрямку автоматизації управління інфраструктурою освітніх закладів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація React Native. [Електронний ресурс] – Режим доступу до ресурсу: https://reactnative.dev/docs/getting-started (дата звернення: 20.03.2025).
2. Офіційна документація Bluetooth Core Specification [Електронний ресурс] – Режим доступу до ресурсу:

   https://www.bluetooth.com/specifications/specs/ (дата звернення: 28.03.2025).
3. Офіційна документація Figma Help Center. [Електронний ресурс] – Режим доступу до ресурсу: https://help.figma.com/ (дата звернення: 06.04.2025).
4. Polidea. react-native-ble-plx: A powerful and comprehensive Bluetooth Low Energy library for React Native. [Електронний ресурс] – Режим доступу до ресурсу: https://polidea.github.io/react-native-ble-plx/ (дата звернення: 03.03.2025).
5. Amazon Web Services. AWS IoT Core Documentation. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/iot/latest/developerguide/iot-dg.pdf (дата звернення: 21.04.2025).
6. Amazon Web Services. Amazon Cognito Developer Guide. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-dg.pdf (дата звернення: 21.04.2025).
7. Sami H. H. Learning React Native: Build native mobile apps with JavaScript and React. Packt Publishing, 2021. – 500 p.

8. Shih J. H. Practical IoT Physical Design: A Practical Approach to Designing Connected Devices. Packt Publishing, 2020. – 350 p.
9. Adam C. G. Full-Stack React, TypeScript, and AWS: Build Modern Web Applications with React, TypeScript, GraphQL, and AWS. O'Reilly Media, 2023. – 700 p.
10. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015. – 300 p.

ДОДАТКИ

ДОДАТОК А

BLEDeviceScanner.jsx

```
import React, { useState,
 useEffect } from 'react';
import {
  Text,
  View,
  ActivityIndicator,
  Alert,
} from 'react-native';
import { BleManager } from 'react-native-ble-plx';
import QRScanner from './QRScanner';
import base64 from 'react-native-base64';
import { AppTextFieldPicker } from '../views/TextField/Picker/AppTextFieldPicker';
import { AppTextField } from '../views/TextField/Base/AppTextField';
import { AppTextFieldPassword } from '../views/TextField/Password/AppTextFieldPassword';
import { AppButton } from '../views/button/AppButton';
import { useAddDeviceNavigator } from '../../navigation/general/AddDeviceNavigation';
import { useStyles } from './BLEDeviceScannerStyles';
import { LText, translate } from '../../locales/Translation';
import Logger from '../../utils/Logger';
import DeviceManager from '../../managers/DeviceManager';

const bleManager = new BleManager();

const BLEDeviceScanner = () => {
  const objectName = "BLEDeviceScanner";
  const navigator = useAddDeviceNavigator();
  const styles = useStyles();
  const [scannedData, setScannedData] = useState(null);
  const [device, setDevice] = useState(null);
  const [isScanning, setIsScanning] = useState(false);
  const [isConnected, setIsConnected] = useState(false);
```

```
const [writeAttempt, setWriteAttempt] = useState(0);
const [notificationMessage, setNotificationMessage] = useState('');
const [networks, setNetworks] = useState([]);
const [selectedNetwork, setSelectedNetwork] = useState('');

const allRooms = DeviceManager.devicesGroups().map(room => ({
  label: room.id,
  value: room .name
}));

const [rooms, setRooms] = useState(allRooms);
const [selectedRoom, setSelectedRoom] = useState('no-room');
const [password, setPassword] = useState('');
const [deviceName, setDeviceName] = useState('');
const [isLoadingNetworks, setIsLoadingNetworks] = useState(false);
const [connectedNetworkMessage, setConnectedNetworkMessage] = useState(null);
const serviceId = '7a6f10e0-dc05-11ec-9d64-0242ac120002';
const characteristicForStatusConnect = '5e427456-fd7c-4956-bde8-1c275a7a1d9b';
const notificationCharacteristicId = '7fe6343d-c796-45b9-bcc8-d4efdd9868f6';
const connectedCharacteristicId = '5c3a659e-897e-45e1-b016-007107c96df7';
const keyCharacteristicId = '2df9da8c-47c4-4e0c-99f2-d90dfd5a4bc3';

useEffect(() => {
  if (scannedData && scannedData.name) {
    setIsScanning(true);
    bleManager.startDeviceScan(null, null, (error, foundDevice) => {
      if (error) {
        Logger.logError(`${objectName}: Error during scan: ${JSON.stringify(error)}`);
        setIsScanning(false);
        return;
      }
      if (foundDevice && foundDevice.name === scannedData.name) {
        setDevice(foundDevice);
        Logger.logDebug(`${objectName}: scanned device: ${JSON.stringify(foundDevice)}`);

        bleManager.stopDeviceScan();
        setIsScanning(false);
```

```
      }
    });
  }
  return () => bleManager.stopDeviceScan();
}, [scannedData]);

useEffect(() => {
  if (device) {
