МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О. Є.

«_ _»_____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
"МАГІСТР"

Тема: Мобільний додаток дистанційного керування електроприладами

Виконавець: Шевченко В.В.

Керівник: Кучеров Д.П.

Нормоконтролер: Тупота Є.В.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютеризованих систем управління

Спеціальність (спеціалізація) 123 «Комп'ютерна інженерія»

(шифр, найменування)

ЗАТВЕРДЖУЮ

Завідувач кафедри

Литвиненко О.Є.

«_____» _____2020 р.

ЗАВДАННЯ

на виконання дипломної роботи

Шевченка Володимира Вячеславовича

(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема дипломної роботи: Мобільний додаток дистанційного керування електроприладами
затверджена наказом ректора від « 27 » серпня 2020 р. № 1203
2. Термін виконання роботи: з 5 жовтня 2020 р. по 13 грудня 2020 р.
3. Вихідні дані до роботи: технічна документація, тестові дані, програмні продукти

4. Зміст пояснювальної записки:
1) Аналіз існуючих додатків для дистанційного керування електроприладами
2) Функціональні особливості та алгоритми роботи веб-додатку для керування електроприладами
3) Тестування та результати функціонування веб-додатку для дистанційного керування електроприладами

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
1) Схема алгоритму роботи системи
2) Діаграма прецедентів додатку дистанційного керування електроприладами
3) Діаграма послідовності додатку дистанційного керування електроприладами
4) Панель авторизації веб-додатку
5) Панель приладів веб-додатку

6) Панель зміни функцій приладу веб-додатку

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати літературу	05.10.2020 – 09.10.2020	
2	Аналіз існуючих систем для дистанційного керування електроприладами	10.10.2020 – 20.10.2020	
3	Визначити технології для створення системи за темою дипломного проекту	21.10.2020 – 09.11.2020	
4	Завантажити усі необхідні програмні компоненти для розробки, тестування додатку	10.11.2020 – 14.11.2020	
5	Створити алгоритм роботи додатку, а також необхідні *UML*-діаграми для проектування системи	15.11.2020 – 20.11.2020	
6	Розробити веб-додаток для дистанційного керування електроприладами	21.11.2020 – 01.12.2020	
7	Протестувати створений функціонал	02.12.2020 – 04.12.2020	
8	Оформити пояснювальну записку	05.12.2020 – 10.12.2020	
9	Оформити графічний та ілюстративний матеріал	10.12.2020 – 13.12.2020	

7. Дата видачі завдання: «05» жовтня 2020 р.

Керівник дипломної роботи (проекту) _____ Кучеров Д.П.
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Шевченко В.В.
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Мобільний додаток дистанційного керування електроприладами»: 87 сторінок, 34 рисунків, 1 таблиця, 21 використаних джерел.

ДИСТАНЦІЙНЕ КЕРУВАННЯ ЕЛЕКТРОПРИЛАДАМИ, ВЕБ-ДОДАТОК, ТЕХНОЛОГІЇ КЕРУВАННЯ ЕЛЕКТРОПРИЛАДАМИ, АЛГОРИТМ РОБОТИ ДОДАТКУ, *UML*-ДІАГРАМИ ДЛЯ ПРОЕКТУВАННЯ ДОДАТКУ.

Об’єкт дослідження дипломної роботи – технології дистанційного керування електроприладами.

Предмет дослідження дипломної роботи – комп’ютерні системи та додатки для дистанційного керування електроприладами.

Мета дипломної роботи – створити зручний для кінцевого користувача веб-додаток, на базі найсучасніших технологій, для домашньої автоматизації.

Методи дослідження – методи проектування, розробки та тестування сучасних систем, перегляд наявних аналогів, перегляд наявних технологій для систем автоматизації керування електроприладами.

Здійснено аналіз існуючих додатків і систем для керування електроприладами; досліджено методи для ефективної розробки та тестування систем; досліджено сучасні технології автоматизації власного дому; реалізовано алгоритми та *UML*-діаграми для веб-додатку дистанційного керування електроприладами; проаналізовано та протестовано результати та зручність роботи створеної системи.

Матеріали дипломної роботи рекомендується використовувати для розвитку технологій автоматизації та ефективнішого керування системами власного дому. .

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ШІ – штучний інтелект

СУБД – Система управління базами даних (комплекс програмного забезпечення, що надає можливості створення, збереження, оновлення та пошуку інформації в базах даних)

API – Application Programming Interface (прикладний програмний інтерфейс)

JSON – JavaScript Object Notation (текстовий формат обміну даними між комп'ютерами)

Laravel – популярний *PHP*-фреймворк з відкритим кодом

REST – Representational State Transfer (архітектурний стиль взаємодії компонентів розподіленого додатка в мережі)

DOM – Document Object Model (об’єктна модуль документа)

MVC – Model-View-Controller (модель-представлення-контролер)

Composer – менеджер залежностей для *PHP*

React–JavaScript – бібліотека з відкритим кодом для розробки користувацьких інтерфейсів

GUI – Graphical User Interface (графічний інтерфейс користувача)

Mysql – реляційна база даних

Docker – програмне забезпечення для автоматизації розробки та управління додатками в середовищах з підтримкою контейнеризації

PHPUnit – це спеціальний фреймворк, призначений для модульного тестування скриптів мови *PHP*

ВСТУП

Одним з найактуальніших питань сьогодні є поліпшення якості, безпеки та зручності життя. Розробники в *IT* – компаніях кожного дня працюють над тим, щоб якомога більше автоматизувати процеси, що відбуваються навколо та з користувачами. Зараз завдяки сучасним технологіям, майже кожна людина може отримати доступ до бажаної інформації або додатку. У мережі Інтернет доступно безліч різноманітних технологій, звісно не всі з них безкоштовні, та за якість завжди доводиться платити. Шаленої популярності нині набувають так звані “Розумні девайси”, а в сукупності це має назву “Розумний дім”. Розумні пристрої – це всі повсякденні об'єкти, зроблені інтелектуальними за допомогою вдосконалених обчислень, включаючи ШІ та машинне навчання, та об'єднані в мережу для формування Інтернету речей (IoT). Основне питання як звичайному користувачу взаємодіяти з такими девайсами. Відповідь на це питання може бути абсолютно різноманітною, ця взаємодія може бути влаштована через веб-додаток, деякі розумні пристрої реагують на рух, деякі на голос користувача.

Технологія “Розумний дім” відноситься до зручного налаштування будинку, де приладами та пристроями можна автоматично дистанційно керувати з будь-якого місця за допомогою підключення до Інтернету та мобільного або іншого мережевого пристрою через веб-застосунок. Пристрої в розумному будинку взаємопов’язані через Інтернет, що дозволяє користувачеві дистанційно керувати такими функціями, як безпечний доступ до дому, температура, освітлення, домашній кінотеатр, тощо.

Веб-програма – це комп’ютерна програма, яка використовує веб-браузер для виконання певної функції. Її також називають веб-додатком. Веб-програми є на багатьох веб-сайтах. Застосунок може бути таким же простим, як дошка оголошень або контактна форма на веб-сайті, або настільки ж складний, як текстовий процесор або багатокористувацький мобільний ігровий додаток, який користувач завантажуєте на телефон. Веб-програма – це програма типу клієнт-сервер. Це означає, що додаток має сторону клієнта та сторону сервера. Термін "клієнт" тут

стосується програми, яку людина використовує. Це частина середовища клієнт-сервер, де багато комп’ютерів обмінюються інформацією. Наприклад, у випадку з базою даних клієнтом є програма, за допомогою якої користувач вводить дані. Сервер – це програма, яка зберігає інформацію.

Взаємодію між сервером та клієнтом можна організувати декількома способами. Однією з найпоширеніших у цьому контексті є технологія *Restful API. RESTful API* – це архітектурний стиль інтерфейсу прикладних програм *(API),* який використовує *HTTP*-запити для доступу та використання даних. Ці дані можуть бути використані для отримання, зберігання, створення та видалення типів даних, що стосуються ресурсів. *RESTful API* – також званий *RESTful* веб-сервісом або *REST API* – заснований на репрезентативній передачі стану (*REST*), що є архітектурним стилем та підходом до комунікацій, часто використовуваних при розробці веб-застосунків.

Для створення веб-додатку керування електронними пристроями буде використана трирівнева клієнт-серверна архітектура. Щоб зрозуміти принцип, варто згадати основні елементи такої архітектури. Це архітектурна модель програмного комплексу, яка передбачає наявність в ній трьох компонентів: клієнта, серверного додатку (до якого підключено клієнтський додаток) та сервера бази даних (з яким працює сервер додатків). Зв'язок відбувається за допомогою мережі.

Зі сторони сервера буде використовуватися популярний фреймворк *Laravel*. Дана технологія містить у собі дуже багато практичних та зручних рішень, має якісну документацію та величезну популярність в усьому світі. Даний фреймворк використовує такий паттерн проектування, як *MVC* (*Model-View-Controller*), що поліпшує розробку та читання коду.

