Міністерство освіти і науки України

Чернівецький національний університет

імені Юрія Федьковича

Навчально-науковий інститут фізико-технічних та комп'ютерних наук
(повна назва інституту/факультету)

Кафедра інформаційних технологій та комп'ютерної фізики
(повна назва кафедри)

Інформаційна система для «розумного» будинку «SweeMe» (Серверна частина)

Кваліфікаційна робота

Рівень вищої освіти - перший (бакалаврський)

Виконав:
студент *4* курсу, групи *417ск*
спеціальності
126 Інформаційні системи та технології
(назва спеціальності)

Любінецький Владислав Ігорович
(прізвище та ініціали)
Керівник *д. фіз.-мат. н, доц. Борча М. Д.*
(науковий ступінь, вчене звання, прізвище та ініціали)

До захисту допущено:

Протокол засідання кафедри № 20

від „15" червня 2023 р.

зав. кафедри доц. Борча М. Д.

Чернівці – 2023

РЕЦЕНЗІЯ НА БАКАЛАВРСЬКУ РОБОТУ

Інформаційна система для «розумного» будинку «SweeMe». Серверна частина

Студента Любінецького Владислава Ігоровича

Інститут фізико-технічних та комп'ютерних наук

Спеціальність інформаційні системи та технології

Чернівецького національного університету імені Юрія Федьковича

Актуальність теми. Розробка інформаційної системи «розумного» будинку передбачає, насамперед, зручне і ефективне використання пристроїв, забезпечення економії ресурсів, автоматизацію рутинної роботи, створення більш безпечного середовища. Тому тема роботи і реалізація серверної частини «розумного» будинку є актуальними.

Практичне значення. Впровадження інформаційної системи та API для взаємодії з нею через мобільний додаток в системі розумного будинку забезпечує зручний та простий спосіб керування всіма функціями розумного будинку. Як наслідок – відбувається оптимізація енергоспоживання та забезпечення більш ефективного управління ресурсами.

Структура та зміст роботи. Робота складається з 3х розділів. У першому розділі неведені теоретичні відомості про інформаційну систему, описано принцип взаємодії всіх елементів з «розумним» будинком. Наведено обгрунтування вибору системи керування базами даних, мови програмування, серверного середовища. У другому розділі визначено набір технологій, які найкраще відповідають вимогам та потребам проекту. Описано функції, структура системи та її робота з прикладом передачі та відображення даних у базу даних. У третьому розділі описана реалізація інформаційної системи та її взаємодія з API. Продемонстрована робота інформаційної системи та її можливості.

Зауваження. Було б добре додати можливість сповіщення клієнта у критичних чи аварійних випадках роботи пристроїв системи.

Оцінка. В результаті виконання бакалаврської роботи студент Любінецький В.І. здійснив розробку API для взаємодії з «розумним» будинком в інформаційній системі "SweeMe". Запропонована інформаційна система та API є гнучкою та має потенціал для модифікацій та покращень. Вважаю, що завдання бакалаврської роботи виконано повністю, а Любінецький В.І. заслуговує оцінки „відмінно" та присвоєння кваліфікації бакалавра зі спеціальності 126 – інформаційні системи та технології.

Рецензент
Доцент кафедри комп'ютерних систем та мереж
Чернівецького національного університету
Кандидат технічних наук
Яковлева Інна Дмитрівна

"14" 06 2023 р.

UNICHECK
by Turnitin

Ім'я користувача:
Кафедра інформаційних технологій та комп фізики

ID перевірки:
1015666821

Дата перевірки:
21.06.2023 13:07:52 EEST

Тип перевірки:
Doc vs Library + My Database

Дата звіту:
21.06.2023 13:09:16 EEST

ID користувача:
36471

Назва документа: Любінецький

Кількість сторінок: 13 Кількість слів: 5977 Кількість символів: 50112 Розмір файлу: 79.75 KB ID файлу: 1015297

9.37%
Схожість

Найбільша схожість: 8.42% з джерелом з Бібліотеки (ID файлу: 1015311305)

Не знайдено джерел з Інтернету

9.37% Джерела з Бібліотеки 2 ..... Сторінка 15

0% Цитат

Не знайдено жодних цитат

Вилучення списку бібліографічних посилань вимкнене

2.81%
Вилучень

Деякі джерела вилучено автоматично (фільтри вилучення: кількість знайдених слів є меншою за 12 слів та 2%)

0.44% Вилучення з Інтернету 23 ..... Сторінка 15

2.68% Вилученого тексту з Бібліотеки 190 ..... Сторінка 15

Міністерство освіти і науки України

Чернівецький національний університет

імені Юрія Федьковича

___Навчально-науковий інститут фізико-технічних та комп'ютерних наук___
(повна назва інституту/факультету)

_____Кафедра інформаційних технологій та комп'ютерної фізики_____
(повна назва кафедри)

Інформаційна система для «розумного» будинку «SweeMe». Серверна частина

Кваліфікаційна робота

Рівень вищої освіти - перший (бакалаврський)

Виконав:
студент *4* курсу, групи *417ск*
спеціальності
126 Інформаційні системи та технології
(назва спеціальності)

Любінецький Владислав Ігорович
(прізвище та ініціали)
Керівник *д. фіз.-мат. н, доц. Борча М. Д.*
(науковий ступінь, вчене звання, прізвище та ініціали)

До захисту допущено:

Протокол засідання кафедри № ____

від „____" _____ 2023 р.

зав. кафедри _____доц. Борча М. Д.

Чернівці – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Навчально-науковий інститут фізико-технічних та комп'ютерних наук
Кафедра комп'ютерних систем та мереж

ЗАТВЕРДЖУЮ
Завідувач кафедри
док. фіз.-мат. наук, доц.
_____М. Д. Борча
”___” _____ 2023 р.

Інформаційна система для «розумного» будинку «SweeMe». Серверна частина

ЛИСТ ЗАТВЕРДЖЕННЯ

УЗГОДЖЕНО
Керівник роботи
докт. фіз.-мат. наук, доцент
_____ М. Д. Борча
“___” _____ 2023 р.

Виконавець
студент 4-го курсу
_____ В. І. Любінецький
“___” _____ 2023 р.

2023

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Любінецькому Владиславу Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна система для «розумного» будинку «SweeMe». Серверна частина*
керівник роботи *Борча Мар'яна Драгошівна,* докт. фіз.-мат. наук, ст.наук.сп.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від "__"_____2023 року №_____

2. Строк подання студентом проекту (роботи) *2023 р.*

3. Вихідні дані до проекту (роботи) Мета роботи – розробити інформаційну систему «розумного» дому "SweeMe" та специфічної частини цієї системи – розробка API. Система повинна реєструвати та авторизовувати користувача, зчитувати інформацію з пристрою, мати можливість додати пристрій до акаунту користувача, зберігати дані про користувачів, пристрої, показники температури та вологості у відповідних таблицях MySQL. Програмне забезпечення розробити на мові JavaScript з використанням серверного середовища Node.js

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

1) дослідити існуючі інформаційні системи-аналоги
2) описати принцип роботи реєстрації, авторизації, додавання пристрою, зчитування та отримування даних з пристрою
3) розробити базу даних для збереження усіх необхідних даних в таблицях
4) розробити API з усім необхідним функціоналом
5) дослідити ефективність роботи розробленої програми

5. Перелік графічного матеріалу

1) Структура таблиць бази даних

Студент _____ *Любінецький В. І.*
(підпис) (прізвище та ініціали)

Керівник проекту (роботи) _____ *Борча М. Д.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота виконана студентом групи 417ск Любінецьким Владиславом Ігоровичем. Тема «Інформаційна система для «розумного» будинку «SweeMe» (Серверна частина)». Робота направлена на здобуття ступеня бакалавр за спеціальністю 126 «Інформаційні системи та технології».

Метою кваліфікаційної роботи є розробка інформаційної системи «розумного» дому "SweeMe" та специфічної частини цієї системи - розробка API. Програмна реалізація системи виконана на мові JavaScript із використанням серверного середовища Node.js.

Бакалаврська робота містить: кількість сторінок – 67, рисунків – 29, додатків – 1, використаних джерел – 13.

Ключові слова: інформаційна система, API, база даних, SQL, JavaScript, Node.js, токен.

Робота містить результати власних досліджень. Використання чужих ідей, результатів і текстів мають посилання на відповідне джерело.

ABSTRACT

The qualification work was carried out by Vladyslav Ihorovych Liubinetskyi, a student of group 417sk. The topic is Information system for the "smart" home "SweeMe" (Server part). The work is aimed at obtaining a bachelor's degree in specialty 126 "Information systems and technologies".

The goal of the qualification work is the development of the information system of the "smart" house "SweeMe" and the specific part of this system - the development of the API. The software implementation of the system is made in the JavaScript language using the Node.js server environment.

The bachelor's work consists of 67 pages, 29 figures, 1 appendix, and references to 13 sources.

Keywords: information system, API, database, SQL, JavaScript, Node.js, token.

The work contains the results of own research. The use of other people's ideas, results and texts are linked to the appropriate source.

ЗМІСТ

ВСТУП

Метою даної кваліфікаційної роботи є розробка інформаційної системи «розумного» дому "SweeMe" та специфічної частини цієї системи – розробка API.

Мета роботи є актуальною тому, що зростає популярність інтелектуальних систем у побутовій сфері та потребі в надійному та ефективному способі керування розумним будинком. Також у зв’язку із воєнним станом та дорожчанням електроенергії постає питання економії. Розумний будинок є хорошим рішенням, адже допомагатиме заощаджувати електроенергію, дозволить мати повний контроль над пристроями та даватиме можливість керувати ними віддалено. Самі пристрої можуть бути розроблені конкретно під вимоги клієнта, а також окремо налаштувати їх взаємодію та параметри на вимогу користувача.

Задачі дослідження включають аналіз предметної області, проектування інформаційної системи та вибір інструментарію та методів для реалізації цієї системи. Аналіз предметної області включатиме вивчення сучасних технологій та рішень у галузі «розумних» будинків, а також виявлення основних потреб і вимог користувачів. На основі цього аналізу буде здійснено проектування інформаційної системи "SweeMe", що включатиме в себе розробку API для взаємодії з «розумним» будинком.

Завдання дослідження включають:

- Аналіз предметної області, дослідження потреб користувачів розумного будинку та визначення необхідного функціоналу для API.
- Проектування інформаційної системи, включаючи створення концептуальної, логічної та фізичної моделей бази даних для збереження інформації про користувачів та пристроїв.

- Вибір та використання необхідних інструментів та методів для реалізації API.
- Розробка функціоналу для реєстрації та авторизації користувачів, управління пристроями, та збереження даних у базі даних.

Програмна реалізація системи виконана мовою Java Script за допомогою серверного середовища Node.js.

У результаті виконання бакалаврської роботи розроблено інформаційну систему та API для взаємодії з нею, яка призначена для системи «розумного» будинку.

Об'єктом дослідження є інформаційна система «розумного» дому "SweeMe".

Предметом дослідження є технології проектування та розробки інформаційних систем, методики аналізу та оцінки інформаційних систем, а також використання популярних технологій Node.js, MySQL для розробки API.

РОЗДІЛ 1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Аналіз об’єкту дослідження та предметної області

1.1.1 Теоретичні відомості про об’єкт дослідження та предметну область

Інформаційна система – це сукупність взаємопов'язаних компонентів, які працюють разом для збору, зберігання, обробки, передачі та використання інформації з метою досягнення конкретних цілей [1]. Інформаційні системи використовуються в різних галузях, включаючи бізнес, науку, та освіту**.**

Основні компоненти інформаційної системи включають:

- Вхідні дані
- Зберігання даних
- Обробка даних
- Вихідні дані
- Користувачі
- Програмне забезпечення
- Апаратне забезпечення

Вхідні дані є одним з основних компонентів інформаційної системи і включають інформацію, яка вводиться або надходить в систему з різних джерел. Ці дані можуть бути отримані від людей, сенсорів або інших систем і можуть мати різні формати та структуру. Дані можуть бути:

- Структуровані – це дані, які організовані в певній структурі або форматі, що дозволяє легко обробляти та аналізувати їх. Наприклад, дані у вигляді таблиць баз даних або XML-файлів є структурованими даними [2].
- Неструктуровані дані – це дані, що не мають чіткої організованої структури. Наприклад, текстові документи, електронні листи, зображення та відеофайли є прикладами неструктурованих даних. Обробка таких даних може бути складнішою, оскільки вони не мають стандартизованого формату [3].

Важливим аспектом вхідних даних є їхнє якість, достовірність та цілісність. Для забезпечення цих аспектів інформаційна система може використовувати різні методи та механізми перевірки та валідації даних, такі як перевірка формату, контроль цілісності, перевірка прав доступу.

Зберігання даних є одним з важливих компонентів інформаційної системи і включає механізми, які забезпечують ефективне та безпечне зберігання інформації. Основні види зберігання даних включають:

- База даних – є центральним механізмом для зберігання даних у структурованому форматі. Бази даних дозволяють організувати дані у відповідних таблицях, колекціях або графах та забезпечувати доступ до них за допомогою мов запитів [4].
- Файлові системи – використовуються для зберігання файлів та документів. Вони надають механізми для організації, категоризації та управління файлами та папками. Файлові системи можуть використовуватись для зберігання неструктурованих даних, таких як текстові документи, зображення, відео та інші мультимедійні файли.
- Хмарне зберігання – є альтернативою традиційним локальним методам зберігання даних. Дані зберігаються на серверах, що розташовані у віддалених дата-центрах, і до них можна отримати доступ через Інтернет. Хмарне зберігання забезпечує гнучкість, масштабованість та можливість резервного копіювання даних.

Ці механізми забезпечують ефективне та надійне зберігання даних, забезпечуючи доступ до них, збереження цілісності та захист від втрати або несанкціонованого доступу

Обробка даних включає ряд процесів, що допомагають здобути корисну інформацію з вхідних даних. Ці процеси включають фільтрацію, сортування, аналіз, обчислення та витягування інформації. Вони дозволяють валідувати, очищувати та перетворювати дані, забезпечуючи їх якість і готовність до використання. Обробка даних грає важливу роль у виконанні різних функцій

інформаційних систем, таких як прийняття рішень, аналіз трендів, прогнозування та автоматизація бізнес-процесів.

Вихідні дані є результатом обробки інформаційною системою і містять корисну інформацію, яка передається користувачеві або іншим системам. Ці дані можуть бути представлені у різних форматах, таких як текстові файли, звіти, графіки, таблиці, діаграми тощо. Вихідні дані готуються з метою надання користувачеві або іншій системі необхідної інформації для подальшого використання, прийняття рішень, сприйняття тенденцій або здійснення дій відповідно до потреб і цілей, для яких була створена інформаційна система.

Користувачі інформаційної системи включають людей та інші системи, які взаємодіють з нею. Люди можуть бути операторами, адміністраторами, аналітиками, менеджерами, клієнтами або іншими ролями, які використовують систему для введення даних, отримання розрахунків, доступу до інформації та прийняття рішень [4]. Інші системи можуть взаємодіяти з інформаційною системою через інтерфейси програмного забезпечення, щоб автоматично передавати дані, отримувати результати обробки або виконувати інші завдання. Наприклад, інтеграція з системою управління виробництвом або електронною комерцією дозволяє автоматично оновлювати запаси, відстежувати замовлення або обмінюватися даними з іншими бізнес-системами. Користувачі використовують інформаційну систему для виконання своїх завдань, забезпечення ефективної комунікації, спільної роботи, отримання потрібної інформації та підтримки прийняття рішень. Інтерфейси користувача, такі як веб-додатки, мобільні додатки або графічні інтерфейси, допомагають забезпечити зручний доступ до функцій системи та спрощують взаємодію з користувачем.