    connectToDevice(device)
      .then(connectedDevice => {
        Logger.logDebug(`${objectName}: Device connected.`);
        setIsConnected(true);
        return connectedDevice.services();
      })
      .then(services => {
        Logger.logDebug(`${objectName}: Available services:`);
        services.forEach(service => {
          Logger.logDebug(`Service UUID: ${service.uuid}`);
          getCharacteristicsForService(service.uuid);
        });
        connectToWifiAndReadCharacteristic();
      })
      .catch(error => {
        Logger.logError(`${objectName} Error while connecting or retrieving services: ${JSON.stringify(error)}`);
      });
  }
}, [device]);

const connectToDevice = device => {
  return device
    .isConnected()
    .then(connected => {
      if (!connected) {
        return device.connect();
      }
      return device;
    })
```

```
    .then(connectedDevice => {
      Logger.logDebug(`${objectName}: Connected to device: ${JSON.stringify(connectedDevice)}`);
      return connectedDevice.discoverAllServicesAndCharacteristics();
    });
};

const getCharacteristicsForService = serviceId => {
  device
    .characteristicsForService(serviceId)
    .then(charList => {
      Logger.logDebug(`${objectName}: Characteristics for service ${serviceId}:`);
      charList.forEach(characteristic => {
        Logger.logDebug(`${objectName}: Characteristic UUID: ${characteristic.uuid}`);

        const isWritable =
          characteristic.isWritableWithResponse ||
          characteristic.isWritableWithoutResponse;

        if (isWritable) {
          Logger.logDebug(`${objectName}: This characteristic supports writing.`);
        } else {
          Logger.logError(`${objectName}: characteristic does not support writing.`);
        }

        if (
          serviceId === serviceId &&
          characteristic.uuid === keyCharacteristicId &&
          isWritable
        ) {
          writeKeyToCharacteristic(characteristic);
        }
      });
    })
    .catch(error => Logger.logError(`${objectName}: Error retrieving characteristics: ${JSON.stringify(error)}`));
};

const writeKeyToCharacteristic = characteristic => {
```

```
  if (scannedData && scannedData.key) {
    const dataToWrite = JSON.stringify({ key: scannedData.key });
    const encodedData = base64.encode(dataToWrite);

    Logger.logDebug(`${objectName}: Data to be written to characteristic \n(base64 encoded): ${encodedData}; \nwrite data ${dataToWrite}`);

    characteristic
      .writeWithResponse(encodedData)
      .then(() => {
        Logger.logDebug(`${objectName}: Data successfully written to characteristic.`);
        readValueFromCharacteristic(characteristic);
        setWriteAttempt(0);
      })
      .catch(error => {
        Logger.logError(`${objectName}: Error writing data to characteristic: ${JSON.stringify(error)}`);
        if (writeAttempt < 3) {
          setWriteAttempt(writeAttempt + 1);
          Logger.logDebug(`${objectName}: to write data... Attempt ${writeAttempt + 1}`);
          setTimeout(() => writeKeyToCharacteristic(characteristic), 2000);
        }
      });
  }
};

const readValueFromCharacteristic = characteristic => {
  characteristic
    .read()
    .then(value => {
      Logger.logDebug(`${objectName}: Raw value from characteristic: ${value}`);

      if (value instanceof ArrayBuffer) {
        const decodedBase64 = new TextDecoder().decode(new Uint8Array(value));
        Logger.logDebug(`${objectName}: Decoded base64 string: ${decodedBase64}`);

        try {
          const jsonValue = JSON.parse(decodedBase64);
```

```
          if (jsonValue.res) {
            Logger.logDebug(`${objectName}: The value is true.`);
          } else {
            Logger.logDebug(`${objectName}: The value is not true.`);
          }
        } catch (error) {
          Logger.logError(`${objectName}: Error parsing JSON: ${JSON.stringify(error)}`);
        }
      } else if (typeof value === 'string') {
        Logger.logDebug(`${objectName}: Received value as string: ${value}`);

        try {
          const jsonValue = JSON.parse(value);
          if (jsonValue.res) {
            Logger.logDebug(`${objectName}: The value is true.`);
          } else {
            Logger.logDebug(`${objectName}: The value is not true.`);
          }
        } catch (error) {
          Logger.logError(`${objectName}: Error parsing JSON from string: ${JSON.stringify(error)}`);
        }
      } else {
        Logger.logDebug(`${objectName}: Received value is not an ArrayBuffer or string: ${JSON.stringify(value)}`);
      }
    })
    .catch(error => {
      Logger.logError(`${objectName}: Error reading characteristic value: ${JSON.stringify(error)}`);