Для клієнтської сторони добре підходить прогресивний *JavaScript*-фреймворк – *React Native*. Перевага *React Native* полягає в тому, що розробка ведеться на базі добре відомої бібліотеки *React*, але при цьому відображення додатків відбувається так, як ніби вони були розроблені природно для кожної платформи. Термін підготовки прототипу додатка виявляється набагато нижчим.

Не потрібно вести окрему розробку для *iOS* і *Android*. Подальша підтримка і розвиток програми відбувається простіше.

Для серверу бази даних добре підходить *SQL* СУБД – *Mysql*. Дана технологія підходить для невеликих мало та середньо-навантажених додатків. У *Laravel* є зручна технологія *ORM Eloquent* для взаємодії з *mysql*.

Говорячи про технологію розумний будинок, слід зазначити, що на сьогоднішній день вона являється однією з найпопулярніших у світі. Багато сучасних великих компаній прагнуть розробляти свою продукцію у даному напрямку, адже зручність – це головний пріоритет для користувача.

Не дивлячись на те, що вже існує велика кількість веб-додатків для контролю електротехніки, у даному проекті буде створено універсальну систему для приладів, що розроблені на основі технології *Z-Wave*. Так у застосунку буде можливість зареєструватися, отримати список усіх приладів у мережі, а також список їх функцій та керування ними. Також ця система буду мультиплатформенною, що дозволяє використовувати її на всіх пристроях – починаючи від персональних комп'ютерів і закінчуючи мобільними пристроями, та під управлінням різних операційних систем.

Метою роботи є проектування та розробка веб-додатку для керування електронними приладами на базі трирівневої архітектури.

Методами дослідження є методи проектування, розробки та тестування сучасних систем, перегляд наявних аналогів, перегляд наявних технологій для систем автоматизації керування електроприладами.

Об'єктом дослідження є технології дистанційного керування електроприладами, веб-додаток для керування електронними приладами, який побудований на базі трирівневої архітектури з використанням технологій *React Native, Laravel* та *Z-Wave, REST API, mysql*.

Предметом дослідження являється аналіз вже існуючих веб-додатків та систем для керування електронними приладами, виявлення їх недоліків та розробка власного програмного продукту з їх усуненням.

Результатом роботи є програмне забезпечення, а саме веб-додаток, що дозволяє зареєструватися, отримати список власних приладів на базі технології *Z-Wave*, отримати список функцій прилада та застосувати їх.

Запропоноване рішення веб-додатку відрізняється від вже існуючих тим, що є абсолютно безкоштовним, може бути використане на будь-якій платформі, та є відносно дешевим у своїй реалізації .

РОЗДІЛ 1
ДОСЛІДЖЕННЯ СИСТЕМ “*SMART HOUSE*” ТА ДОДАТКІВ ДЛЯ КЕРУВАННЯ НИМИ

1.1. Поняття “*Smart House*” системи

Розумний дім – це резиденція, яка використовує підключені до Інтернету пристрої для віддаленого моніторингу та управління приладами та системами, такими як освітлення та опалення.

Технологія розумного будинку, яку також часто називають домашньою автоматизацією або домотикою (від латинського "*domus*" означає будинок), забезпечує власникам будинків безпеку, комфорт, зручність та енергоефективність, дозволяючи їм керувати розумними пристроями, часто за допомогою програми розумного будинку на своєму смартфон або іншому мережевому пристрої. Частина Інтернету речей (*IoT*), розумні будинкові системи та пристрої часто працюють разом, обмінюючись споживчими даними про споживання та автоматизуючи дії на основі уподобань власників будинків.

Майже у кожному аспекті життя, коли технологія потрапила у домашній простір (лампочки, посудомийні машини тощо), з'явилася інтелектуальна альтернатива у розрізі технологій розумного будинку:

– Смарт-телевізори, що підключаються до Інтернету, щоб отримати доступ до вмісту через програми, такі як відео на замовлення та музика. Деякі смарт-телевізори також мають розпізнавання голосу або жестів;

– Інтелектуальні системи освітлення, такі як *Hue* від *Philips Lighting Holding B.V.*, окрім можливості дистанційного керування, можуть виявляти перебування мешканців у приміщенні та регулювати освітлення за потреби. Розумні лампочки також можуть регулювати себе залежно від наявності денного світла.

– Розумні термостати, такі як *Nest* від *Nest Labs Inc*., мають вбудований *Wi-Fi*, що дозволяє користувачам планувати, контролювати та дистанційно контролювати домашню температуру. Ці пристрої також вивчають поведінку власників будинків та автоматично змінюють налаштування, щоб забезпечити мешканцям максимальний комфорт та ефективність. Розумні термостати також можуть повідомляти про споживання енергії та нагадувати користувачам, серед іншого, про заміну фільтрів;

– Користуючись розумними замками та відкривачами гаражних дверей, користувачі можуть надати або заборонити доступ відвідувачам. Розумні замки також можуть виявити, коли мешканці знаходяться поруч, і розблокувати двері для них;

– За допомогою розумних камер безпеки мешканці можуть стежити за своїми будинками, коли вони відсутні або перебувають у відпустці. Розумні датчики руху також можуть виявити різницю між мешканцями, відвідувачами, домашніми тваринами та грабіжниками, а також можуть повідомляти органи влади про підозрілу поведінку;

– Догляд за домашніми тваринами можна автоматизувати за допомогою підключених годівниць. Поливати кімнатні рослини та газони можна за допомогою підключених таймерів;

– Монітори побутових систем можуть, наприклад, відчути струм напруги та вимкнути електроприлади, або виявити збій води чи замерзання труб, і перекрити воду, щоб підвал, наприклад, не затопив.

Рис. 1.1. Розумний дім

Однією з найбільш розрекламованих переваг домашньої автоматизації є забезпечення спокою власникам будинків, що дозволяє їм дистанційно стежити за своїми будинками, протидіючи таким небезпекам, як забута кавоварка, або залишені незамкненими вхідні двері.

Домотика також корисна для людей похилого віку, забезпечуючи моніторинг, який може допомогти їм залишатися вдома комфортно та безпечно, а не переїжджати до будинку престарілих або вимагати постійного догляду вдома.

Розумні будинки можуть швидко підлаштовуватися під уподобаннями користувачів. Наприклад, користувач може запрограмувати відкрити свої гаражні ворота, увімкнути світло, увімкнути камін та відтворити свої улюблені мелодії.

Домашня автоматизація також допомагає споживачам підвищити ефективність. Замість того, щоб залишати кондиціонер увімкненим протягом усього дня, розумна домашня система може навчитися поведінці та переконатися, що будинок охолоне до моменту прибуття власників дому з роботи. Те саме стосується побутової техніки. Завдяки розумній зрошувальній системі газон буде

поливатися лише за необхідності. При домашній автоматизації енергія, вода та інші ресурси використовуються ефективніше, що допомагає економити як природні ресурси, так і гроші для споживача.

Недоліком розумних будинків є сприймана ними складність; деякі люди відчувають труднощі з технологією або відмовляться від неї з першими незручностями. Виробники розумних будинків та альянси працюють над зменшенням складності та покращенням користувальницького досвіду, щоб зробити його приємним і корисним для користувачів усіх типів та технічного рівня.

Щоб системи домашньої автоматизації були справді ефективними, пристрої повинні бути сумісними незалежно від виробника та використовувати один і той же протокол або, принаймні, додаткові. Оскільки це відносно новий ринок, ще не існує золотого стандарту для автоматизації будинків. Однак стандартні альянси співпрацюють з виробниками та протоколами, щоб забезпечити взаємодію та безперебійну роботу систем для користувачів.

Ще одна важлива проблема – це розумна домашня безпека. Звіт *NTT Data Corp.* за 2016 рік виявив, що 80% американських споживачів стурбовані безпекою даних своїх розумних будинків. Якщо хакери зможуть проникнути на розумний пристрій, вони потенційно можуть вимкнути світло, сигналізацію та розблокувати двері. Крім того, хакери потенційно можуть отримати доступ до мережі власників будинків, що призведе до вилучення даних.

На додаток до домашньої безпеки, багатьох людей, що є противниками розумних домашніх систем турбує конфіденційність даних. У звіті *NTT Data* виявлено, що 73% споживачів стурбовані конфіденційністю даних, якими користуються їхні розумні домашні пристрої. Хоча виробники розумних домашніх пристроїв та платформ можуть збирати споживчі дані для кращого адаптування своїх продуктів або пропонувати нові та вдосконалені послуги для споживачів, довіра та прозорість мають вирішальне значення для виробників, які прагнуть залучити нових клієнтів.

Новозбудовані будинки часто проектуються з інфраструктурою розумного дому. З іншого боку, старі будинки можна модернізувати за допомогою розумних

технологій. Хоча багато інтелектуальних домашніх систем все ще працюють на *X10* або *Insteon*, популярність *Bluetooth* і *Wi-Fi* зросли.