Програмне забезпечення включає широкий спектр програм, що забезпечують контроль роботи інформаційної системи. Операційні системи є основою для виконання і керування різноманітними програмами та ресурсами системи. Бази даних забезпечують зберігання, організацію та керування

доступом до даних, дозволяючи ефективно зберігати великі обсяги інформації [4]. Додатково, існують програми для обробки даних, включаючи програми аналізу даних, статистичні пакети, інструменти для машинного навчання та штучного інтелекту, що допомагають відшукати тенденції, залежності та іншу корисну інформацію з вхідних даних. Веб-додатки дозволяють взаємодіяти з інформаційною системою через інтернет, надаючи зручний спосіб доступу та обміну інформацією за допомогою веб-браузера. Інші програми можуть включати спеціалізовані додатки для конкретних функцій, такі як системи управління відносинами з клієнтами (CRM), системи управління проектами, фінансові програми.

Апаратне забезпечення – фізичні компоненти, такі як комп'ютери, сервери, мережеве обладнання, які забезпечують функціонування інформаційної системи [4]. Воно включає комп'ютери для виконання програм, сервери для зберігання та обробки даних, а також мережеве обладнання для забезпечення зв'язку та обміну даними. Апаратне забезпечення визначає продуктивність та надійність системи.

API (Application Programming Interface) – це набір правил і протоколів, які визначають, як різні програми або компоненти програм можуть взаємодіяти між собою. API визначає, які функції та операції доступні для використання, як передавати дані між програмами та як обробляти результати [5].

Воно може бути використано для спрощення розробки програмного забезпечення, створення розширень для інших програм, інтеграції різних систем, а також для доступу до функціональності веб-сервісів. Його перевагою є те, що воно дозволяє програмістам використовувати готові компоненти або сервіси без необхідності знати деталі їх реалізації.

API може бути представленим у різних форматах, таких як REST (Representational State Transfer), SOAP (Simple Object Access Protocol), JSON-RPC (JSON Remote Procedure Call) [6]. В даній бакалаврській роботі використовується REST API.

Крім того, існують API для різних сфер, включаючи соціальні мережі, фінансові сервіси, картографію, медіа та багато інших. Його використання дозволяє програмам взаємодіяти між собою, обмінюватися даними та виконувати різні функції безпосередньо. Це спрощує розробку програмного забезпечення, розширює можливості програм та дозволяє створювати інтегровані рішення.

1.1.2 Принцип взаємодії інформаційної системи та API з «розумним» будинком

В основі роботи «розумного» будинку стоїть API, яка базується на використанні HTTP запитів і відповідей. Додаток взаємодіє з API, надсилаючи HTTP запити на сервер і отримуючи відповіді з необхідною інформацією.

Для спрощення взаємодії та забезпечення стандартизації, API використовує архітектурний стиль REST (Representational State Transfer). За допомогою REST, різні операції над даними (створення, читання, оновлення, видалення) виконуються за допомогою відповідних HTTP методів, таких як GET, POST, PUT і DELETE [7].

Додаток використовує URL-адреси (ендпоінти) API для доступу до різних ресурсів і операцій. Наприклад, для отримання списку пристроїв з бази даних, додаток може виконати GET запит до URL-адреси «/devices». Для створення нового пристрою, він може відправити POST запит до URL-адреси «/devices» з даними в тілі запиту про новий пристрій.

Для передачі даних між додатком і API використовується формат даних, такий як JSON (JavaScript Object Notation). Додаток може надсилати та отримувати дані у вигляді JSON-об'єктів [8].

Взаємодія з API включає такі операції, як реєстрація та авторизація користувача, отримання і оновлення даних про пристрої, отримання стану датчиків, керування пристроями. Додаток виконує відповідні HTTP запити до

відповідних URL-адрес, а API обробляє ці запити, виконує необхідні операції з базою даних та повертає відповіді з необхідною інформацією.

1.2. Постановка задачі

1.2.1 Вимоги до системи

Розроблена інформаційна система складається з бази даних та API для налагодження взаємодії із «розумним» будинком.

До бази даних ставляться такі вимоги:

- Побудувати фізичну модель бази даних, яка буде зберігати дані про користувача після його реєстрації, налаштування користувача (колірна схема в мобільному додатку), список пристроїв з інформацією про тип та серійний номер, дані з датчиків пристроїв.
- Виконувати програмну реалізацію за допомогою системи керування базами даних MySQL засобами Node.js.

До API ставляться такі вимоги:

- Налаштувати локальний сервер, який забезпечуватиме з'єднання з базою даних та взаємодію з нею за допомогою HTTP запитів.
- Реалізувати реєстрацію, авторизацію та автентифікацію користувача, використовуючи технології токенів та сесій для забезпечення безпеки, створити рівні доступу для користувачів.
- Створити моделі для бази даних, включаючи взаємозв'язки між таблицями, міграції для керування схемою бази даних, та контролери для обробки запитів.
- Реалізувати CRUD (Create, Read, Update, Delete) маршрутизацію для кожної таблиці бази даних, що дозволить виконувати операції з даними, такі як створення, читання, оновлення та видалення. Описати ендпоінти.

Програмну реалізацію API виконати на мові JavaScript використовуючи серверне середовище Node.js.

1.2.2 Очікування від впровадження системи

Впровадження інформаційної системи та API для взаємодії з нею через мобільний додаток в систему розумного будинку забезпечує зручний та простий спосіб керування всіма функціями розумного будинку через мобільний пристрій. Користувачі можуть легко управляти пристроями використовуючи зрозумілий інтерфейс мобільного додатку. Це робить життя мешканців більш комфортним та зручним. Також використання API для взаємодії з інформаційною системою відкриває додаткові можливості для розширення функціональності розумного будинку. Розробники можуть створювати власні додатки або інтегрувати систему розумного будинку з іншими системами, наприклад, з системою розумного міста чи розумної енергетики. Це дозволяє створювати інноваційні рішення та досліджувати нові способи оптимізації енергоспоживання та забезпечення більш ефективного управління ресурсами.

Висновки до розділу 1

У даному розділі було неведені теоретичні відомості про інформаційну систему, описано принцип взаємодії всіх елементів з «розумним» будинком. Також було виконано постановку задачі з описом вимог до системи та очікуваннями від її впровадження.

Після проведення аналізу ринку компаній, що надають послуги по розробці систем «розумного» будинку, було прийнято рішення про розробку власної інформаційної системи. Це рішення було обумовлено фінансовими міркуваннями, оскільки вартість комерційних варіантів систем розумного будинку часто є значною. Розробка власної системи дозволить значно знизити витрати на реалізацію цього проекту.

Хоча власна система може не мати такої самої високої якості, як комерційні варіанти, вона має потенціал для подальшого розвитку. Власна система дозволить гнучко адаптуватися до потреб користувачів. Крім того, це

надасть можливість розширювати функціональність системи та розробляти власні пристрої, що є важливим перевагою в умовах швидкого технологічного розвитку.

Для розробки інформаційної системи було обрано систему керування базами даних MySQL. Це розповсюджена та надійна система, яка забезпечує ефективне зберігання та обробку даних. Використання MySQL дозволить забезпечити стабільну роботу бази даних та оптимальну продуктивність системи.

Мова програмування JavaScript та серверне середовище Node.js були обрані для розробки API. JavaScript є широко використовуваною мовою програмування з великим екосистемою інструментів та бібліотек. Node.js забезпечує високу продуктивність та масштабованість додатків, а також дає змогу використовувати JavaScript як мову програмування як на front-end, так і на back-end. Це дозволяє забезпечити єдність мови та зручну розробку всієї інформаційної системи.

РОЗДІЛ 2 ПОБУДОВА СИСТЕМИ

2.1 Моделювання системи

2.1.1 Функції та структура системи

Структура інформаційної системи складається із декількох таблиць в базі даних (рис. 2.1).

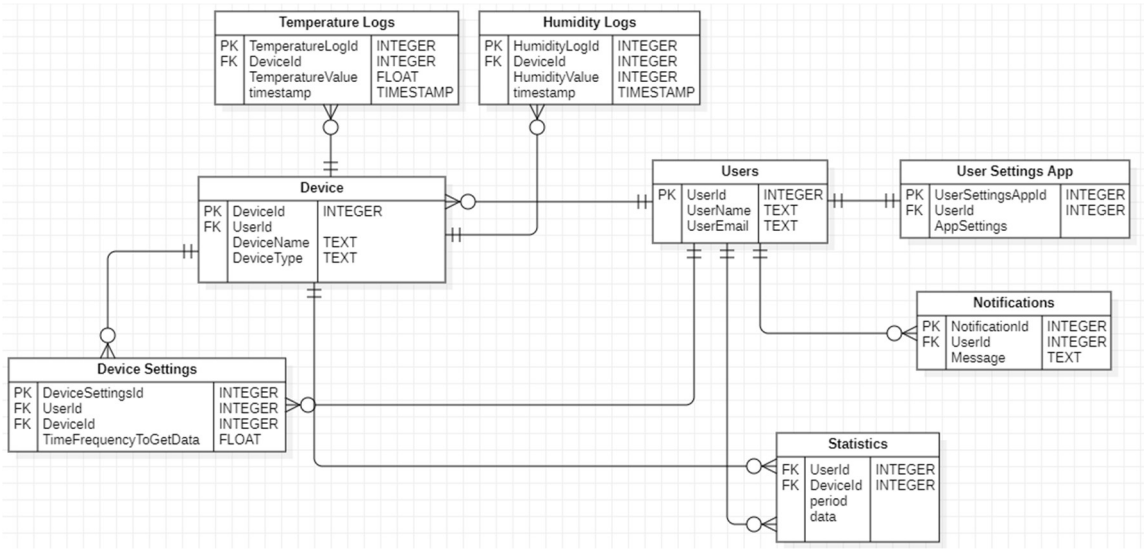

Рисунок 2.1 – Початкова UML-діаграма бази даних для інформаційної системи «розумного» будинку “SweeMe”

Функціями інформаційної системи є:

- Реєстрація користувачів – дозволяє користувачам створювати облікові записи шляхом надання необхідних даних (ім'я, електронна пошта, пароль) і зберігання їх у таблиці «users».
- Автентифікація та авторизація – після реєстрації користувачі отримують токен для автентифікації. Пароль користувачів також хешується для безпеки. Значення поля «is_verified» встановлюється на «1» після успішної підтвердження електронної пошти облікового

запису. Це дозволяє авторизованим користувачам використовувати додаток.

- Додавання пристроїв – користувачі можуть додавати пристрої до своїх облікових записів шляхом сканування QR-коду або введення серійного номера. Інформація про пристрої зберігається в таблиці «devices», де кожен пристрій має унікальний номер та відповідний «user_id», що посилається на власника.
- Отримання даних від пристроїв – пристрої здатні збирати різні дані, такі як температура, вологість, відео з IP-камери та інше. Ці дані відправляються користувачам в залежності від типу пристрою. Інформація з пристроїв може бути збережена відповідно до потреб користувача.

2.1.2 Опис роботи системи

Для того, щоб користуватися «розумним» будинком, користувачу потрібно зареєструватися в додатку. Для цього потрібно розробити ендпоінт реєстрації. Процес реєстрації (рис. 2.2) відбувається таким чином:

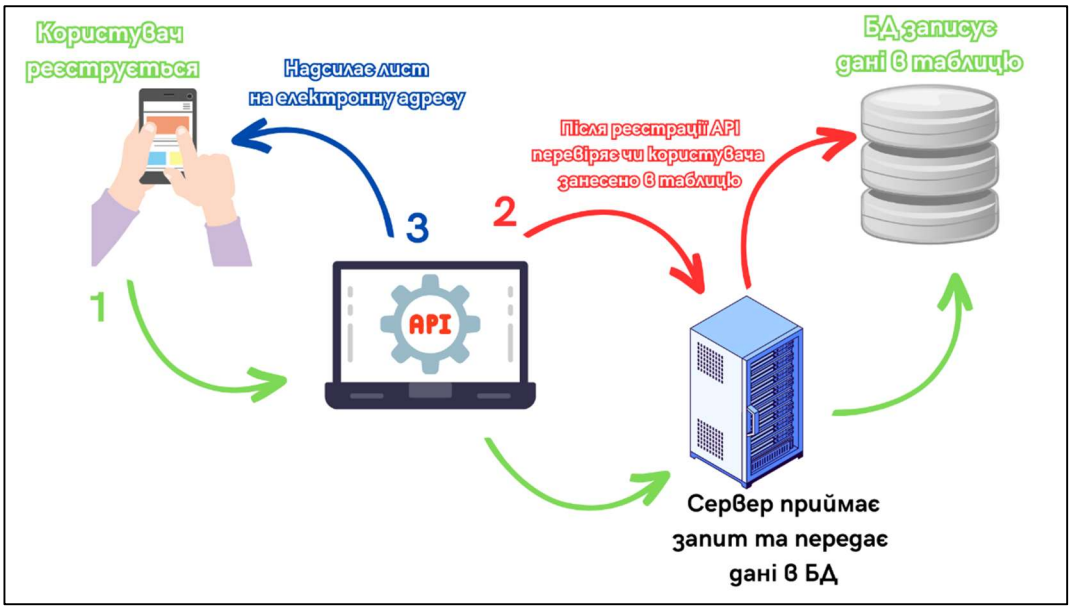

Рисунок 2.2 – Процес реєстрації нового користувача

коли користувач реєструється він надсилає POST-запит у форматі JSON його дані (рис. 2.3) – “name”, “email” та “password” на ендпоінт реєстрації.