    });
};

const handleAddButtonPress = async () => {
  Logger.logDebug(`${objectName}: Selected Network: ${JSON.stringify(selectedNetwork)}`);
  Logger.logDebug(`${objectName}: Password: ${password}`);

  if (!selectedNetwork) {
    Alert.alert('Choosing a network ', ' Please select a Wi-Fi network.');
```

```
    return;
  }

  if (!password) {
    Alert.alert('Password ', ' Please enter the password. ');
    return;
  }

  const writeCharacteristicUUID = connectedCharacteristicId;
  const networkData = JSON.stringify({
    ssid: selectedNetwork,
    pass: password,
  });

  Logger.logDebug(`${objectName}: Data to write in characteristic: ${networkData}`);

  try {
    await device.writeCharacteristicWithResponseForService(
      serviceId,
      writeCharacteristicUUID,
      base64.encode(networkData),
    );

    await new Promise(resolve => setTimeout(resolve, 1000));
    let decodedValue = '';

    for (let attempt = 0; attempt < 5; attempt++) {
      const characteristicValue = await device.readCharacteristicForService(
        serviceId,
        characteristicForStatusConnect,
      );

      decodedValue = characteristicValue.value
        ? atob(characteristicValue.value)
        : '';
      Logger.logDebug(`${objectName}: Read value from characteristics (attempted' + (attempt + 1) + '): ${decodedValue}`);
```

```
  if (decodedValue.includes('conn')) {
    break;
  }

  await new Promise(resolve => setTimeout(resolve, 1000));
}

if (decodedValue.includes('conn')) {
  let groupId = undefined;
  if (selectedRoom) {
    groupId = rooms.find(option => option.value === selectedRoom)?.label || undefined;
  }

  const deviceId = scannedData.name;
  const deviceKey = scannedData.key;

  if (!deviceId) {
    Logger.logError(`${objectName}: device id not found`);
    throw new Error('Device id not found')
  }

  if (!deviceKey) {
    Logger.logError(`${objectName}: device key not found`);
    throw new Error('Device key not found')
  }

  await DeviceManager.addDevice(deviceId, deviceKey, deviceName, groupId);

  Logger.logDebug(`${objectName}: The connection is successful.`);
  Alert.alert("Connection", "Successful connection!", [
    {
      text: 'OK',
    },
  ]);
  stopBleProcesses();
} else if (decodedValue.includes('check')) {
  Logger.logError(`${objectName}: Error: Check password`);
```

```
      Alert.alert('Error ', ' Incorrect data, try again.');
    } else if (decodedValue.includes('wait')) {
      Logger.logError(`${objectName}: Error: Developer`);
      Alert.alert('Error ', "Can't connect, try again.");
    }
  } catch (error) {
    Logger.logError(`${objectName}: Error when recording or reading data: ${JSON.stringify(error)}`);
    Alert.alert(
      'Error',
      'Failed to read or record data to the device.',
    );
  }
};

const stopBleProcesses = () => {
  if (device.isScanning) {
    device.stopScan();
    Logger.logDebug(`${objectName}: Ble scan stopped`);
  }

  if (device.notificationSubscription) {
    device.notificationSubscription.remove();
    Logger.logDebug(`${objectName}: Written from Ble notifications`);
  }

  device.cancelConnection();
  Logger.logDebug(`${objectName}: Ble connection is closed`);
};

const connectToWifiAndReadCharacteristic = () => {
 setIsLoadingNetworks(true);
  device
    .requestMTU(500)
    .then(() => {
      Logger.logDebug(`${objectName}:MTU set to 500`);
      return device.characteristicsForService(serviceId);
    })
```

```
    .then(charList => {
      const characteristic = charList.find(
        char => char.uuid === characteristicForStatusConnect,
      );
      if (characteristic) {
        Logger.logDebug(`${objectName}: Found characteristic ${characteristicForStatusConnect}, reading value...`);
        return characteristic.read();
      } else {
        Logger.logDebug(`${objectName}: Characteristic ${characteristicForStatusConnect} not found.`);
      }
    })
    .then(value => {
      const decodedValue = value.value ? atob(value.value) : '';
      Logger.logDebug(`${objectName}: Value from characteristic after Wi-Fi connection:' ${JSON.stringify(decodedValue)}`);


      if (decodedValue.includes('wait')) {
        Logger.logDebug(`${objectName}: Value contains 'wait', subscribing to notifications on ${notificationCharacteristicId}`);
        return device.monitorCharacteristicForService(
          serviceId,
          notificationCharacteristicId,
          (error, characteristic) => {
            if (error) {
              Logger.logError(`${objectName}: Error subscribing to notifications: ${JSON.stringify(error)}`);
              return;
            }
            const notificationValue = characteristic?.value