Zigbee та *Z-Wave* – два найпоширеніші протоколи зв'язку домашньої автоматизації, що нині популярні. Обидва використовують сітчасті мережеві технології, радіосигнали малої потужності для підключення розумних домашніх систем. Хоча обидва націлені на одні й ті самі розумні домашні програми, *Z-Wave* має радіус дії 30 метрів, *Zigbee* – до 10 метрів, причому *Zigbee* часто сприймається як більш складний з двох. Чіпи *Zigbee* доступні у багатьох компаній, тоді як чіпи *Z-Wave* доступні лише від *Sigma Designs*.

Розумний дім – це не сукупність різнорідних інтелектуальних пристроїв та приладів, а ті, що працюють разом, щоб створити дистанційно керовану мережу. Всі пристрої управляються головним контролером домашньої автоматизації, який часто називають розумним домашнім концентратором. Розумний домашній концентратор – це апаратний пристрій, який виступає центральною точкою системи розумного будинку і здатний сприймати, обробляти дані та здійснювати бездротовий зв'язок. Він поєднує всі різні програми в єдину розумну програму для дому, якою можуть керувати власники будинків віддалено. Прикладами розумних домашніх концентраторів є *Amazon Echo, Google Home, Insteon Hub Pro, Samsung SmartThings* та *Wink Hub.*

У простих сценаріях розумного будинку події можуть бути синхронізовані або активовані. Заплановані події базуються на годиннику, наприклад, опускання штор о 18:00, тоді як активовані події залежать від дій в автоматизованій системі; наприклад, коли власник зі смартфоном наближається до дверей, розумний замок розблоковується, а розумні ліхтарі вмикаються.

Машинне навчання та штучний інтелект (ШІ) стають все більш популярними в розумних домашніх системах, що дозволяє програмам домашньої автоматизації адаптуватися до свого середовища. Наприклад, голосові системи, такі як *Amazon Echo* або *Google Home*, містять віртуальних помічників, які вивчають та персоналізують розумний будинок відповідно до уподобань та шаблонів мешканців.

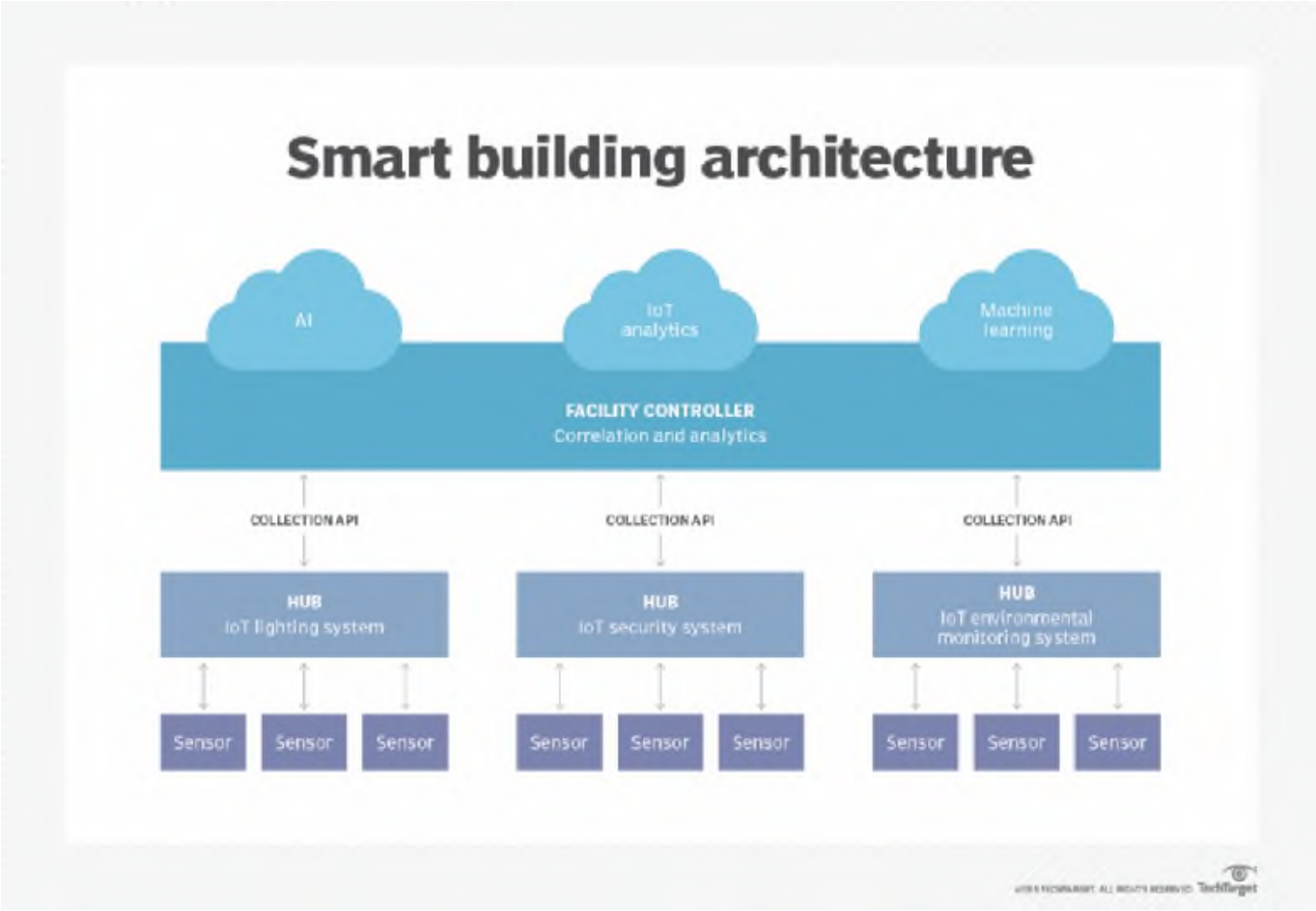

Рис. 1.2. Архітектура розумного будинку

1.2. Розгляд додатків для керування розумним будинком

Оскільки більшістю розумних пристроїв керують програми для смартфонів та планшетів, варто ближче розглянути програмне забезпечення, завдяки якому так зручно та ефективно для користувача розумний дім працює разом.

Хоча багато розумних домашніх пристроїв, особливо концентраторів та інших продуктів для підключення, поєднуються зі своїми власними програмами, існує велика кількість програм, які працюють незалежно для підключення та управління пристроями.

Деякі програми, програми працюють лише для певних пристроїв, інші є більш універсальними. Хоча багато з цих програм виконують подібні функції, наприклад, повідомляють вас про важливі події вдома, або віддалено включають

світло або побутові прилади, є деякі важливі відхилення з дуже специфічними функціями.

Незалежно від того, чи ваша розумна домашня екосистема працює як єдине ціле, або у домі використовується лише кілька розумних девайсів, надійний додаток для управління буде клеєм, який пов’язує все це в одне ціле враження.

Серед найкращих додатків для керування розумним домом, слід відокремити такі:

Amazon Alexa. Завдяки понад 20 000 інтеграцій сторонніх розробників, *Alexa*, безсумнівно, є однією з найповніших екосистем розумного будинку, доступних на сьогодні.

Alexa полегшує доступ до кожного аспекту вашого розумного будинку та керування ним. Користувач може використовувати *Alexa*, щоб говорити з такими програмами, як *Spotify*, так само легко, як і він можете попросити її вимкнути світло. Оскільки екосистема *Amazon* є однією з найпоширеніших у цій галузі, більшість розумних продуктів легко інтегруються з *Alexa*, включаючи продукти *Philips, Samsung, Nest* та *Schlage*, тобто *Alexa* тепер може закрити ваш гараж, заблокувати двері та налаштувати температура вашого будинку.

Зрештою, здатність *Alexa* інтегруватися та спілкуватися з більшістю інших інтелектуальних пристроїв та додатків робить її одним із найкращих варіантів для розумного будинку.

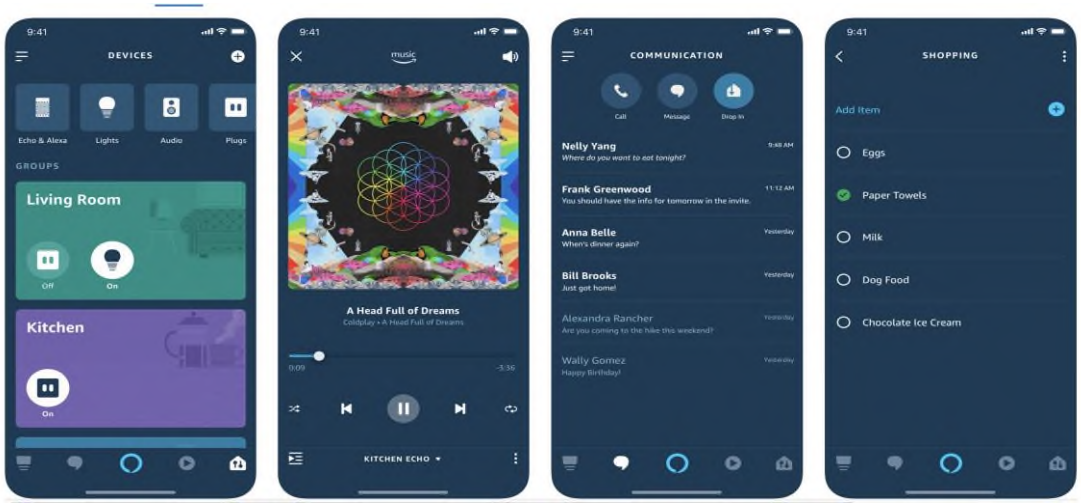

Рис. 1.3. *Alexa app*

Google Assistant. Незважаючи на те, що *Google Assistant* має менше сторонніх інтеграцій, він часто може відповідати на запитання та виконувати команди, яких *Alexa* не може, завдяки основному володінню *Google* простором пошукової системи. Дослідження цифрового агентства *Dentsu* 360*i* показало, що *Google Assistant* у п'ять разів частіше дасть правильну відповідь, ніж *Alexa*. Зрештою, Асистент виграє, коли справа доходить до розуміння того, як люди говорять природно.