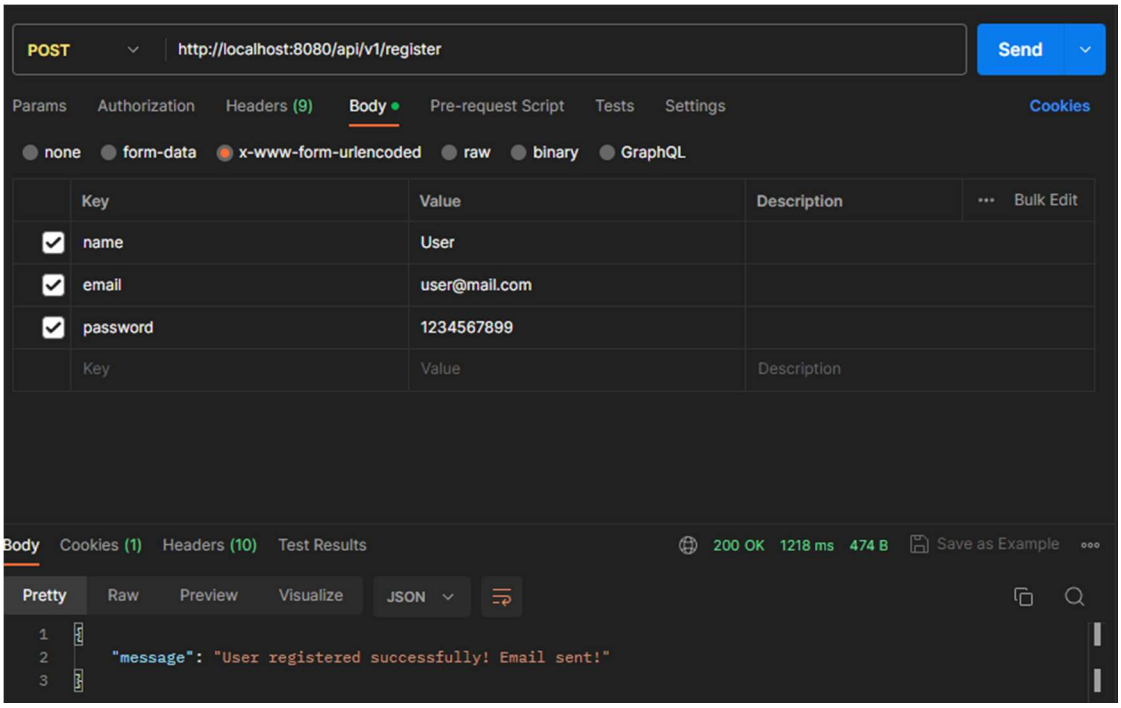

Рисунок 2.3 – Приклад надсилання запиту на ендпоінт реєстрації

API повинно перевірити правильність цих даних, наприклад, чи відповідає пароль мінімальним вимогам – мати не менше ніж 8 символів, чи введена електронна пошта є дійсно електронною поштою. Після обробки, якщо не виникає помилок, користувачу присвоюється токен верифікації, формується посилання для верифікації електронної пошти, яке містить даний токен і надсилається лист з посиланням на вказану електронну пошту користувача, у якому буде повідомлення про реєстрацію та посилання для верифікації. Дані користувача записуються у таблицю “users” (рис. 2.4) у відповідні поля.

id	role_id	name	email	password	restoration_token	created_at	updated_at	verification_token	is_verified
1	NULL	User	user@mail.com	$2a$10$Gs3crNIDEITbRJa5aeQDfegk0cqBvhlnljKOOjfTiTi...	NULL	2023-06-12 14:25:41	2023-06-12 14:26:11	847002a4-e42c-49af-b777-5e28b534f6ed	1

Рисунок 2.4 – Приклад відображення користувача в таблиці

Відбувається хешування паролю, а у поле “is_verified” за замовчуванням встановлюється значення «0». Поки це значення не зміниться на «1», користувач не зможе авторизуватися і користуватися додатком. Якщо лист успішно відправлено, повертається відповідь зі статусом «200» та повідомленням про успішну реєстрацію. Користувач відкриває посилання з отриманого листа, який переадресує його на ендпоінт верифікації. Після переходу за посиланням, програмний код перевіряє, чи існує користувач з вказаним токеном верифікації. Якщо користувача не знайдено, повертається помилка. Якщо користувач з таким токеном існує, відповідно, значення в таблиці зміниться на 1 і він зможе здійснити авторизацію.

Після успішної реєстрації, користувачу потрібно авторизуватися. Для цього використовується ендпоінт авторизації. Користувач надсилає POST-запит з обов'язковими даними – електронною поштою і паролем. Код перевіряє, чи існує користувач з вказаною електронною поштою та паролем. Якщо такого користувача не знайдено, повертається помилка. Далі перевіряється, чи користувач підтвердив свою пошту. Якщо електронна пошта не підтверджена, повертається помилка з відповідним повідомленням. Якщо усі дані пройшли перевірку, генерується JWT з інформацією про користувача.

Коли користувача було авторизовано, він зможе додати собі на головну сторінку його пристрої. Для цього була розроблена система із серійним номером. Кожен тип пристрою має свій унікальний номер. Всі можливі типи пристроїв зберігаються в таблиці “devices”. Користувачу потрібно сканувати QR-код, або ввести серійний номер вручну. Надсилається POST-запит із серійним номером, проводиться перевірка JWT користувача, що міститься у заголовку запиту. Код перевіряє, чи існує такий пристрій із таким ідентифікатором, далі перевіряється чи користувач не має даного пристрою. Якщо пристрій вже призначений даному користувачу, повернеться помилка із повідомленням «У вас вже є даний пристрій». Якщо всі умови коду виконані правильно, в таблиці “devices” присвоюється “user_id” користувача і цей пристрій додається в його акаунт. Далі девайс зможе відправляти дані

користувачу (залежно від девайсу це може бути температура, вологість, відео через IP камеру).

2.2 Аналіз та вибір технологій для інформаційної системи

Проводячи аналіз сучасних засобів розробки та технологій для інформаційної системи «розумного» дому, я вибрав наступні компоненти і технології: Node.js, MySQL, GitHub, Express, Sequelize, JWT, Google OAuth2, Nodemailer, UUID, BcryptJS, Passport, Postman.

Серверне середовище Node.js є популярним для розробки серверної частини інформаційної системи з наступними перевагами:

- Дозволяє використовувати одну мову програмування, а саме JavaScript, яка вже знайома багатьом розробникам. Це полегшує розробку, обмін кодом та співпрацю між розробниками.
- Побудований на базі архітектури, що дозволяє ефективно обробляти багато запитів одночасно з використанням одного потоку. Це забезпечує швидку відповідь на запити і високу продуктивність додатків [9].
- Добре підходить для побудови великих додатків. Він також підтримує кластеризацію, що дозволяє використовувати кілька ядер процесора для обробки запитів, покращуючи продуктивність системи [9].
- Має активну спільноту розробників, яка розробляє безліч сторонніх модулів і пакетів. Це дозволяє використовувати готові рішення для швидкої розробки та функціональності системи, що дозволяє скоротити час розробки і підвищити ефективність.

GitHub – це хостингова платформа для зберігання та керування проектами, розробленими з використанням системи контролю версій Git [10]. Він надає сервіси для спільної роботи над проектами, відстеження змін, управління версіями, зберігання коду та обговорення проблем [11].

Express є одним з найпопулярніших веб-фреймворків для розробки серверних додатків з використанням Node.js. Він має простий та експресивний синтаксис, що дозволяє швидко створювати потужні веб-додатки та API.

Основні переваги Express:

- Він надає потужну систему маршрутизації, яка дозволяє легко визначати різні маршрути та обробники для обробки HTTP запитів. Це дозволяє ефективно керувати маршрутами додатку [12].
- Базується на концепції middleware (проміжне програмне забезпечення), що дозволяє послідовно виконувати функції обробки, які можуть бути виконані перед або після виконання запиту. Гнучкість та модульність Express дозволяє легко розширювати функціональність додатку шляхом використання різних пакетів та middleware [12].
- Як і Node.js, Express має широке та активне співтовариство розробників, яке забезпечує наявність великої кількості ресурсів, документації, прикладів та пакетів. Це спрощує розробку, надаючи доступ до вже наявних рішень та підтримки в разі виникнення питань.

Sequelize є одним з найпопулярніших ORM (Object-Relational Mapping) для Node.js, який дозволяє зручно працювати з базами даних у стилі об'єктно-орієнтованого програмування [13]. MySQL є популярною та потужною реляційною системою управління базами даних, і разом із Sequelize має кілька переваг:

- Дозволяє легко визначати моделі даних, які відповідають таблицям у базі даних. Можна використовувати JavaScript-класи для створення моделей, визначати зв'язки між таблицями та виконувати додаткову логіку, пов'язану з даними.
- Sequelize дозволяє легко керувати міграціями бази даних. Це дає можливість визначати міграційні файли, які містять зміни схеми бази даних, такі як створення таблиць, додавання або видалення стовпців

тощо. Потім можна легко виконувати ці міграції для оновлення схеми бази даних без втрати даних.

Bcrypt є популярним алгоритмом хешування паролів, який забезпечує безпеку. При реєстрації або зміні пароля користувача, пароль проходить процес хешування, що перетворює його на непередбачуваний набір символів. У даному алгоритмі використовуються солі (salt), що є додатковими випадковим значеннями, яке додаються до паролю перед хешуванням. Це робить більшість атак непрактичними. Важливою перевагою bcrypt є його обчислювальна складність, яка дозволяє уповільнити атаки методом перебором всіх можливих комбінацій паролів.

JWT (JSON Web Tokens) є відкритим стандартом для представлення безпеки в форматі JSON. Він використовується для створення токенів, які можуть бути передані між сторонами в безпечному форматі. JWT складається з трьох основних частин: заголовок (header), тіло (payload) та підпис (signature).

Для даної інформаційної системи JWT використовується для забезпечення безсесійної автентифікації та авторизації користувачів. Після успішної аутентифікації користувача, сервер генерує JWT, який містить інформацію про користувача та його права доступу. Цей токен передається клієнту, який може використовувати його для авторизації при наступних запитах до сервера. Сервер перевіряє цілісність та достовірність JWT, що дозволяє контролювати доступ до захищених ресурсів.

У JWT є декілька переваг, зокрема простота використання, масштабованість та можливість передавати додаткову інформацію в токені. Він також не вимагає зберігання стану сесії на сервері, що робить його дуже зручним для розподілених систем та мікросервісної архітектури.

Google OAuth2 є протоколом автентифікації, який дозволяє користувачам використовувати свої облікові записи Google для входу в різноманітні додатки та сервіси. Завдяки пакету passport-google-oauth20, можна легко інтегрувати автентифікацію через Google у додаток.

При використанні Google OAuth2, система буде перенаправляти користувачів на сторінку входу Google, де вони зможуть ввести свої облікові дані. Після успішного входу, Google повертає авторизаційний код до сервера. За допомогою OAuth2 можна обміняти цей код на токен доступу, який можна використовувати для отримання інформації про користувача з облікового запису Google.

Однією з переваг використання даного протоколу є зручність для користувачів, оскільки вони можуть використовувати свій існуючий обліковий запис Google замість створення нового облікового запису. Це зменшує бар'єри для входу в систему та покращує користувацький досвід.

Nodemailer – це популярний пакет для відправки електронних листів з додатків, розроблених на Node.js. Використовуючи його можна легко налаштувати та надіслати електронні повідомлення з інформаційної системи.

Він підтримує різні транспортні методи, такі як SMTP, Sendmail, Amazon SES, що дозволяє розробнику вибрати найбільш зручний метод для конкретної потреби.

Завдяки цьому пакету можна відправляти електронні повідомлення з будь-якої частини системи, включаючи автоматичну відправку листів для підтвердження облікових записів, нагадувань паролів, сповіщень про події або будь-яких інших повідомлень, які потрібні користувачам.

Nodemailer також надає зручний функціонал для створення та форматування електронних повідомлень. Він також підтримує використання шаблонів для створення персоналізованих повідомлень.

UUID (Universally Unique Identifier) – це стандартизований формат для генерації унікальних ідентифікаторів. UUID забезпечує гарантію того, що ідентифікатори, які він генерує, будуть унікальними навіть у розподілених системах без потреби в централізованому механізмі управління.

В інформаційній системі UUID використовується для генерації унікальних ідентифікаторів для різних об'єктів. Наприклад, можна

використовувати UUID для генерації унікальних ідентифікаторів користувачів, записів в базі даних, файлів, сесій тощо.

Використання UUID має декілька переваг. По-перше, воно забезпечує глобальну унікальність, що дозволяє уникнути конфліктів між різними об'єктами у системі. По-друге, UUID є досить великими (128 біт), що робить ймовірність виникнення колізій (повторення ідентифікаторів) дуже малою.

Ним можна легко створювати нові унікальні ідентифікатори за допомогою методів, наданих цим пакетом. Це дозволяє забезпечити унікальність ідентифікаторів у системі без зайвих зусиль.

Середовищем для розробки було обрано WebStorm від JetBrains. WebStorm надає розширений набір функцій, призначених для поліпшення продуктивності розробника веб-додатків. Серед інших інтегрованих середовищ розробки (IDE) його перевагми є:

- Потужний редактор коду з підсвічуванням синтаксису, автодоповненням, перевіркою помилок і великою кількістю корисних функцій редагування.
- Надає широкий набір інструментів для рефакторингу коду, таких як перейменування, витягнення функцій, оптимізація і інше. Це допомагає полегшити процес модифікації та покращення кодової бази.
- WebStorm інтегрується з популярними системами контролю версій, такими як Git, SVN і Mercurial, дозволяючи вам взаємодіяти з репозиторієм, комітити зміни, переглядати історію і розв'язувати конфлікти.
- Інтегрується з інструментами збирання та пакування, такими як webpack, npm і gulp, що дозволяє автоматизувати процеси збирання та пакування проекту.

У якості веб-серверу було обрано Apache HTTP Server. Він відомий своєю стабільністю, надійністю та гнучкістю, підтримує багато протоколів, включаючи HTTP, HTTPS, FTP. За допомогою Apache HTTP Server можна

хостити веб-сайти, створювати веб-додатки, забезпечувати доступ до ресурсів через мережу Інтернет.

Для тестування API використовувався Postman. Це популярний інструмент який надає зручне середовище для створення, відправлення та отримання HTTP-запитів до веб-серверів або інших API.

Висновки до розділу 2

В результаті проведеного аналізу та вибору технологій для інформаційної системи «розумного» дому, було визначено набір технологій, які найкраще відповідають вимогам та потребам проекту.

Також, у розділі були описані функції, структура системи та її робота з прикладом передачі та відображення даних у базу даних.

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Засоби розробки

Розробку інформаційної системи для «розумного» будинку “SweeMe” та API виконано на мові JavaScript та серверному середовищі Node.js.

Системою управління базами даних було обрано MySQL.

Платформою для зберігання та керування проектами, з використанням системи контролю версій Git обрано GitHub.

Для тестування API використовується Postman.

У програмний код було імпортовано такі бібліотеки:

1. Express – веб-фреймворк для розробки серверних додатків з використанням Node.js.
2. Sequelize – ORM (Object-Relational Mapping) для Node.js, який дозволяє зручно працювати з базами даних.
3. JWT – використовується для створення токенів, які можуть бути передані між сторонами в безпечному форматі.
4. Bcryptjs – алгоритм хешування паролів
5. Mysql2 – є покращеною альтернативою популярному пакету mysql для Node.js, пропонуючи вдосконалену продуктивність та підтримку новіших функцій MySQL.
6. Passport – пакет для автентифікації
7. Passport-Google-OAuth2 – протокол автентифікації, який дозволяє користувачам використовувати свої облікові записи Google для входу.
8. Nodemailer – пакет для відправки електронних листів з додатків, розроблених на Node.js.
9. UUID – стандартизований формат для генерації унікальних ідентифікаторів.
10. Cors (Cross-Origin Resource Sharing) – механізм безпеки веб-браузерів, який обмежує взаємодію між веб-додатками, запущеними на різних доменах.

11. Swagger-UI-express – є пакетом для Node.js, який дозволяє генерувати та відображати інтерактивну документацію API засобами Swagger UI
12. Swagger-jsdoc – є пакетом для Node.js, який дозволяє створювати специфікацію API засобами JSDoc та Swagger.

3.2 Опис реалізації системи

За допомогою Node.js реалізовано підсистеми бази даних, MySQL використано як систему керування базою даних. Розроблено фізичну модель бази даних. Кількість таблиць становить 10 таблиць (рис. 3.1).