              ? atob(characteristic.value)
              : '';
            Logger.logDebug(`${objectName}: Notification received: ${JSON.stringify(notificationValue)}`);


            const parsedNetworks = JSON.parse(notificationValue);
            const networkObjects = parsedNetworks.map(network => ({
              label: network[0],
              value: network[0],
            }));
```

```
          Logger.logDebug(`${objectName}: Parsed Networks: ${JSON.stringify(networkObjects)}`);

          setNetworks(networkObjects);
          setIsLoadingNetworks(false);
        },
      );
    } else if (decodedValue.includes('conn')) {
      Logger.logDebug(`${objectName}: Value contains 'conn', reading connected characteristic ${connectedCharacteristicId}...`);
      return device
        .readCharacteristicForService(serviceId, connectedCharacteristicId)
        .then(connectedValue => {
          const connectedData = connectedValue.value
            ? atob(connectedValue.value)
            : '';
          Logger.logDebug(`${objectName}: Device is already connected to the network: ${connectedData}`);

          const parsedConnectedData = JSON.parse(connectedData);
          setConnectedNetworkMessage(
            `Device is already connected to the network ${parsedConnectedData.ssid}.`,
          );
          setIsLoadingNetworks(false);
        });
    } else {
      Logger.logDebug(`${objectName}: No need to subscribe or read, 'wait' or 'conn' not found in value.`);
      setIsLoadingNetworks(false);
    }
  })
  .catch(error => {
    Logger.logError(`${objectName}: Error connecting to Wi-Fi or reading characteristic: ${JSON.stringify(error)}`);
    setIsLoadingNetworks(false);
  });
};

const clearSsidAndReconnect = () => {
  const emptySsid = JSON.stringify({ ssid: '' });
```

```
  device
    .writeCharacteristicWithResponseForService(
      serviceId,
      connectedCharacteristicId,
      btoa(emptySsid),
    )
    .then(() => {
      Logger.logDebug(`${objectName}: Cleared SSID. Reconnecting...`);
      connectToWifiAndReadCharacteristic();
    })
    .catch(error => {
      Logger.logError(`${objectName}: Error clearing SSID: ${JSON.stringify(error)}`);
    });
};

return (
  <View style={styles.container}>
    {!scannedData ? (
      <QRScanner
        onRead={data => {
          if (data) {
            Logger.logInfo(`${objectName}: QRScanner did read data - ${JSON.stringify(data)}`);
            const parsedData = JSON.parse(data);
            setScannedData(parsedData);
          }
        }}
      />
    ) : (
      <View style={styles.infoContainer}>
        {isScanning ? (

          <Text>
            {translate(LText.BLEScanningDevice)} {scannedData.name}...
          </Text>
        ) : isConnected ? (
          <>
            <Text style={styles.connectedText}>
```

```
  {translate(LText.BLEConnectionSucceeded)}
</Text>

<AppTextFieldPicker
  style={styles.field}
  title="Select Room"
  placeholder="Без кімнати"
  value={selectedRoom}
  onSelectionChange={(value) => {
    setSelectedRoom(value);
  }}
  options={rooms}
/>

<AppTextField
  style={styles.field}
  title="Device Name"
  placeholder="Enter device name"
  value={deviceName}
  onTextChange={setDeviceName}
/>

{isLoadingNetworks ? (
  <ActivityIndicator size="large" color="white" />
) : networks.length > 0 ? (
  <>
    <AppTextFieldPicker
      style={styles.field}
      title="Select Wi-Fi Network"
      placeholder="Choose a network"
      value={selectedNetwork}
      onSelectionChange={setSelectedNetwork}
      options={networks}
    />

    <AppTextFieldPassword
      style={styles.field}
```

```
                title="Wi-Fi Password"
                placeholder="Enter password"
                onTextChange={setPassword}
              />
              {/* Добавити if */}
              <Text style={styles.textTip}>
                Якщо мережа відкрита, залиште поле Пароль пустим
              </Text>
              <AppButton title="Додати" onPress={handleAddButtonPress} />
            </>
          ) : null}

          {connectedNetworkMessage && (
            <View style={styles.connectedNetworkContainer}>
              <Text style={styles.connectedNetworkText}>
                {connectedNetworkMessage}
              </Text>
              <AppButton
                title="Reconnect to new Wifi"
                onPress={clearSsidAndReconnect}
                style={styles.reconnectButton}
              />
            </View>
          )}
        </>
      ) : (
        <Text>Device found, attempting to connect...</Text>
      )}
    </View>
  )}
 </View>
 );
};

export default BLEDeviceScanner;
```