Наприклад, якщо споживач скаже *Assistance* «Мені не подобається ця пісня» на *Spotify*, вона перейде до наступної, тоді як *Alexa* просто скаже вам: «Великі пальці вгору та вниз не підтримуються *Spotify»*.

Помічник також може інтегруватися з продуктами більшості основних брендів, включаючи *Philips, Belkin, August, Nest* та популярними програмами, такими як *Spotify* та *Uber*.

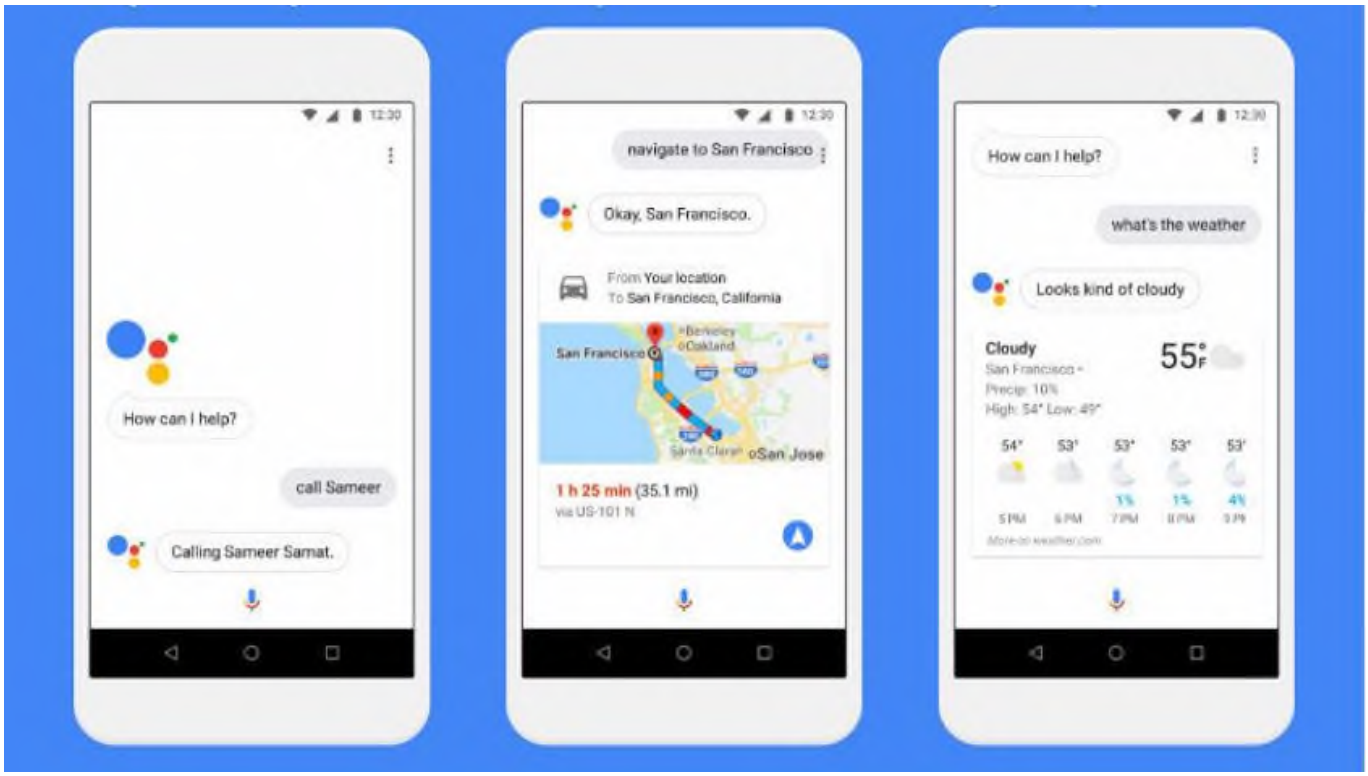

Рис. 1.4. *Google Assistant app*

Samsung SmartThings Hub. Ця система *Samsung* об'єднує широкий спектр інтелектуальних пристроїв різних марок, включаючи розумні термостати, маршрутизатор *Wi-Fi*, лампочки та пристрої безпеки. Користувачі інтелектуальної

системи отримують настінне кріплення *SmartThings Hub* і можуть отримати повний доступ до інтелектуальних пристроїв, підключених до концентратора за допомогою програми *SmartThings IOS* або *Android*.

Протягом останніх кількох років *SmartThings Hub* від *Samsung* розширює власні пропозиції смарт-пристроїв. Насправді лінійка продуктів *Hub* нещодавно зросла завдяки додаванню штекера *Wi-Fi SmartThing*, лампочки *SmartThings* та камери *SmartThings*.

За допомогою *SmartThings Hub* користувач також може встановити сумісні інтелектуальні пристрої для здійснення різних дій, таких як вмикання та вимикання, коли людина входить в кімнату або виходить з неї. Хоча споживач може передавати голосові команди настінному концентратору або додатку *SmartThings*, також існує можливість інтеграції в систему *Amazon Alexa* або *Google Assistant.*

Рис. 1.5. *Samsung SmartThings Hub app*

Apple Home Kit App. *Apple Home Kit* – це, мабуть, одна з найповніших систем автоматизації розумних будинків на ринку, і додаток для автоматизації дому *ios* є

ідеальним супутником для неї. Він призначений для роботи як з розумними домашніми пристроями *Apple*, так і з іншими розумними домашніми пристроями.

Додаток для домашнього комплекту *Apple* можна використовувати з будь-якого *apple*-девайсу, будь то *iPhone, iPad, MacBook*. Додаток постачається з інформаційною панеллю розумного будинку, яка полегшує роботу з усіма аспектами вашого дому.

Додаток дозволяє користувачам створювати "сценарії", що дозволяє виконувати кілька дій одним натисканням на смартфоні. Подібно до сценарію виходу з дому все, що вам потрібно зробити, – це натиснути один раз на смартфон, щоб переконатися, що штори закриті по всьому будинку, вимкнене світло та термостат встановлений в енергоефективний режим. На даний момент існує більше сотні брендів із пристроями, що підтримуються *HomeKit*.