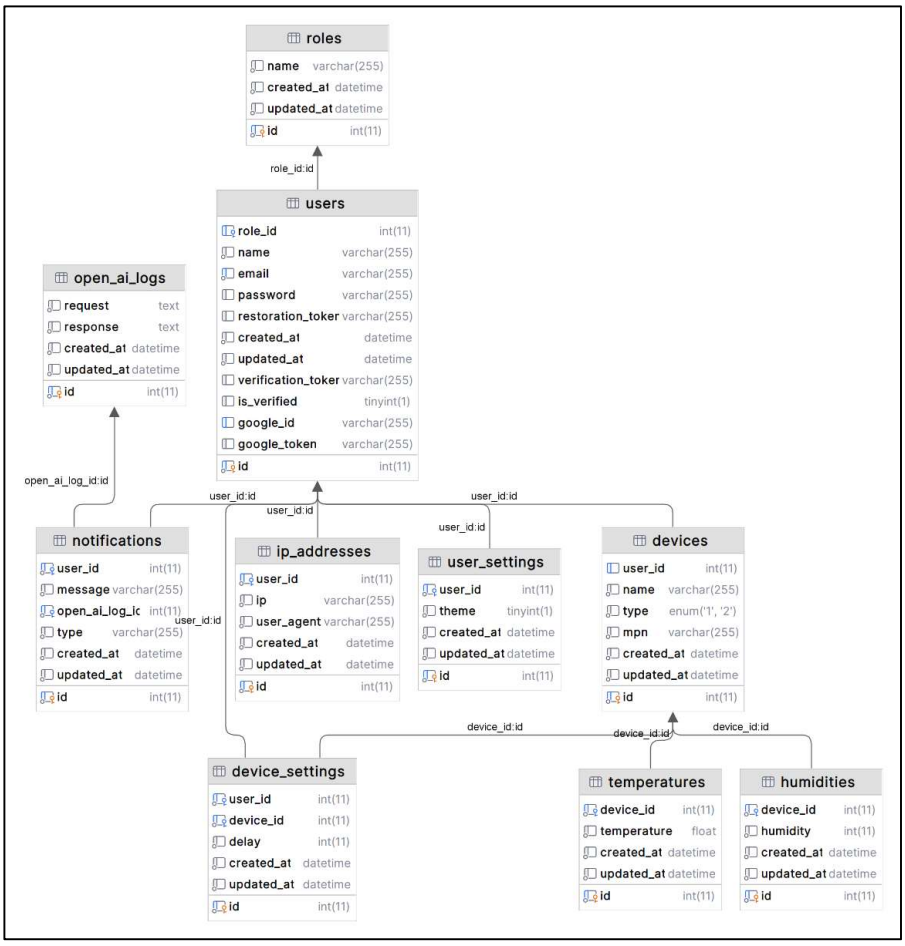

Рисунок 3.1 – Остаточна UML-діаграма бази даних для інформаційної системи «розумного» будинку “SweeMe”

Таблиця «users» призначена для збереження інформації про користувачів. Опис полів:

- id – ключове поле таблиці (primary key);
- role_id – поле призначене для ролі користувача (superuser, user), зовнішній ключ (foreign key);
- name – ім’я користувача;
- email – електронна пошта користувача;
- password – пароль користувача;
- restoration_token – код для відновлення у разі втрати паролю;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці;
- verification_token – токен, який генерується для верифікації акаунту через електронну пошту;
- is_verified – поле, яке вказує чи користувач підтвердив свою пошту (за замовчуванням поле має значення 0);
- google_id – поле, яке зберігає Google id для автентифікації через google-oauth2;
- google_token – поле, яке зберігає Google token для автентифікації через google-oauth2.

Таблиця «devices» призначена для зберігання даних про пристрої. Опис полів:

- id – ключове поле таблиці (primary key);
- user_id – поле для зберігання ідентифікатора користувача (foreign key);
- name – ім’я девайсу;
- type – тип девайсу;
- mpn – серійний номер девайсу;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «user_settings» призначена для зберігання налаштувань користувача в середині додатку. Опис полів:

- id – ключове поле таблиці (primary key);

- user_id – поле для зберігання ідентифікатора користувача (foreign key);
- theme – колірна схема інтерфейсу;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «device_settings» призначена для зберігання налаштувань пристрою. Опис полів:

- id – ключове поле таблиці (primary key);
- user_id – поле для зберігання ідентифікатора користувача (foreign key);
- device_id – поле для збереження ідентифікатора пристрою;
- delay – поле для вибору частоти надходження інформації;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «roles» призначена для зберігання ролей користувачів. Опис полів:

- id – ключове поле таблиці (primary key);
- name – ім’я ролі;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «notifications» призначена для зберігання інформації сповіщень від пристроїв. Опис полів:

- id – ключове поле таблиці (primary key);
- user_id – поле для зберігання ідентифікатора користувача (foreign key);
- message – поле для зберігання повідомлення;
- open_ai_log_id – поле для зберігання ідентифікатора логів ChatGPT;
- type – тип сповіщення;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «open_ai_logs» призначена для зберігання інформації під час обміну повідомленнями з ChatGPT. Опис полів:

- id – ключове поле таблиці (primary key);
- request – поле для зберігання запиту;
- response – поле для збереження відповіді на запит;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «temperatures» призначена для зберігання показників температури з пристрою. Опис полів:

- id – ключове поле таблиці (primary key);
- device_id – поле для збереження ідентифікатора пристрою (foreign key);
- temperature – поле для збереження даних про температуру;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «humidities» призначена для зберігання показників вологості з пристрою. Опис полів:

- id – ключове поле таблиці (primary key);
- device_id – поле для збереження ідентифікатора пристрою (foreign key);
- humidity – поле для збереження даних про вологість;
- created_at – дата та час творення об’єкту в таблиці;
- updated_at – дата та час редагування об’єкту в таблиці.

Таблиця «ip_adresses» призначена для збереження IP-адрес. Опис полів:

- id – ключове поле таблиці (primary key);
- user_id – поле для зберігання ідентифікатора користувача (foreign key);
- ip – поле для збереження IP-адреси;
- user_agent – дані про браузер;
- created_at – дата та час творення об’єкту в таблиці;

- updated_at – дата та час редагування об’єкту в таблиці.

За допомогою веб-серверу Apache був налаштований локальний сервер, який дозволить надсилати і приймати запити (рис. 3.2).

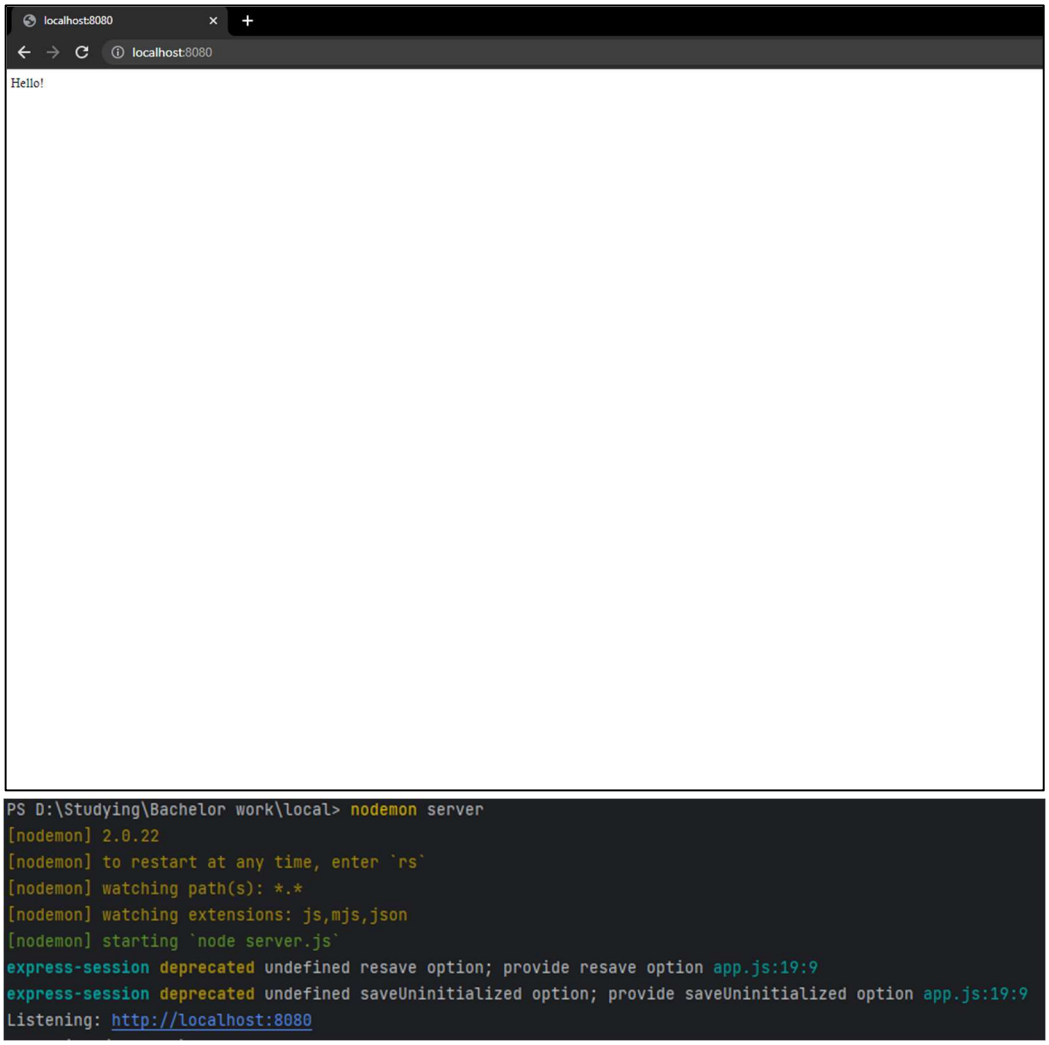

Рисунок 3.2 – Демонстрація роботи серверу.

За допомогою Sequelize було налаштовано з’єднання з базою даних та створено всі необхідні моделі згідно з UML-діаграмою (рис. 3.3), а також налаштовані зв’язки між таблицями.

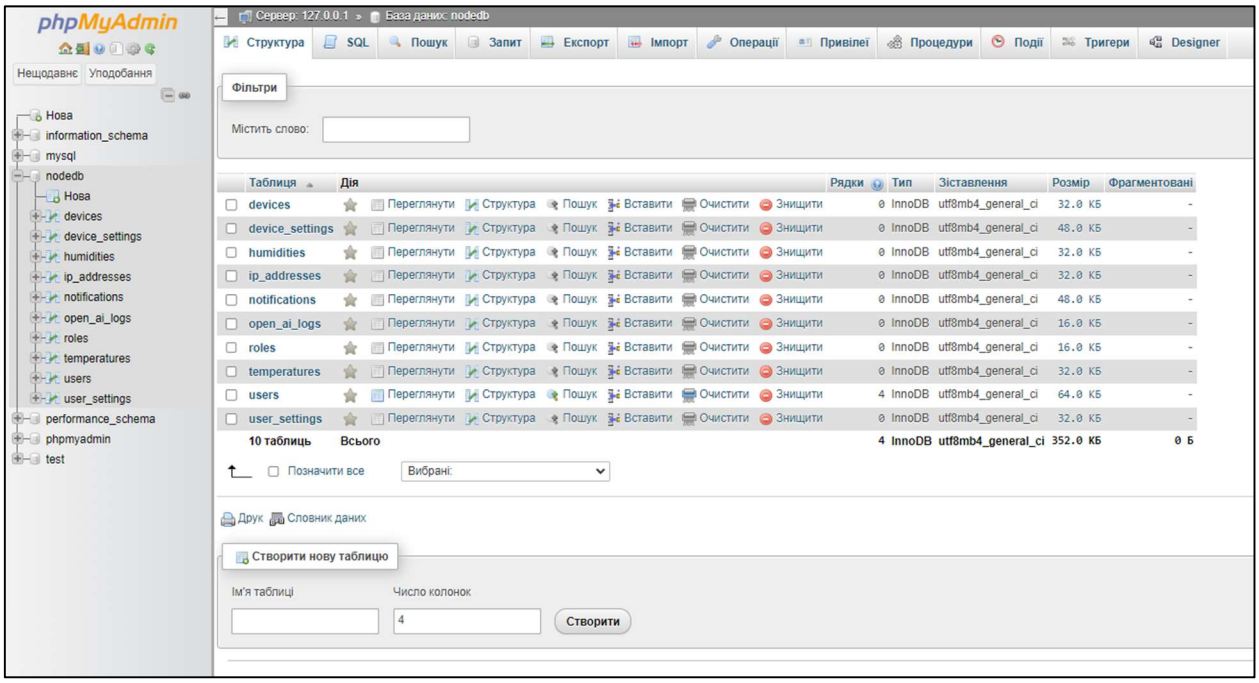

Рисунок 3.3 – Відображення таблиць у базі даних «nodedb»

Для створення ендпоінту реєстрації використані такі пакети: bcryptjs, nodemailer, uuid

Використовуючи POST-запит був написаний код, який приймає запит від користувача з його даними у тілі запиту, а саме «name», «email», «password» (див. рис. 2.3). Після того як користувач надіслав запит, програма перевіряє чи відповідають дані вимогам, а саме: чи поля не пусті, чи відповідає пароль довжині 8 символів, та чи введена електронна пошта написана у відповідному форматі.

Після обробки даних, якщо в процесі роботи програми не виникає помилок, користувачу присвоюється токен верифікації, який генерується за допомогою UUIDv4. Коли токен згенеровано, він зберігається в таблиці «user» в полі «verification_token». Створюється ендпоінт для верифікації, який містить токен для порівняння із токеном користувача для верифікації електронної пошти. Використовуючи пакет nodemailer надсилається лист з посиланням на вказану електронну пошту користувача, у якому буде повідомлення про реєстрацію та сформоване посилання для верифікації (рис. 3.4).

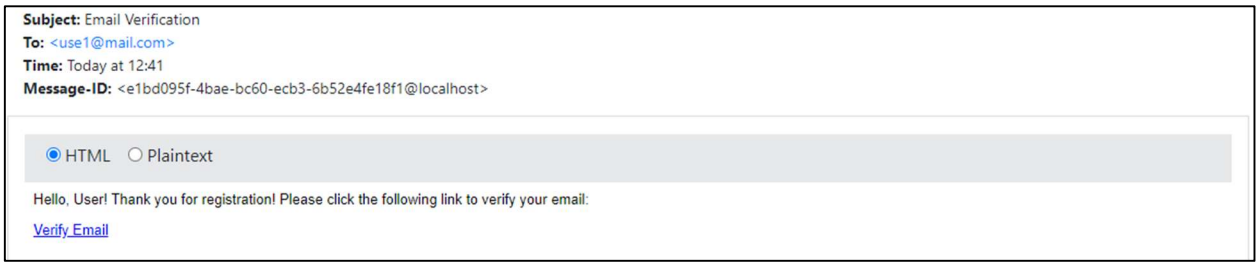

Рисунок 3.4 – Приклад листа для верифікації користувача

Для того, щоб nodemailer зміг надсилати лист користувачу, його потрібно налаштувати. Потрібно створити файл конфігурації, який буде містити наступні дані про адресу SMTP-сервера, що використовується, порт за допомогою якого відбувається з'єднання з сервером, тип з'єднання (захищене чи незахищене) та дані про електронну пошту, яка призначена для надсилання листів (рис. 3.5).

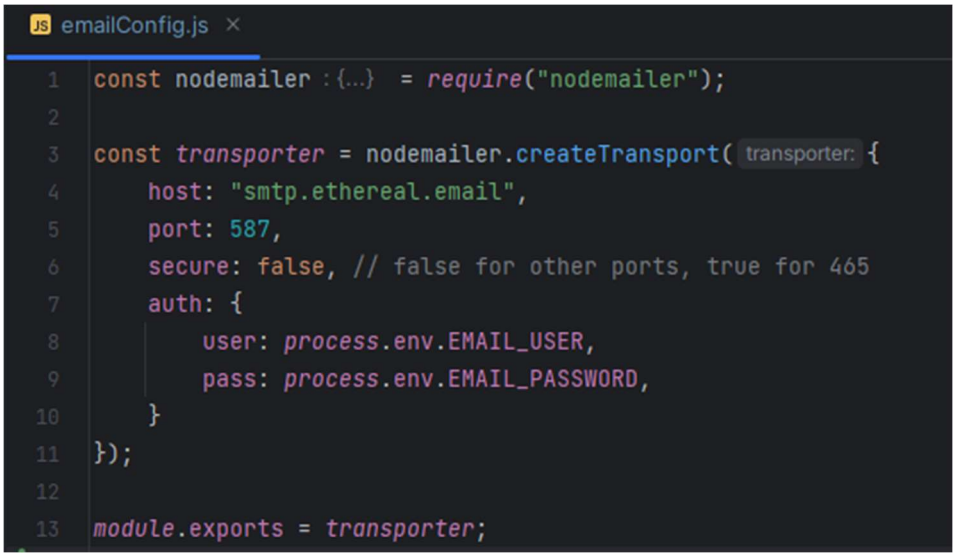

Рисунок 3.5 – Файл конфігурації nodemailer

Після успішної реєстрації дані користувача записуються у таблицю "users" у відповідні поля.

Коли користувач переходить по посиланню він отримує відповідь від API, що його пошта була успішно верифікована (рис 3.6).

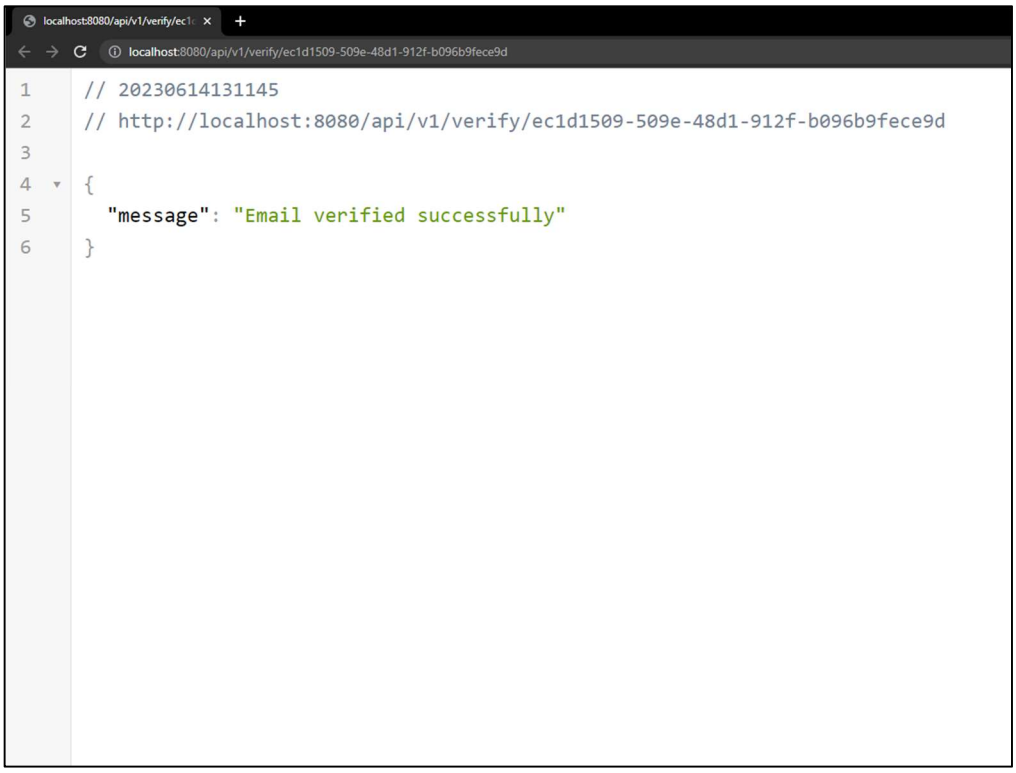

Рисунок 3.6 – Відповідь сервера на успішну верифікацію користувача

Відбувається хешування паролю, а у поле “is_verified” за замовчуванням встановлюється значення «0». Поки це значення не зміниться на «1», користувач не зможе авторизуватися і користуватися додатком.

Додатковим методом реєстрації та авторизації було обрано протокол автентифікації Google OAuth2, який дозволяє користувачам використовувати свої облікові записи Google. Щоб підключити його до програми потрібно створити файл конфігурації passport у якому будуть прописані методи автентифікації (рис. 3.7).

Для авторизації користувача було свторено ендпоінт у якому користувач надсилає POST-запит у тілі якого є обов'язкові дані – електронна пошта і пароль. Програмний код перевіряє, чи існує користувач з вказаною електронною поштою та паролем. Якщо такого користувача не знайдено, повертається помилка. Якщо в ході виконання помилки відсутні, перевіряється чи користувач верифікував свою пошту. Якщо електронна пошта не

підтверджена, повертається помилка з відповідним повідомленням. Якщо усі дані пройшли перевірку, генерується JWT з інформацією про користувача.

```
passport.use(
    new GoogleStrategy( options: {
        clientID: process.env.GOOGLE_CLIENT_ID,
        clientSecret: process.env.GOOGLE_CLIENT_SECRET,
        callbackURL: "http://localhost:8080/api/v1/google/callback"
    },
        verify: async (accessToken, refreshToken, profile, done) : Promise<...>  => {
            try {
                const { id, displayName, emails } = profile;

                // Check if the user already exists in the database
                const [user : Model<any, TModelAttributes> , created : boolean ] = await User.findOrCreate( options: {
                    where: { google_id: id },
                    defaults: {
                        name: profile.displayName,
                        email: profile.emails[0].value,
                        google_id: profile.id,
                        google_token: accessToken,
                    },
                });

                if (!created) {
                    return done(null, user);
                }

                user.is_verified = 1;
                await user.save();

                return done(null, user);
            } catch (error) {
                console.error('Error saving Google user:', error);
                return done(error, null);
            }
        }
));
```

Рисунок 3.7 – Фрагмент коду по реалізації автентифікації користувача через Google

Для комфортної роботи з API було створено Swagger документацію (рис. 3.8).

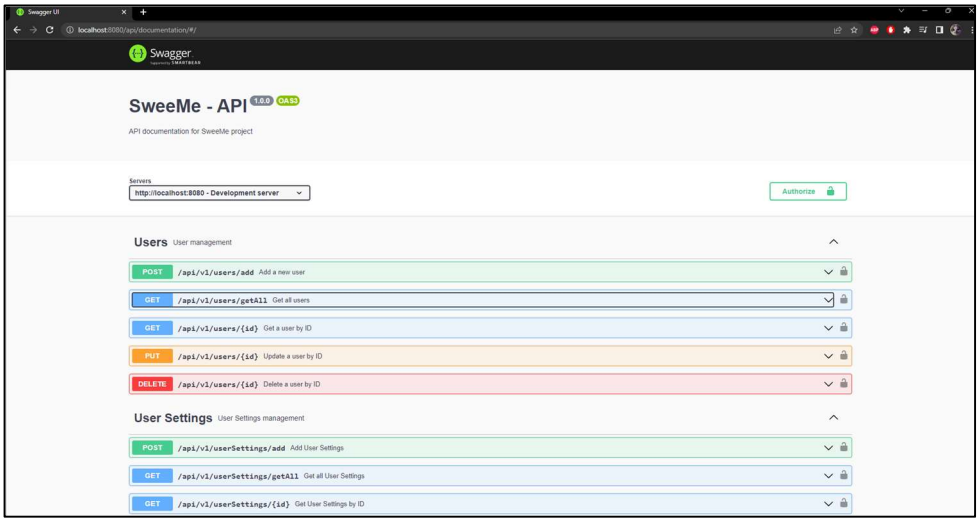

Рисунок 3.8 – Вигляд Swagger документації

Вона надає структуроване описання всіх доступних ендпоінтів, параметрів, типів даних і операцій, що дозволяє розробникам легко зрозуміти та використовувати API. Swagger документація використовуєься для створення автоматизованих тестів API, що допомагає впевнитися в правильному функціонуванні API перед випуском.

Для того, щоб Swagger документація коректно працювала з базою даних та API були прописані ендпоінти, маршрути та контролери.

Для контролю версій був створений репозиторій на платформі GitHub (рис. 3.9).

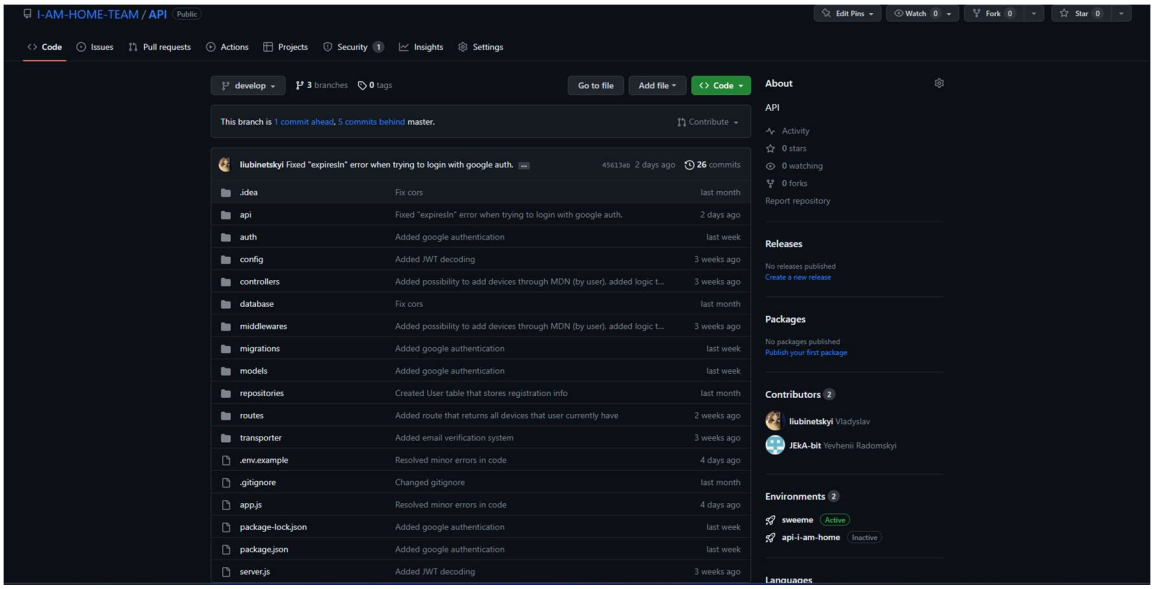

Рисунок 3.9 – Репозиторій програмного коду на GitHub

3.3 Аналіз отриманих результаті

Розглянемо процес реєстрації, авторизації, додавання пристрою до акаунту користувача.

Користувач вводить дані в тіло та відправляє запит. Сервер відповідає йому, що користувача було зареєстровано та лист для верифікації електронної пошти надіслано (рис. 3.10).

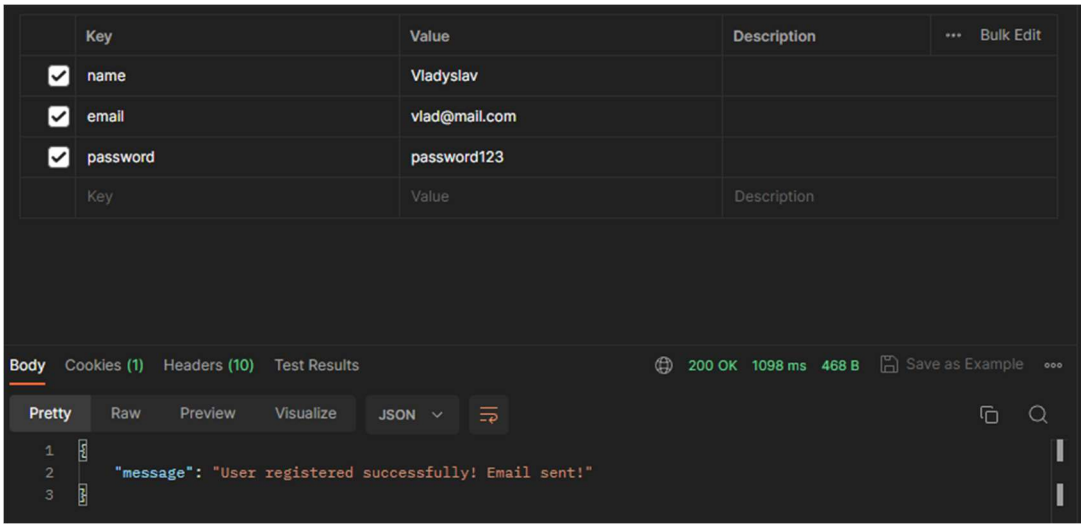

Рисунок 3.10 – Надсилання запиту на реєстрацію

Дані, які користувач вводив, передаються в базу даних, поле «is_verified» встановлює значення «0», а API генерує «verification_token» через який буде здійснена верифікація користувача через електронну пошту (рис. 3.11).

Рисунок 3.11 – Створений користувач у таблиці «users»

Для того, щоб авторизуватися потрібно підтвердити пошту. Якщо цього не зробити, повернеться помилка, що наш акаунт не верифікований (рис. 3.12).

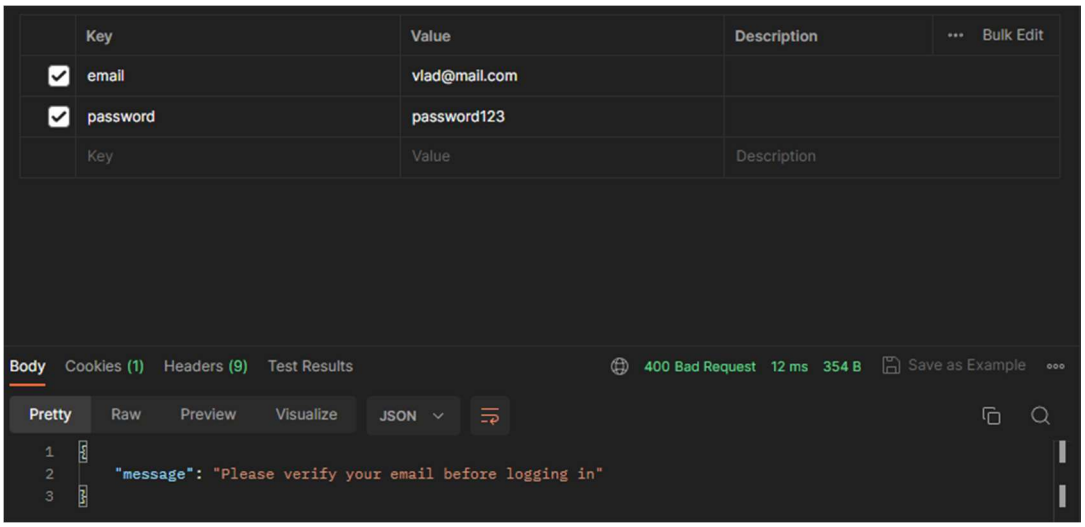

Рисунок 3.12 – Помилка авторизації через не підтверджену електронну пошту

Щоб підтвердити пошту потрібно перейти по згенерованому посиланню, в якому передається і звіряється «verification_token» (рис. 3.13).