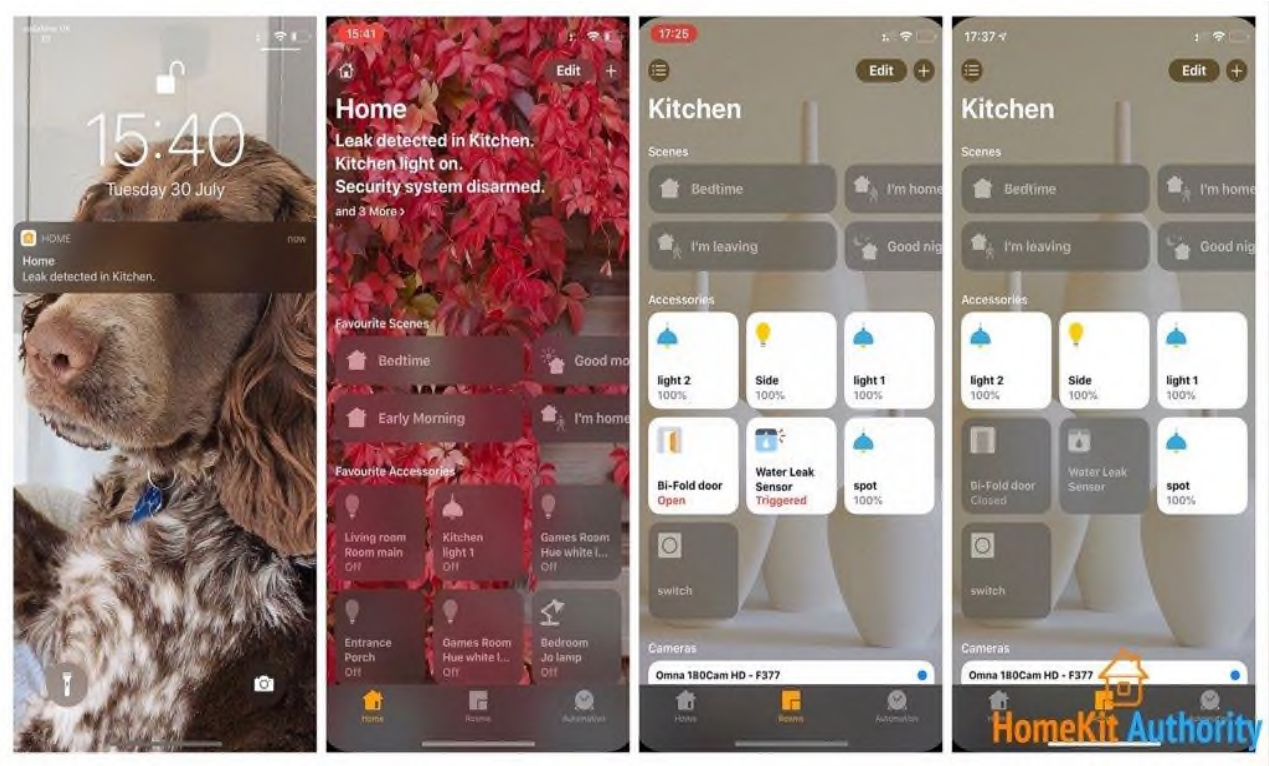

Рис. 1.6. *Apple Home Kit App*

Honeywell Home app. Honeywell Home пропонує безліч розумних домашніх пристроїв, які варіюються від домашньої безпеки до очищувача повітря. А домашня програма *Honeywell* дозволяє зручно керувати всіма девайсами у вашому смартфоні.

Компанія *Honeywell* має ряд пристроїв, які включають термостати, охоронну камеру, витік води та систему запобігання замерзанню. За допомогою програми юзер може керувати та змінювати налаштування пристроїв, а також отримувати сповіщення в реальному часі від домашніх пристроїв *IoT*, щоб повідомити вас про події у вашому домі в реальному часі.

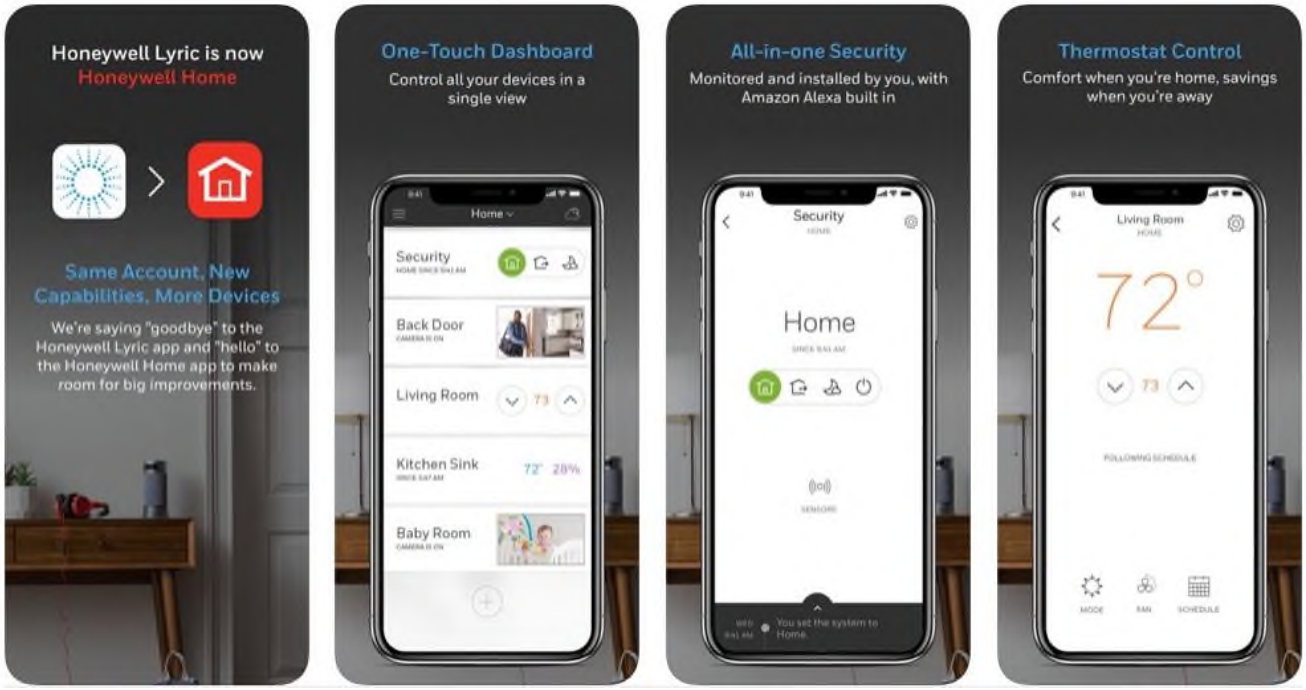

Рис. 1.7. *Honeywell Home app*

1.3. Технологія *Z-Wave*

На сьогоднішній день найбільш надійна і зручна бездротова технологія для будівництва розумного будинку – технологія *Z-Wave*.

Технологія *Z-Wave* розвивається і підтримується компанією *Sigma Design*, відомим виробником напівпровідникових пристроїв і мікросхем. *Z-Wave* – це набір пропрієтарних протоколів фізичного (що визначають на якій частоті працює мережа, рівень сигналу, що передається, його модуляцію) і логічного (такі протоколи задають адресацію пристроїв, набір команд, послідовність обміну інформацією) рівня, реалізованих в декількох наборах мікросхем *Sigma Design*.

«Пропрієтарна» – в даному випадку означає, що вся документація, детально описує протоколи – закрита, а кожен виробник, який хоче розробляти пристрої,

керовані за технологією *Z-Wave* повинен підписати угоду про нерозголошення деталей протоколу. Безумовно, такий підхід накладає певні обмеження, але в той же час є і важлива перевага – абсолютно всі пристрої, з підтримкою *Z-Wave* гарантовано сумісні між собою. Тобто, ким би не був випущений датчик або центральний контролер, він майже напевно буде працювати з будь-яким іншим пристроєм, який орієнтовано на роботу в мережі *Z-Wave*. «Майже» – тому що в кількох випадках, виробники можуть програмно обмежити підтримку до продуктів тільки одного бренду.

Мережа *Z-Wave* на фізичному рівні організована за принципом *mesh*-мережі. *Mesh*, в перекладі з англійської означає «сітка, стільники». У звичних нам мережах, наприклад, мережах *Wi-Fi*, кожен пристрій, що працює в мережі має бути безпосередньо пов'язаним з центральним контролером (в разі *Wi-Fi* – з *Wi-Fi*-роутером) – такий спосіб організації мережі зазвичай називають «зіркою». Якщо рівень сигналу контролера або кінцевого пристрою занадто малий, щоб забезпечити прямий зв'язок, значить такий пристрій підключити буде неможливо.

У мережах *Mesh* – кожен компонент мережі завжди виступає в якості своєрідного ретранслятора сигналу – це означає, що навіть якщо якийсь датчик або актуатор не може безпосередньо підключитися до хабу «розумного будинку» через слабкий сигнал (а саме такий рівень сигналу передбачається до використання в бездротовій мережі «розумного будинку»), але «бачить» якийсь інший датчик або компонент мережі *Z-Wave* – він може бути підключений до мережі і центральний контролер зможе ним керувати. Таким чином, незважаючи на невеликі значення потужності радіосигналу в мережі *Z-Wave* – вона може покривати значні відстані і масштабні об'єкти (наприклад, багатоповерхові будівлі).

Рис. 1.8. Технологія *Z-Wave*

Серед інших важливих переваг технології *Z-Wave* можна відзначити такі:

– Легке підключення в мережу. Для того, щоб підключити датчик або актуатор до мережі *Z-Wave* досить на пару секунд натиснути одночасно на контролері і на самому пристрої невелику кнопку – це все, що необхідно;

– Високий рівень безпеки. Всі повідомлення в мережі *Z-Wave* шифруються за допомогою криптостійкого 128-бітного ключа. З огляду на, що сигнал мережі *Z-Wave* практично неможливо зловити поза приміщенням, де розгорнута бездротова мережа – зламати *Z-Wave* "зовні» практично неможливо. Це підтверджують практики кіберзлочинності – незважаючи на багаторічні спроби, отримати несанкціонований доступ до мережі «розумного будинку», побудованого на базі *Z-Wave*, так і не вдалося;

– Офіційно дозволений діапазон. Протокол Z-Wave працює в частотному діапазоні (869 МГц), який офіційно дозволений для використання пристроями з малою величиною радіосигналу. У різних країнах, діапазон частот, виділених під малопотужні мережі датчиків і актуаторов

відрізняється і зараз на українському ринку присутні компоненти Z-Wave, з робочими частотами – найчастіше, з американським і європейським діапазоном.