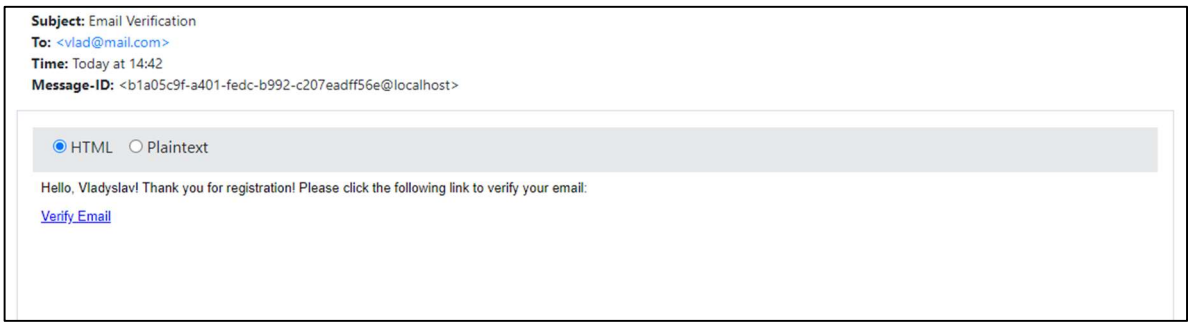

Рисунок 3.13 – Електронний лист із згенерованим посиланням верифікації

Після того, як користувач переходить по посиланню, він бачить повідомлення, що його пошту верифіковано (рис. 3.14), а поле «is_verified» зммінює значення на «1» (рис. 3.15).

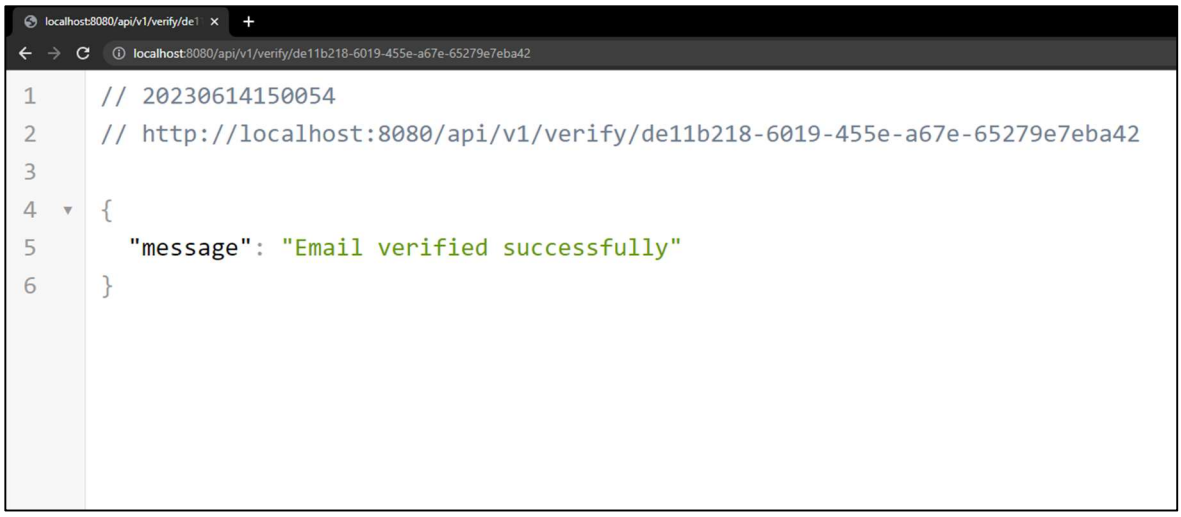

Рисунок 3.14 – Повідомлення про успішну верифікацію електронної пошти

Рисунок 3.15 – Поле «is_verified» має значення «1»

Коли акаунт було верифіковано, можна здійснити авторизацію. Для цього потрібно відправити запит на ендпоінт авторизації з даними в тілі запиту. Якщо користувач успішно авторизований, він отримає повідомлення про успішну авторизацію, а також згенерований токен (рис. 3.16).

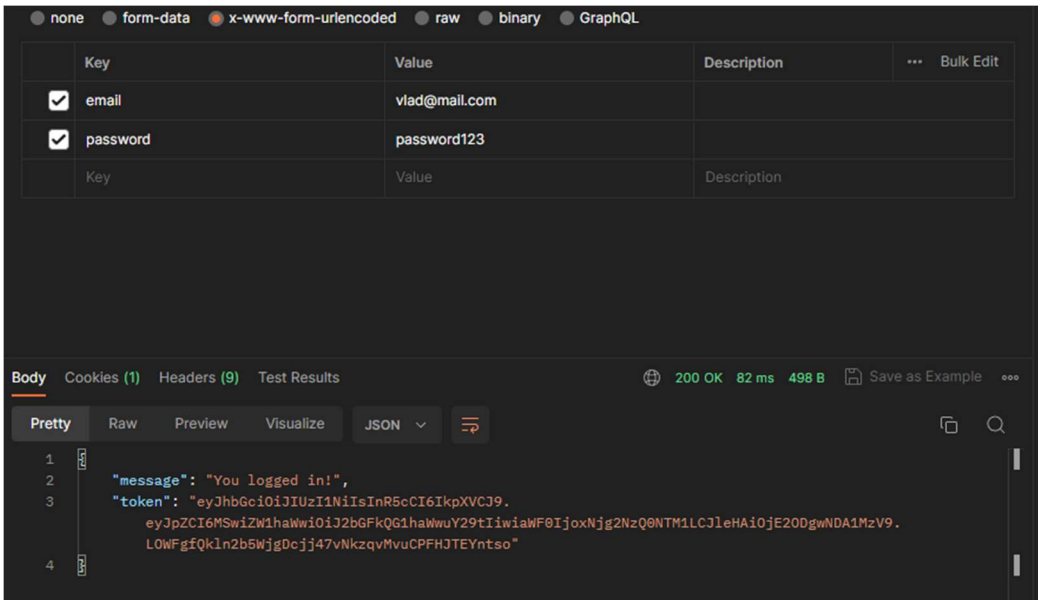

Рисунок 3.16 – Приклад успішної авторизації користувача

Тепер авторизований користувач може додати пристрій. Оскільки він має згенерований токен, ендпоінт додавання пристроїв зможе дізнатися ідентифікатор користувача і присвоїти йому пристрій. В таблиці було попередньо створено пристрій (рис. 3.17).

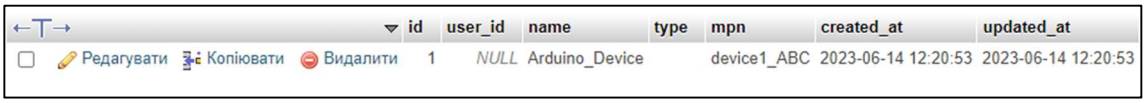

Рисунок 3.17 – Пристрій в таблиці «devices»

Щоб користувачу додати пристрій, потрібно надіслати запит на ендпоінт додавання пристрою. API перевірить, чи існує такий пристрій із таким ідентифікатором. Якщо так, то в таблиці «devices» у полі «user_id» пристрою буде присвоєно ідентифікатор користувача.

Оформлюється запит таким чином: в тілі запиту потрібно вказати серійний номер (рис. 3.18),

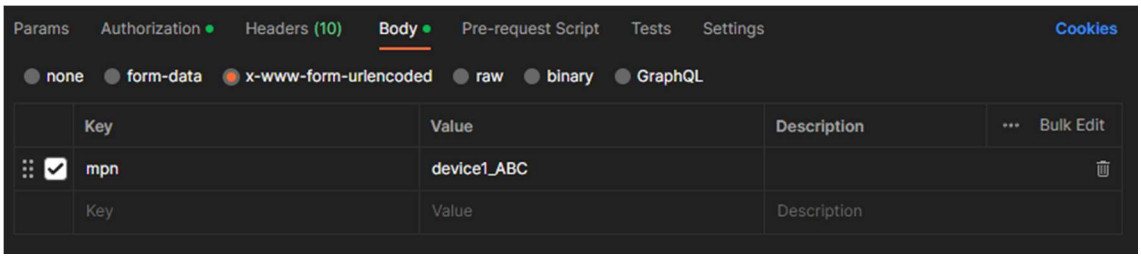

Рисунок 3.18 – Тіло запиту для додав

а в заголовках запиту потрібно вказати токен користувача (рис. 3.19).

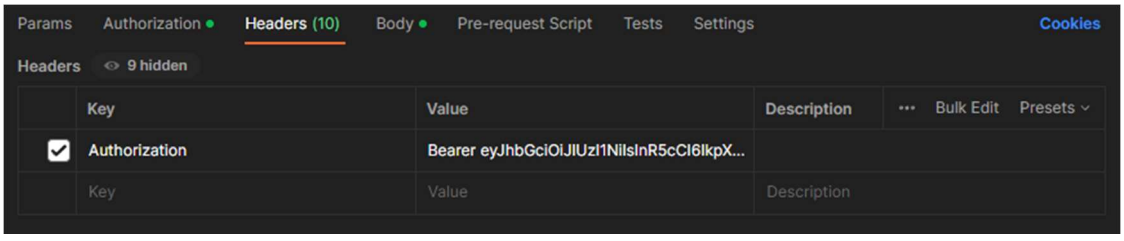

Рисунок 3.19 – Заголовок запиту з токеном користувача

Якщо всі дані введено правильно API поверне пристрій у JSON-форматі, де вже буде змінено поле «user_id» на ідентифікатор користувача (рис. 3.20).

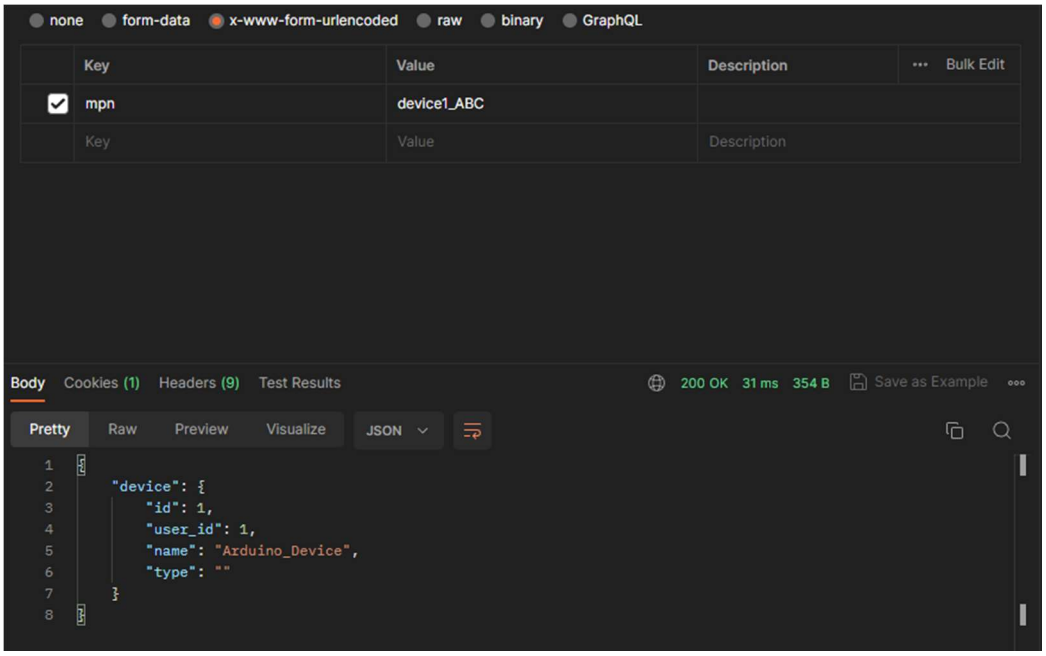

Рисунок 3.20 – Успішна відповідь на запит користувача про додавання пристрою

В базі даних дані також оновилися, поле «user_id» змінилося на ідентифікатор користувача, який надіслав запит, і тепер цей пристрій належить йому (рис. 3.21).

Рисунок 3.21 – Оновлена інформація в таблиці «devices»

Якщо той самий користувач знову відправить запит на додавання того ж самого пристрою (з таким ідентифікаційним номером), то отримає помилку, що такий пристрій уже є у користувача (рис. 3.22).

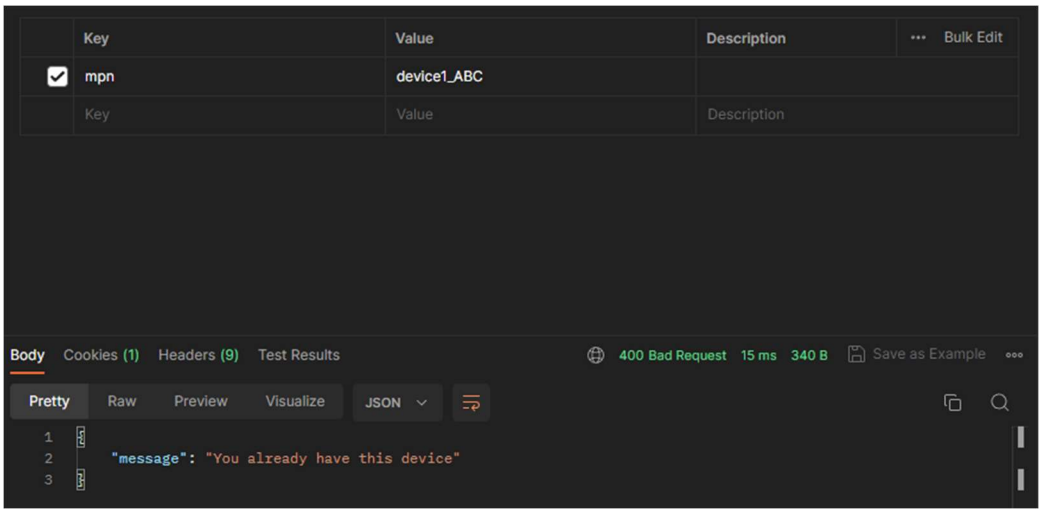

Рисунок 3.22 – Приклад помилки, коли користувач намагається знову додати пристрій

Тепер пристрій може надсилати дані користувачу. Для цього було розроблено ендпоінти, які будуть брати інформацію з бази даних та повертатися у форматі JSON.

Щоб отримати дані від пристрою про температуру, користувач в заголовках зпиту надсилає JWT (рис. 3.23),

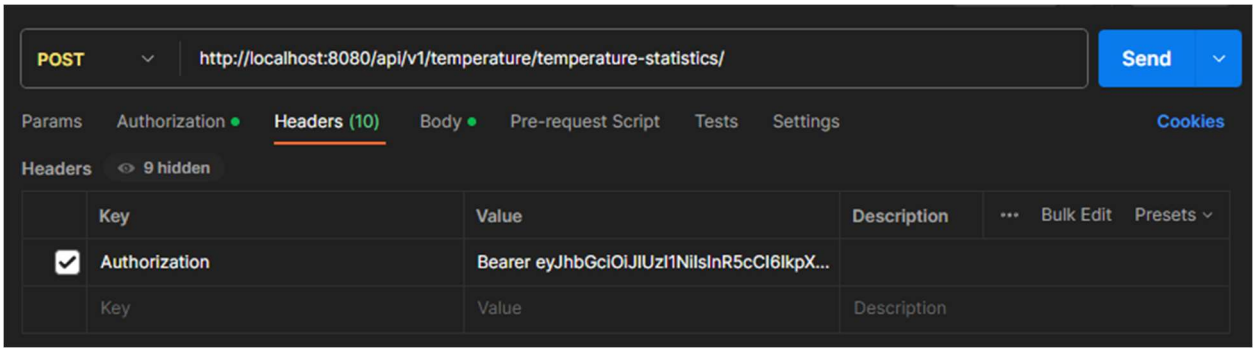

Рисунок 3.23 – Заголовки запиту користувача

а в тілі – параметр у якому визначено за який час він хоче отримати дані. Для прикладу користувач хоче отримати статистику зміни температури за останні 60 хвилин. Він повинен зробити запит з параметром «delay» в якому вказане значення «60» (рис. 3.24).

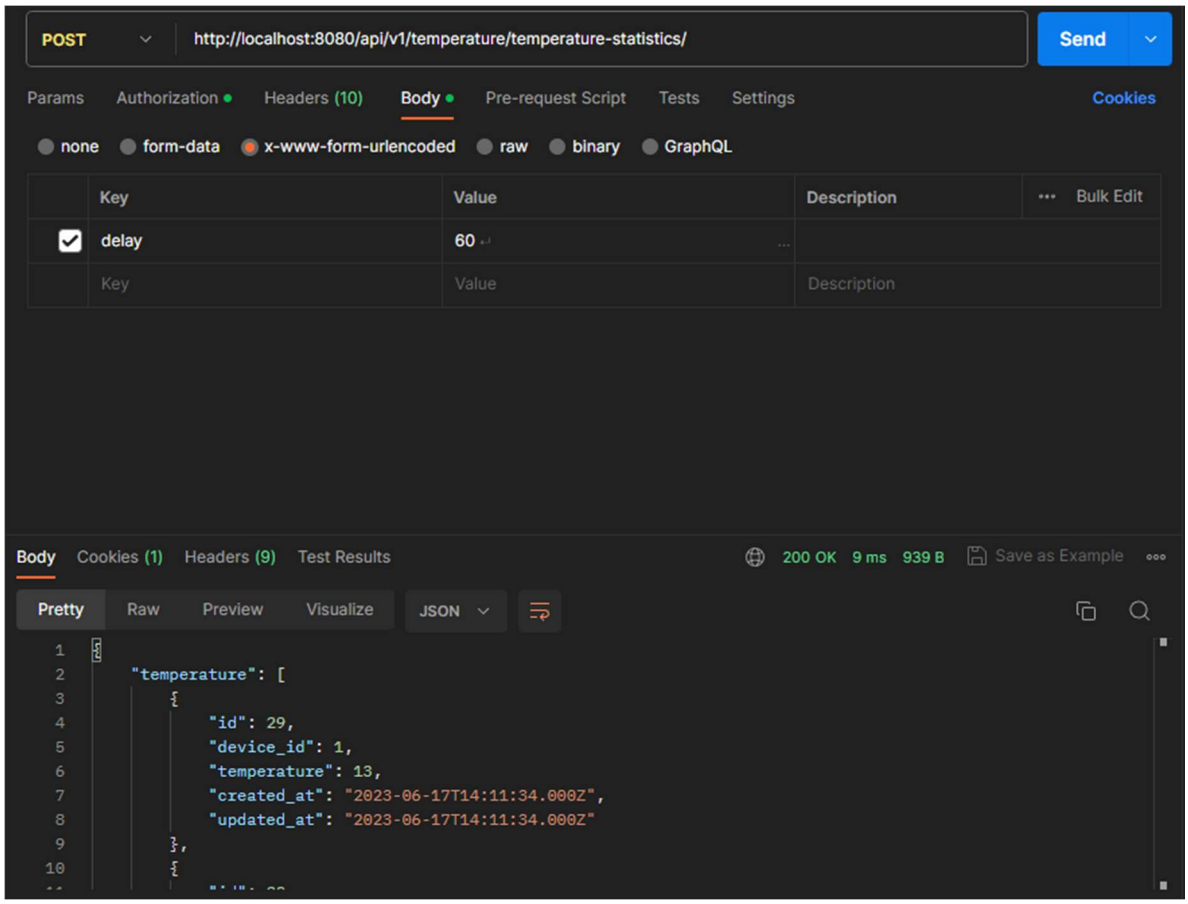

Рисунок 3.24 – Тіло запиту користувача з параметром «delay» 60 хвилин

В результаті запиту він отримає відповідь у форматі JSON про дані з пристрою за останні 60 хвилин, а також середнє значення температури за цей проміжок часу (рис 3.25).

Рисунок 3.25 – Дані про температуру за останні 60 хвилин

Висновки до розділу 3

У даному розділі були розглянуті та обрані засоби розробки, описані їхні переваги і аргументи щодо доцільності їхнього використання. Була описана реалізація інформаційної системи, як вона взаємодіє з API.

Фінальною частиною розділу була демонстрація роботи інформаційної системи, де було продемонстровано основні можливості та те, як система взаємодіє з користувачем.

Робота інформаційної системи та API виконується без помилок та здатна працювати з додатком, який розробляється на її основі. Система є досить гнучкою та має великий потенціал для модифікацій та покращень.

ВИСНОВКИ

В ході проектування та розробки інформаційно системи та API для «розумного» будинку «SweeMe» була досягнута мета, та виконані усі поставлені задачі.

Задачі дослідження включали в себе аналіз предметної області, проектування інформаційної системи та вибір інструментарію та методів для реалізації цієї системи. Аналіз предметної області містив вивчення сучасних технологій та рішень у галузі «розумних» будинків, а також виявлення основних потреб і вимог користувачів. На основі цього аналізу було здійснено проектування інформаційної системи "SweeMe" та розробку API для взаємодії з «розумним» будинком.

В ході виконання розробки був отриманий досвід та навички, які дозволять краще працювати з подібними інформаційними системами, а також полегшить подальшу розробку вже існуючої системи.

Розробка даної системи виявилася актуальною. В ході аналізу ринку та подібних систем виявилося, що розробити власну систему дозволить заощадити кошти, а також вона буде мати потенціал для подальшого розвитку і удосконалення. Дана система може конкурувати з іншими, оскільки вона передбачає розробку і впровадження пристроїв, які можуть бути заточені під конкретного клієнта. Також дана система може використовуватися не тільки в приватних будинках чи квартирах, а також на підприємствах та різних бізнесах.

Застосування даної інформаційної системи дозволить економити електроенергію, зручно керувати пристроями, розробляти власні системи та легко інтегрувати їх.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Грицунов О. В. Інформаційні системи та технології. Навчальний посібник. — Х. / О. В. Грицунов. – Харків: ХНАМГ, 2010. – 222 с.
2. Берко А. Ю. Організація баз даних: практичний курс: Навч. посіб. для студ. / А. Ю. Берко, О. М. Верес. – Львів: Львівська політехніка, 2003. – 152 с.
3. Трофименко О. Г. Організація баз даних. Навчальний посібник / О. Г. Трофименко, Ю. В. Прокоп, Н. І. Логінова, І. М. Копитчук. – Одеса: Фенікс, 2019. – 241 с.
4. Mike McGrath. SQL in easy steps, 3rd edition: For web developers, programmers and students In Easy Steps / Mike McGrath. – In Easy Steps, 2012. – 192 с.
5. API-fication Core Building Block of the Digital Enterprise [Електронний ресурс]. – Режим доступу: https://www.hcltech.com/sites/default/files/documents/resources/whitepaper/files/apis_for_dsi.pdf
6. Fernando Doglio. REST API Development with Node.js: Manage and Understand the Full Capabilities of Successful REST Development / Fernando Doglio. – Apress, 2018. – 323 с.
7. Tom Hughes-Croucher. Node: Up and Running: Scalable Server-Side Code with JavaScript / Tom Hughes-Croucher, Mike Wilson. – "O'Reilly Media, Inc.", 2012. – 204 с.
8. Cory Gackenheimer. Node.js Recipes: A Problem-Solution Approach / Cory Gackenheimer. – Apress, 2013. – 376 с.
9. Azat Mardan. Full Stack JavaScript: Learn Backbone.js, Node.js, and MongoDB / Azat Mardan. – Apress, 2018. – 300 с.
10. Peter Bell. Introducing GitHub: A Non-Technical Guide / Peter Bell, Brent Beer. – "O'Reilly Media, Inc.", 2014. – 142 с.

11. Sarah Guthals. GitHub For Dummies / Sarah Guthals. – John Wiley & Sons, 2023. – 352 c. Dhruti-Shah. NS Guidebookode.J:
12. Ethan Brown. Web Development with Node and Express: Leveraging the JavaScript Stack / Ethan Brown. – "O'Reilly Media, Inc.", 2019. – 346 c.
13. Comprehensive guide to learn Node.js / Dhruti-Shah. – BPB Publications, 2019. – 271 c.

Додаток А. Код програми

app.js

```
const bodyParser = require("body-parser");
const express = require("express");
const session = require("express-session");
const cors = require("cors")
const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');
require("dotenv").config();


require('./models/user');

const api = require ('./api');

const isLoggedIn = (req, res, next) => {
   req.user ? next() : res.sendStatus(401);
}

const app = express();
app.use(session({ secret: process.env.SESSION_SECRET }));
const passport = require("passport");
require("./auth/passport");

app.use(passport.initialize());
app.use(passport.session());

const whitelist = ["http://localhost:8080"]
const corsOptions = {
   origin: function (origin, callback) {
      if (!origin || whitelist.indexOf(origin) !== -1) {
         callback(null, true)
      } else {
         callback(new Error("Not allowed by CORS"))
      }
   },
   credentials: true,
}
app.use(cors(corsOptions))

const swaggerOptions = {
   openapi: '3.0.0',
   info: {
      title: 'SweeMe - API',
      version: '1.0.0',
      description: 'API documentation for SweeMe project'
   },
   components: {
      securitySchemes: {
         BearerAuth: {
            type: 'http',
            scheme: 'bearer',
            bearerFormat: 'JWT'
         }
      }
```

```
  },
  security: [
    {
      BearerAuth: []
    }
  ],
  servers: [
    {
      url: 'http://localhost:8080',
      description: 'Development server'
    },
    {
      url: 'https://sweeme.com.ua',
      description: 'Production server'
    }
  ],
};

const swaggerDocs = swaggerJsdoc({
  swaggerDefinition: swaggerOptions,
  apis: ['./routes/userRouter.js', './routes/userSettingsRouter.js', './routes/devicesRouter.js',
'./routes/deviceSettingsRouter.js', './routes/rolesRouter.js', './routes/temperatureRouter.js', './routes/humidityRouter.js',
'./routes/ipAddressRouter.js', './routes/notificationRouter.js', './routes/openAiLogsRouter.js']
});

app.use(bodyParser.urlencoded({extended: true}));
app.use(bodyParser.json());

/** --- Routers --- */

const userRouter = require('./routes/userRouter');
app.use("/api/v1/users", userRouter);

const userSettingsRouter = require('./routes/userSettingsRouter');
app.use("/api/v1/userSettings", userSettingsRouter);

const devicesRouter = require('./routes/devicesRouter');
app.use("/api/v1/devices", devicesRouter);

const deviceSettingsRouter = require('./routes/deviceSettingsRouter');
app.use("/api/v1/deviceSettings", deviceSettingsRouter);

const rolesRouter = require('./routes/rolesRouter');
app.use("/api/v1/roles", rolesRouter);

const temperatureRouter = require('./routes/temperatureRouter');
app.use("/api/v1/temperature", temperatureRouter);

const humidityRouter = require('./routes/humidityRouter');
app.use("/api/v1/humidity", humidityRouter);

const ipAddressRouter = require('./routes/ipAddressRouter');
app.use("/api/v1/ipaddress", ipAddressRouter);

const notificationRouter = require('./routes/notificationRouter');
app.use("/api/v1/notification", notificationRouter);

const openAiLogsRouter = require('./routes/openAiLogsRouter');
```

```
app.use("/api/v1/openailogs", openAiLogsRouter);

app.use("/api/v1", api);

app.use('/api/documentation', swaggerUi.serve, swaggerUi.setup(swaggerDocs));

app.use('/api/v1/google-test', (req, res) => {
   res.send('<a href="/api/v1/google">Authenticate with Google</a>')
});

app.get('/api/v1/google', passport.authenticate('google', { scope: ['email', 'profile'] }));

app.get('/api/v1/google/callback',
   passport.authenticate('google', {
     successRedirect: '/',
     failureRedirect: '/api/v1/register'
   })
)

app.use(express.json());

app.get('/', isLoggedIn, (req, res) => {
   res.send('Hello!')
});

module.exports = app;
```

server.js

```
const app = require('./app');

const port = process.env.APP_PORT || 8080;
app.listen(port, () => {
   console.log(`Listening: http://localhost:${port}`);
});
```

api/index.js

```
const express = require("express");
const registerApi = require('./register');
const loginApi = require('./login')

const router = express.Router();

router.use(registerApi);
router.use(loginApi);

module.exports = router;
```

register.js

```
const express = require('express');
const User = require('../models/user');
const bcrypt = require('bcryptjs');
const { v4: uuidv4 } = require('uuid');
const nodemailer = require('nodemailer');

const transporter = require('../transporter/emailConfig');

const router = express.Router();

// Registration endpoint
router.post('/register', async (req, res) => {

    const { name, email, password } = req.body;

    // Check if password is 8 symbols long
    if (password.length < 8) {
        return res.status(400).json({ error: "Password should be at least 8 characters long" });
    }

    // Check if user already exists
    const userAlreadyExists = await User.findOne({ where: { email } }).catch((err) => {
        console.log("Error: ", err);
    });

    if(userAlreadyExists){
        return res.status(400).json({ message: "User with this email already exists"});
    }

    // Generate verification token
    const verificationToken = uuidv4();
    console.log(verificationToken);

    // Hash the password
    const hashedPassword = await bcrypt.hash(password, 10);

    // Insert user data and verification token into the database
    const newUser = new User({ name, email, password: hashedPassword, verification_token: verificationToken,
is_verified: false });
    const savedUser = await newUser.save().catch((err) => {
        console.log("Error: ", err);
        res.status(400).json({ error: "Can't register user at the moment..." });
    });

    if (savedUser) {

        const verificationLink = `http://localhost:8080/api/v1/verify/${verificationToken}`;

        // Send verification email
        const msg = {
            from: '"SweeMe Smart Home"',
            to: email,
            subject: 'Email Verification',
            html: `<p>Hello, ${name}! Thank you for registration! Please click the following link to verify your
email:</p> <a href="${verificationLink}">Verify Email</a>`
        };
```

```
        const info = await transporter.sendMail(msg);
        console.log("Preview URL: %s", nodemailer.getTestMessageUrl(info));

        res.status(200).json({ message:'User registered successfully! Email sent!' });
    }
});

router.get('/verify/:token', async (req, res) => {
    const { token } = req.params;

    try {

        const user = await User.findOne({ where: { verification_token: token } });

        if (!user) {
            return res.status(400).json({ message: 'Invalid or expired verification token' });
        }

        // Update the user's isVerified status to true
        user.is_verified = true;
        await user.save();

        return res.status(200).json({ message: 'Email verified successfully' });
    } catch (error) {
        console.error('Error verifying email:', error);
        return res.status(500).json({ message: 'Failed to verify email' });
    }
});

module.exports = router;
```