Рис. 1.9. Лампочка *Z-Wave*

При створенні розумного будинку в рамках окремої квартири використання «чужих» частот таїть в собі лише одну істотну неприємність: в майбутньому, якщо користувач вирішить додати в мережу додаткові датчики або актуатори, він може зіткнутися з тим, що компонентів з потрібною йому частотою не буде в продажу.

1.4. Висновки до розділу

В першому розділі було проведено дослідження систем “*Smart House*”, додатків для керування ними та технології *Z-Wave*. Розумний дім – це резиденція, яка використовує підключені до Інтернету пристрої для віддаленого моніторингу та управління приладами та системами, такими як освітлення та опалення.

Zigbee та *Z-Wave* – два найпоширеніші протоколи зв'язку домашньої автоматизації, що нині популярні. Обидва використовують сітчасті мережеві технології, радіосигнали малої потужності для підключення розумних домашніх систем. Хоча обидва націлені на одні й ті самі розумні домашні програми, *Z-Wave* має радіус дії 30 метрів, *Zigbee* – до 10 метрів, причому *Zigbee* часто сприймається як більш складний з двох. Чіпи *Zigbee* доступні у багатьох компаній, тоді як чіпи *Z-Wave* доступні лише від *Sigma Designs*.

Z-Wave – це набір пропрієтарних протоколів фізичного (що визначають на якій частоті працює мережа, рівень сигналу, що передається, його модуляцію) і логічного (такі протоколи задають адресацію пристроїв, набір команд, послідовність обміну інформацією) рівня, реалізованих в декількох наборах мікросхем *Sigma Design*.

Хоча багато розумних домашніх пристроїв, особливо концентраторів та інших продуктів для підключення, поєднуються зі своїми власними програмами, існує велика кількість програм, які працюють незалежно для підключення та управління пристроями.

Деякі програми, програми працюють лише для певних пристроїв, інші є більш універсальними. Хоча багато з цих програм виконують подібні функції, наприклад, повідомляють вас про важливі події вдома, або віддалено включають світло або побутові прилади, є деякі важливі відхилення з дуже специфічними функціями.

Серед основних функціональних характеристики веб-додатку для розумного дому можно виокремити наступні: швидкість, надійність, простота, захищеність, мультиплатформеність, вартість, можливість інтеграції з різними девайсами. Найкращими додатками, на які слід орієнтуватися у розробці є *Amazon Alexa*, *Google Assistant*, *Samsung SmartThings Hub*.

РОЗДІЛ 2
ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ВЕБ-ДОДАТКУ

2.1. Трирівнева клієнт-серверна архітектура

Трирівнева архітектура – це тип програмної архітектури, що складається з трьох «рівнів» логічних обчислень. Вона часто використовуються в додатках як певний тип системи клієнт-сервер. Трирівневі архітектури забезпечують багато переваг для середовищ виробництва та розробки, модулюючи користувальницький інтерфейс, бізнес-логіку та рівні зберігання даних. Це дає більшу гнучкість командам розробників, дозволяючи їм оновлювати певну частину програми незалежно від інших частин. Ця додаткова гнучкість може покращити загальний час випуску на ринок та зменшити час циклу розробки, надаючи командам розробників можливість змінювати або модернізувати незалежні рівні, не впливаючи на інші частини системи.

Наприклад, користувальницький інтерфейс веб-програми може бути перероблений або модернізований, не впливаючи на основний функціональний бізнес та логіку доступу до даних. Ця архітектурна система часто ідеальна для вбудовування та інтеграції програмного забезпечення сторонніх розробників у існуючу програму. Ця гнучкість інтеграції також робить її ідеальним для вбудовування програмного забезпечення у вже існуючі програми, а також їх аналітики і з цієї причини часто використовується постачальниками вбудованих аналітичних інструментів. Трирівневі архітектури часто використовуються в хмарних або локальних додатках, а також у додатках *software-as-a-service* (*SaaS*).

Трирівнева архітектура складається з:

– Рівень презентації. Рівень презентації є початковим кінцевим шаром у трирівневій системі та складається з користувацького інтерфейсу. Цей користувальницький інтерфейс часто є графічним, доступним через веб-браузер або веб-програму, який відображає вміст та інформацію, корисні кінцевому користувачеві. Цей рівень часто побудований на веб-технологіях,

таких як *HTML5, JavaScript, CSS*, або за допомогою інших популярних фреймворків веб-розробки, і спілкується з іншими рівнями за допомогою викликів *API*;

– Рівень додатків. Рівень додатків містить функціональну бізнес-логіку, яка керує основними можливостями програми. Це часто пишеться на *Java, .NET, C #, Python, C ++, PHP* тощо;

– Рівень даних. Рівень даних складається з бази даних / системи зберігання даних та рівня доступу до даних. Прикладами таких систем є *MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB* тощо. Доступ до даних здійснюється рівнем додатків через виклики API.

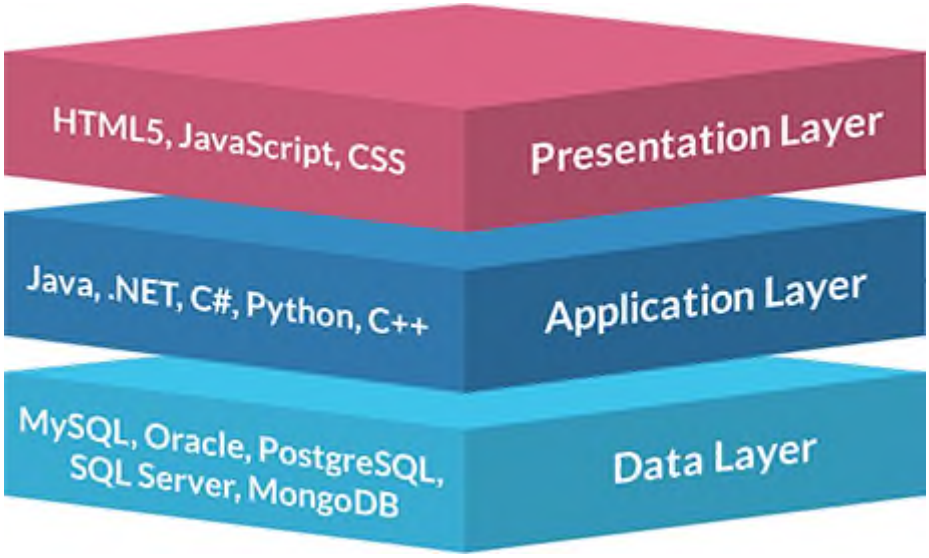

Рис. 2.1. Складові трирівневої архітектури

Типовою структурою для розгортання архітектури з 3 рівнями буде рівень презентації, розгорнутий на робочому столі, ноутбуці, планшеті чи мобільному пристрої через веб-браузер або веб-програму, що використовує веб-сервер. Базовий рівень додатків зазвичай розміщується на одному або декількох серверах, але також може розміщуватися у хмарі або на спеціальній робочій станції залежно від складності та обчислювальної потужності, необхідної додатку. А рівень даних зазвичай складався б з однієї або декількох реляційних баз даних, джерел великих даних або інших типів систем баз даних, розміщених або локально, або в хмарі.

Простим прикладом трирівневої архітектури може бути вхід в медіа-акаунт, такий як *Netflix*, і перегляд відео. Користувач починає з входу через Інтернет або через мобільний додаток. Після входу юзер може отримати доступ до певного відео через інтерфейс *Netflix*, який є рівнем презентації, який використовується вами як кінцевим користувачем. Після того, як споживач вибрав відео, інформація передається на рівень програми, який запитує рівень даних, щоб викликати інформацію, або в цьому випадку відео, резервне копіювання на рівень презентації. Це трапляється кожного разу, коли користувач отримує доступ до відео з більшості медіа-сайтів.

Використання 3-рівневої архітектури має багато переваг, включаючи швидкість розробки, масштабованість, продуктивність та доступність. Як вже згадувалося, модульність різних рівнів програми надає командам розробників можливість розробляти та вдосконалювати продукт із більшою швидкістю, ніж розробка бази єдиного коду, оскільки певний шар можна модернізувати з мінімальним впливом на інші рівні. Це також може допомогти підвищити ефективність розвитку, дозволяючи командам зосередитись на своїх основних компетенціях. Багато команд розробників мають окремих розробників, які спеціалізуються на розробці інтерфейсу, серверного серверного середовища та внутрішнього розвитку даних, модулюючи ці частини програми, компанії більше не доведеться покладатися на розробників повних стеків, і це може краще використовувати можливості кожного розробника.

Використання 3-шарової архітектури має багато переваг, включаючи швидкість розробки, масштабованість, продуктивність та доступність. Як вже згадувалося, модулярізація різних рівнів програми надає командам розробників можливість розробляти та вдосконалювати продукт із більшою швидкістю, ніж розробка бази єдиного коду, оскільки певний шар можна модернізувати з мінімальним впливом на інші рівні. Це також може допомогти підвищити ефективність розвитку, дозволяючи командам зосередитись на своїх основних компетенціях. Багато команд розробників мають окремих розробників, які спеціалізуються на розробці інтерфейсу, серверного серверного середовища та

внутрішнього розвитку даних, модулюючи ці частини програми, вам більше не доведеться покладатися на розробників повних стеків, і можна краще використовувати спеціальності кожного учасника команди.