login.js

```
const express = require('express');
const User = require('../models/user');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

const router = express.Router();

// Authorization endpoint
router.post("/login", async (req, res) => {
    const { email, password } = req.body;

    try {
        // Check if email already exists
        const userWithEmail = await User.findOne({
            where: { email },
        });

        if (!userWithEmail) {
            return res.status(400).json({ message: "Email or password does not match..." });
        }

        if (!userWithEmail.is_verified) {
```

```
            return res.status(400).json({ message: "Please verify your email before logging in" });
        }

        if (userWithEmail.google_id) {
            // Handle the login flow for Google users
            const jwtToken = jwt.sign(
                {
                    id: userWithEmail.id,
                    email: userWithEmail.email,
                },
                process.env.JWT_SECRET,
                { expiresIn: process.env.JWT_EXP_VALUE }
            );

            return res.status(200).json({ message: "You logged in!", token: jwtToken });
        }
        const isPasswordMatch = await bcrypt.compare(password, userWithEmail.password);

        if (!isPasswordMatch) {
            return res.status(400).json({ message: "Email or password does not match..." });
        }

        // Generate JWT token
        const jwtToken = jwt.sign(
            {
                id: userWithEmail.id,
                email: userWithEmail.email,
            },
            process.env.JWT_SECRET,
            { expiresIn: process.env.JWT_EXP_VALUE }
        );

        res.status(200).json({ message: "You logged in!", token: jwtToken });
    } catch (error) {
        console.log("Error: ", error);
        res.status(500).json({ message: "An error occurred during login" });
    }
});

router.get('/decode-jwt', (req, res) => {
    const authHeader = req.headers.authorization;
    const splitedStr = authHeader.split(' ');
    const token = splitedStr[1];

    jwt.verify(token, process.env.JWT_SECRET, (err, decodedToken) => {
        if (err) {
            return res.status(500).json({ message: "Cannot decode JWT" });
        } else {
            const { id } = decodedToken;

            // Retrieve additional user information from the database based on the id
            User.findOne({ where: { id } })
                .then(user => {
                    if (!user) {
                        return res.status(404).json({ message: "User not found" });
                    }

                    // Combine the additional user information with the decoded token
```

```
                const userWithInfo = {
                    id: user.id,
                    role_id: user.role_id,
                    name: user.name,
                    email: user.email,
                };

                return res.status(200).json({ user: userWithInfo });
            })
            .catch(error => {
                console.error("Error retrieving user", error);
                return res.status(500).json({ message: "Failed to retrieve user" });
            });
    }
  });
});

module.exports = router;
```

passport.js

```
const passport = require("passport");
const passportJwt = require("passport-jwt");
const ExtractJwt = passportJwt.ExtractJwt;
const StrategyJwt = passportJwt.Strategy;
const User = require("../models/user");
const GoogleStrategy = require("passport-google-oauth20")

passport.use(
  new StrategyJwt({
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    secretOrKey: process.env.JWT_SECRET
  },
  function (jwtPayload, done) {
    return User.findOne({ where: { id: jwtPayload.id } })
      .then((user) => {
        return done(null, user);
      })
      .catch((err) => {
        return done(err);
      });
  }
  )
);

passport.use(
  new GoogleStrategy({
    clientID: process.env.GOOGLE_CLIENT_ID,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    callbackURL: "http://localhost:8080/api/v1/google/callback"
  },
    async (accessToken, refreshToken, profile, done) => {
      try {
        const { id, displayName, emails } = profile;

        // Check if the user already exists in the database
```

```
            const [user, created] = await User.findOrCreate({
                where: { google_id: id },
                defaults: {
                    name: profile.displayName,
                    email: profile.emails[0].value,
                    google_id: profile.id,
                    google_token: accessToken,
                },
            });

            if (!created) {
                return done(null, user);
            }

            user.is_verified = 1;
            await user.save();

            return done(null, user);
        } catch (error) {
            console.error('Error saving Google user:', error);
            return done(error, null);
        }
    }
));

passport.serializeUser(function (user, done) {
    done(null, user);
});

passport.deserializeUser(function (user, done) {
    done(null, user);
});
```

database/index.js

```
const {Sequelize} = require("sequelize");

const sequelize = new Sequelize({
    dialect: process.env.DATABASE_PROVIDER,
    host: process.env.DATABASE_HOST,
    port: process.env.DATABASE_PORT,
    username: process.env.DATABASE_USER,
    password: process.env.DATABASE_PASSWORD,
    database: process.env.DATABASE_NAME
});

(async () => {
    try {
        await sequelize.authenticate();
        console.log("Connected successfully");
        await sequelize.sync();
    }catch (error){
        console.error("Unable to connect to the database", error)
    }
})();
```

```
module.exports = sequelize;
```

authenticateToken.js

```
const jwt = require('jsonwebtoken');

const authenticateToken = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  const token = authHeader.split(' ')[1];

  jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
    if (err) {
      console.log(err);
      return res.status(401).json({ message: 'Unauthorized' });
    }

    req.user = decoded;
    next();
  });
};

module.exports = authenticateToken;
```

user-repository.js

```
class UserRepository {
  constructor() {
    this.users = [];
  }

  async findByEmail(email) {
    const user = this.users.find((user) => user.email === email);
    return user || null;
  }

  async save(user) {
    user.id = this.users.length + 1;
    this.users.push(user);
  }
}

module.exports = UserRepository;
```

emailConfig.js

```
const nodemailer = require("nodemailer");
```

```
const transporter = nodemailer.createTransport({
   host: "smtp.ethereal.email",
   port: 587,
   secure: false, // false for other ports, true for 465
   auth: {
      user: process.env.EMAIL_USER,
      pass: process.env.EMAIL_PASSWORD,
   }
});

module.exports = transporter;
```

models/index.js

```
'use strict';

const fs = require('fs');
const path = require('path');
const Sequelize = require('sequelize');
const process = require('process');
const {DataTypes} = require("sequelize");
const basename = path.basename(__filename);
const env = process.env.NODE_ENV || 'development';
const config = require(__dirname + '/../config/config.json')[env];
const db = {};

let sequelize;
if (config.use_env_variable) {
  sequelize = new Sequelize(process.env[config.use_env_variable], config);
} else {
  sequelize = new Sequelize(config.database, config.username, config.password, config);
}

fs
  .readdirSync(__dirname)
  .filter(file => {
    return (
      file.indexOf('.') !== 0 &&
      file !== basename &&
      file.slice(-3) === '.js' &&
      file.indexOf('.test.js') === -1
    );
  })

Object.keys(db).forEach(modelName => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

db.users = require('./user'); // db.tableName
db.user_settings = require('./userSettings');
db.devices = require('./device');
```

```
db.device_settings = require('./deviceSettings');
db.roles = require('./roles');
db.temperatures = require('./temperature');
db.humidities = require('./humidity');
db.ip_ddresses = require('./ipAddresses');
db.notifications = require('./notifications');
db.open_ai_logs = require('./openAiLogs');

db.sequelize.sync({ force: false })
   .then(() => {
     console.log('Re-sync done');
   });
module.exports = db;
```

user.js

```
const User = sequelize.define('User', {
        id: {
           type: DataTypes.INTEGER,
           primaryKey: true,
           autoIncrement: true
        },
        role_id: {
           type: DataTypes.INTEGER,
           allowNull: true,
           references: {
              model: 'roles',
              key: 'id',
           }
        },
        name: {
           type: DataTypes.STRING,
           allowNull: false
        },
        email: {
           type: DataTypes.STRING,
           allowNull: false,
           unique: true
        },
        password: {
           type: DataTypes.STRING,
           allowNull: true
        },
        restoration_token: {
           type: DataTypes.STRING,
           allowNull: true,
        },
        created_at: {
           type: DataTypes.DATE,
           allowNull: true,
        },
        updated_at: {
           type: DataTypes.DATE,
           allowNull: true,
        },
```

```
        verification_token: {
          type: DataTypes.STRING,
          allowNull: true,
        },
        is_verified: {
          type: DataTypes.BOOLEAN,
          defaultValue: false,
        },
        google_id: {
          type: DataTypes.STRING,
          allowNull: true,
          unique: true,
        },
        google_token: {
          type: DataTypes.STRING,
          allowNull: true,
        },
    }, {
        timestamps: true,
        underscored: true,
        tableName: 'users',
      }
    );

User.associate = function(models) {
    User.hasOne(models.UserSettings, { foreignKey: 'user_id',  onDelete: 'CASCADE'});
    User.hasMany(models.Notifications, { foreignKey: 'user_id', onDelete: 'CASCADE' });
    User.hasMany(models.DeviceSettings, { foreignKey: 'user_id', onDelete: 'CASCADE' });
    User.belongsTo(models.Roles, { foreignKey: 'role_id', onDelete: 'CASCADE' });
    User.hasMany(models.IpAddresses, { foreignKey: 'user_id', onDelete: 'CASCADE' });
};
module.exports = User;
```

userSettings.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const User = require('./user');

const UserSettings = sequelize.define('UserSettings', {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
        model: 'users',
        key: 'id',
      }
    },
    theme: {
      type: DataTypes.BOOLEAN,
```

```
      allowNull: false
    },
    created_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
    updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'user_settings',
    },
);

UserSettings.associate = function(models) {
    UserSettings.belongsTo(models.User, { foreignKey: 'user_id', onDelete: 'CASCADE' });
};
module.exports = UserSettings;
```

device.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const User = require('./user');
const DeviceSettings = require('./deviceSettings');
const Temperature = require('./temperature');
const Humidity = require('./humidity');

const Device = sequelize.define('Device', {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    type: {
      type: DataTypes.ENUM('1', '2'),
      allowNull: false,
    },
    mpn: {
      type: DataTypes.STRING,
      allowNull: false
    },
    created_at: {
      type: DataTypes.DATE,
```

```
      allowNull: true,
    },
    updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'devices',
    indexes: [
      { fields: ['user_id'], name: 'user_id_index', using: "BTREE" }
    ]
});

Device.associate = function(models) {
    Device.hasMany(models.DeviceSettings, {foreignKey: 'device_id', onDelete: 'CASCADE'});
    Device.hasMany(models.Temperature, { foreignKey: 'device_id', onDelete: 'CASCADE' });
    Device.hasMany(models.Humidity, { foreignKey: 'device_id', onDelete: 'CASCADE' });
};
module.exports = Device;
```

deviceSettings.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const User = require('./user');
const Device = require('./device');

const DeviceSettings = sequelize.define('DeviceSettings', {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
        model: 'users',
        key: 'id',
      }
    },
    device_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
        model: 'devices',
        key: 'id'
      }
    },
    delay: {
      type: DataTypes.INTEGER,
      allowNull: false,
    },
```

```
    created_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
    updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'device_settings',
});

DeviceSettings.associate = function(models) {
    DeviceSettings.belongsTo(models.User, {foreignKey: 'user_id', onDelete: 'CASCADE'});
    DeviceSettings.belongsTo(models.Device, {foreignKey: 'device_id', onDelete: 'CASCADE' });
};
module.exports = DeviceSettings;
```

roles.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const User = require('./user');

const Roles = sequelize.define('Roles', {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    created_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
    updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
    },
    }, {
      timestamps: true,
      underscored: true,
      tableName: 'roles',
    });
Roles.associate = function(models) {
    Roles.hasMany(models.User, { foreignKey: 'role_id', onDelete: 'CASCADE' });
};
module.exports = Roles;
```

```
const { DataTypes } = require('sequelize');
const sequelize = require('../database');
const User = require('./user');
const OpenAiLogs = require('./openAiLogs');

const Notifications = sequelize.define('Notifications', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true,
  },
  user_id: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: {
      model: 'users',
      key: 'id',
    }
  },
  message: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  open_ai_log_id: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: {
      model: 'open_ai_logs',
      key: 'id',
    }
  },
  type: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  created_at: {
    type: DataTypes.DATE,
    allowNull: true,
  },
  updated_at: {
```

```
        type: DataTypes.DATE,
        allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'notifications',
});

Notifications.associate = function (models) {
    Notifications.belongsTo(models.User, { foreignKey: 'user_id', onDelete: 'CASCADE' });
    Notifications.belongsTo(models.OpenAiLogs, { foreignKey: 'open_ai_log_id' });
};
module.exports = Notifications;
```

openAiLogs.js

```
const {DataTypes, Op} = require('sequelize');
const sequelize = require('../database');
const Notifications = require('./notifications');

const OpenAiLogs = sequelize.define('OpenAiLogs', {
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
    },
    request: {
        type: DataTypes.TEXT,
        allowNull: false,
    },
    response: {
        type: DataTypes.TEXT,
        allowNull: false,
    },
    created_at: {
        type: DataTypes.DATE,
        allowNull: true,
    },
    updated_at: {
        type: DataTypes.DATE,
        allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'open_ai_logs',
});

OpenAiLogs.associate = function (models) {
    OpenAiLogs.hasOne(models.Notifications, { foreignKey: 'open_ai_log_id', onDelete: 'CASCADE' });
```

```
};
module.exports = OpenAiLogs;
```

ipAddresses.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const User = require('./user');

const IpAddresses = sequelize.define('IpAddresses', {
   id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
   },
   user_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
         model: 'users',
         key: 'id',
      }
   },
   ip: {
      type: DataTypes.STRING,
      allowNull: false,
   },
   user_agent: {
     type: DataTypes.STRING,
     allowNull: false,
   },
   created_at: {
      type: DataTypes.DATE,
      allowNull: true,
   },
   updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
   },
}, {
   timestamps: true,
   underscored: true,
   tableName: 'ip_addresses',
});

IpAddresses.associate = function(models) {
   IpAddresses.belongsTo(models.User, {foreignKey: 'user_id', onDelete: 'CASCADE'});
};
module.exports = IpAddresses;
```

humidity.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const Device = require('./device');

const Humidity = sequelize.define('Humidity', {
   id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
   },
   device_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: {
         model: 'devices',
         key: 'id',
      }
   },
   humidity: {
      type: DataTypes.INTEGER,
      allowNull: false,
   },
   created_at: {
      type: DataTypes.DATE,
      allowNull: true,
   },
   updated_at: {
      type: DataTypes.DATE,
      allowNull: true,
   },
}, {
   timestamps: true,
   underscored: true,
   tableName: 'humidities',
});

Humidity.associate = function(models) {
   Humidity.belongsTo(models.Device, { foreignKey: 'device_id', onDelete: 'CASCADE' });
};
module.exports = Humidity;
```

temperature.js

```
const {DataTypes} = require('sequelize');
const sequelize = require('../database');
const Device = require('./device');

const Temperature = sequelize.define('Temperature', {
   id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
   },
   device_id: {
      type: DataTypes.INTEGER,
```

```
        allowNull: false,
        references: {
            model: 'devices',
            key: 'id',
        }
    },
    temperature: {
        type: DataTypes.FLOAT,
        allowNull: false,
    },
    created_at: {
        type: DataTypes.DATE,
        allowNull: true,
    },
    updated_at: {
        type: DataTypes.DATE,
        allowNull: true,
    },
}, {
    timestamps: true,
    underscored: true,
    tableName: 'temperatures',
});
Temperature.associate = function(models) {
    Temperature.belongsTo(models.Device, { foreignKey: 'device_id', onDelete: 'CASCADE' });
};

module.exports = Temperature;
```