Масштабованість – ще одна велика перевага 3-шарової архітектури. Виділяючи різні шари, можна масштабувати кожен незалежно залежно від потреби в будь-який момент часу. Наприклад, якщо отримується багато веб-запитів, але не так багато запитів, які впливають на рівень вашого додатка, розробник може масштабувати свої веб-сервери, не торкаючись серверів своїх додатків. Подібним чином, якщо отримується багато запитів на великі програми лише від кількох веб-користувачів, можна масштабувати свої додатки та рівні даних, щоб задовольнити ці запити, не торкаючись веб-серверів. Це дозволяє збалансувати навантаження кожного шару самостійно, покращуючи загальну продуктивність з мінімальними ресурсами. Крім того, незалежність, створена від модуляризації різних рівнів, дає безліч варіантів розгортання. Наприклад, можна створити налаштування, щоб веб-сервери додатку розміщувались у загальнодоступній або приватній хмарі, поки розробники використовують додатки та шари даних на місці. Або можна розміщувати свою програму та рівні даних у хмарі, тоді як веб-сервери можуть розміщуватися локально.

Маючи різні шари, також можна підвищити надійність та доступність, розміщуючи різні частини програми на різних серверах та використовуючи кешовані результати. З повною стековою системою вам доведеться турбуватися про те, що сервер буде падати і це буде сильно впливати на продуктивність усієї вашої системи, але з 3-рівневим додатком підвищується його незалежність, що створюється при фізичному розділенні різних частин програми та мінімізуються проблеми з продуктивністю, коли сервер ламається.

2.2. *Laravel Framework*

Laravel – це фреймворк з відкритим кодом *PHP*, розроблений для спрощення та швидкої розробки веб-програм завдяки вбудованим функціям. Ці функції є

частиною того, що робить *Laravel* настільки широко використовуваним веб-розробниками:

– Модульна пакувальна система з управлінням залежностями. Це означає, що розробники можуть легко додавати функціональні можливості у свій додаток *Laravel*, не записуючи їх з нуля. Існує можливість створити власні пакети для коду, який регулярно використовується, або встановити готові до використання пакети через *Composer*;

– Повна система аутентифікації;

– Об'єктно-реляційне відображення. Красномовний *ORM*, що входить до складу *Laravel*, представляє таблиці баз даних як класи для полегшення доступу до даних та керування ними;

– Інтерфейс командного рядка (*CLI*), який постачається з десятками попередньо побудованих команд (*Artisan*);

– Автоматичне тестування. Автоматизовані тести пропонуються як невід'ємна частина *Laravel*;

– Портативне віртуальне середовище розробки. *Homestead* надає розробникам всі інструменти, необхідні для розробки *Laravel*;

– *MVC* структура коду. Структура коду *Laravel framework* відповідає популярному паттерну проектування *MVC*, тобто в ньому можна виділити моделі (*models*), уявлення (*views*) і контролери (*controllers*). Даний шаблон проектування зарекомендував себе як перевірене часом рішення ефективної структури додатків, що дозволяє відокремити логіку додатку від його візуальної частини;

– Власний blade шаблонізатор.

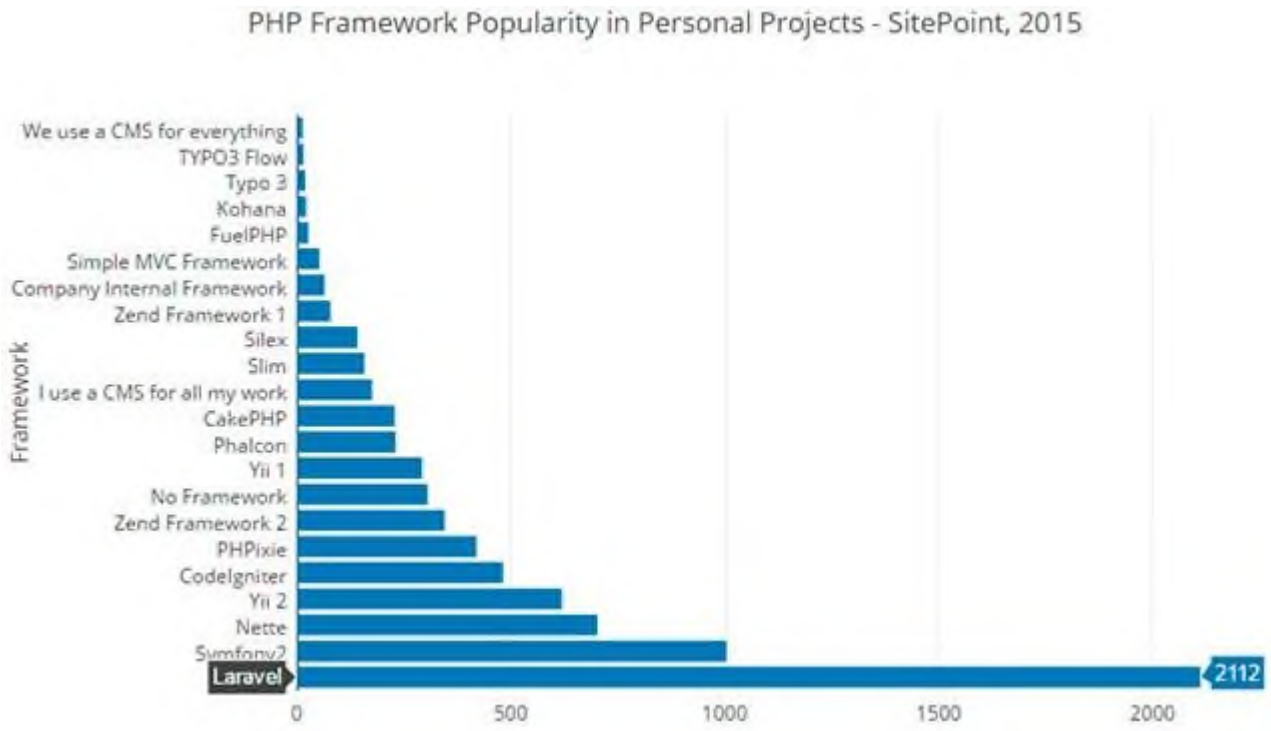

Рис. 2.2. Популярність *PHP* Фреймворків

Веб-сайти на *Laravel* відрізняються:

– широким функціоналом (можна зробити проект з практично будь-яким необхідним функціоналом);

– зручною адміністративною панеллю (можна зробити панель конкретно під певний проект і його завдання);

– високим рівнем безпеки баз даних (сайти надійно захищені від *SQL*-ін'єкцій);

– масштабованість (функціонал проекту можна легко розширити).

2.2.1. Менеджер залежностей *Composer*

Composer – менеджер залежностей для *PHP*. Його можна використовувати для встановлення, відстеження та оновлення залежностей вашого проекту. *Composer* також дбає про автоматичне завантаження залежностей, на які покладається ваша програма, дозволяючи вам легко використовувати залежність всередині вашого проекту, не турбуючись про те, щоб підключити їх у верхній частині будь-якого файлу.

Залежності для вашого проекту перераховані у файлі *composer.json,* який зазвичай знаходиться в корені вашого проекту. Цей файл містить інформацію про необхідні версії пакунків для виробництва, а також для розробки.

Цей файл можна редагувати вручну за допомогою будь-якого текстового редактора або автоматично через командний рядок за допомогою таких команд, як *composer require <package>* або *composer require-dev <package>*.

Щоб почати використовувати composer у своєму проекті, вам потрібно буде створити файл *composer.json*. Розробник може створити його вручну або просто запустити init composer. Після запуску композиційного *init* у своєму терміналі він запитає у вас основну інформацію про ваш проект: ім'я пакета (постачальник / пакет – наприклад, *laravel / laravel*), опис – необов’язково, автор та деяка інша інформація, така як мінімальна стабільність, ліцензія та необхідні пакети.

Ключ *require* у вашому файлі *composer.json* визначає від яких пакунків залежить ваш проект. *require* приймає об'єкт, який відображає імена пакетів (наприклад, *monolog / monolog*) до обмежень версії (наприклад, 1.0. *).

```
{
   "require": {
      "composer/composer": "1.2.*"
   }
}
```

Щоб встановити визначені залежності, вам потрібно буде запустити команду *composer install*, а потім вона знайде визначені пакети, що відповідають наданому обмеженню версії, і завантажить її в каталог постачальника. Це домовленість про розміщення стороннього коду в каталозі з ім’ям постачальника.

Можна помітити, що команда встановлення також створила файл *composer.lock.*

Файл *composer.lock* автоматично генерується *Composer*. Цей файл використовується для відстеження встановлених версій та стану ваших залежностей.

2.2.2. *Laravel migrations*

Міграції – це як контроль версій для вашої бази даних, що дозволяє вашій команді змінювати та надавати спільний доступ до схеми бази даних програми. Міграції, як правило, поєднуються з конструктором схем *Laravel* для побудови схеми бази даних програми.

Фасад схеми *Laravel* забезпечує агностичну підтримку бази даних для створення та обробки таблиць у всіх підтримуваних системах баз даних *Laravel*.

Клас міграції містить два методи: *up* та *down*. Метод *up* використовується для додавання нових таблиць, стовпців або індексів у вашу базу даних, тоді як метод *down* повинен змінити операції, виконані методом *up*.

В обох цих методах існує можливість використовувати конструктор схем *Laravel* для виразного створення та модифікації таблиць. Наприклад, наступна міграція створює таблицю польотів:

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
```

```
            $table->timestamps();
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Щоб запустити всі ваші міграції, виконайте команду *Artisan*:

php artisan migrate

2.2.3. *MVC* структура

Model-View-Controller (MVC) – це архітектурний шаблон, який розділяє додаток на три основні логічні компоненти: модель, вигляд та контролер. Кожен із цих компонентів створений для обробки конкретних аспектів розробки програми. *MVC* – одна з найбільш часто використовуваних галузевих стандартів веб-розробки для створення масштабованих та розширюваних проектів.

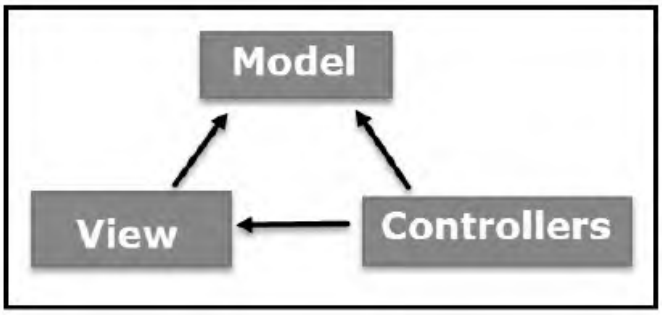

Рис. 2.3. Компоненти *MVC*

Компонент *Model* відповідає всій логіці даних, з якою працює користувач. Це може представляти або дані, що передаються між компонентами *View* та *Controller*, або будь-які інші дані, пов’язані з бізнес-логікою. Наприклад, об'єкт "Клієнт" буде отримувати інформацію про клієнта з бази даних, маніпулювати нею та оновлювати дані назад у базу даних або використовувати її для рендерингу даних.

Компонент *View* використовується для всієї логіки інтерфейсу програми. Наприклад, подання Клієнт включатиме всі компоненти інтерфейсу, такі як текстові поля, випадаючі меню тощо, з якими взаємодіє кінцевий користувач.

Контролери виступають інтерфейсом між компонентами *Model* і *View* для обробки всієї бізнес-логіки та вхідних запитів, маніпулювання даними за допомогою компонента *Model* та взаємодії з поданнями для надання кінцевого результату. Наприклад, контролер замовника буде обробляти всі взаємодії та входи з перегляду замовника та оновлювати базу даних за допомогою моделі замовника. Той самий контролер буде використовуватися для перегляду даних Клієнта.

Створюючи *PHP*-програми, може бути нормально мати багато файлів, що літають у дуже маленьких проектах. Однак, коли проект стає навіть трохи більшим за п’ять файлів або точок входу, що мають структуру, це може суттєво покращити ремонтопридатність.

Коли вам доводиться працювати з базами кодів, які не мають архітектури, це стане надзвичайно виснажливим, особливо якщо проект великий і вам доводиться мати справу з неструктурованим кодом, що лежить скрізь. Використання *MVC* може надати коду певну структуру та полегшити роботу.

Більш технічно, при побудові з використанням архітектури *MVC* існують наступні стратегічні переваги:

– Розподілити ролі у вашому проекті простіше. У компанії може бути серверний розробник, який працює над логікою контролера, тоді як розробник інтерфейсу працює над поданнями. Це дуже поширений спосіб роботи в компаніях, і наявність *MVC* робить це набагато простішим, ніж коли кодова база має код спагетті;

– Структурно "добре". *MVC* може змусити вас розділити файли на логічні каталоги, що полегшує пошук файлів під час роботи над великими проектами;

– Ізоляція відповідальності;

– Коли використовується архітектура *MVC*, кожна широка відповідальність є ізольованою. Наприклад, можна вносити зміни у подання та моделі окремо, оскільки модель не залежить від представлень;

– Повний контроль над *URL*-адресами додатків. За допомогою архітектури *MVC* є повний контроль над тим, як ваша програма виглядає у світі, вибравши маршрути додатків. Це стає в нагоді, коли розробники намагаєтеся вдосконалити свою програму для цілей *SEO*;

– Написати *SOLID* код простіше. За допомогою *MVC* легше дотримуватися принципу *SOLID*.

2.2.4. *ORM Eloquent*

Фреймворк *PHP Laravel* оснащений *Eloquent Object Relational Mapper (ORM)*, який забезпечує надзвичайно простий спосіб спілкування з базою даних. Оскільки розробникам потрібно створювати складні веб-сайти та інші програми, вони віддають перевагу безпроблемному та коротшому часу розробки. *Eloquent* допомагає пришвидшити розробку та забезпечує адекватне вирішення більшості проблем, що виникають. Різні вимоги бізнесу вирішуються за допомогою швидкого розвитку, а також добре організованого, багаторазового використання, що підтримується та масштабованого коду. *Eloquent* працює зі спеціальними веб-додатками, оскільки може обслуговувати декілька баз даних та виконувати загальні операції з базами даних.

Розробники можуть ефективно працювати з кількома базами даних, використовуючи реалізацію *ActiveRecord*. Це архітектурний шаблон, де модель, створена в структурі *Model-View-Controller* (*MVC*), відповідає таблиці в базі даних. Перевага полягає в тому, що моделі можуть виконувати загальні операції з базами даних без кодування тривалих запитів *SQL*. Моделі дозволяють запитувати дані у

ваших таблицях, а також вставляти нові записи в таблиці. Надано спрощений процес синхронізації декількох баз даних, що працюють в різних системах. Немає необхідності писати запити *SQL* взагалі. Все, що вам потрібно зробити, це визначити таблиці бази даних та відносини між ними, і *Eloquent* виконає решту роботи.

Шаблон *ActiveRecord* – це підхід до доступу до даних у базі даних. Таблиця бази даних або подання переносяться у клас. Таким чином, екземпляр об'єкта прив'язаний до одного рядка в таблиці. Після створення об'єкта до таблиці при збереженні додається новий рядок. Будь-який завантажений об’єкт отримує свою інформацію з бази даних. Коли об’єкт оновлюється, відповідний рядок у таблиці також оновлюється. Клас обгортки реалізує методи доступу або властивості для кожного стовпця таблиці або подання.

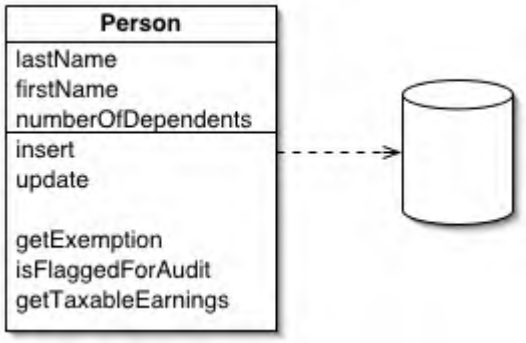

Рис.2.4. Шаблон *ActiveRecord*

Eloquent підтримує такі типи відносин:

– *One To One*. Відносини *One To One* – це дуже базове відношення. Наприклад, модель *User* може мати один *Phone*:

```
class User extends Model {
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

– *One To Many*. Прикладом відношення *One To Many* є публікація в блозі, яка "має багато" коментарів:

```
class Post extends Model {
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

– *Many To Many*. Відносини *Many To Many* є більш складним типом відносин. Прикладом таких відносин є користувач із багатьма ролями, де ролі також спільно використовуються іншими користувачами. Наприклад, багато користувачів можуть виконувати роль "Адміністратора". Для цього взаємозв'язку потрібні три таблиці бази даних: *users*, *roles* та *role_user*. Таблиця *role_user* походить з алфавітного порядку відповідних імен моделей і повинна мати стовпці *user_id* та *role_id*. Ми можемо визначити відношення *Many To Many* за допомогою методу *belongsToMany*:

```
class User extends Model {
    public function roles()
    {
        return $this->belongsToMany('App\Role', ‘user_roles’, ‘user_id’, ‘role_id’);
    }
}
```

– *Has Many Through*. Співвідношення *Has Many Through* забезпечує зручний ярлик для доступу до віддалених відносин через проміжне відношення. Наприклад, у моделі *Country* може бути багато публікацій через модель користувача:

```
class Country extends Model {
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

– *Polymorphic Relations.* Поліморфні співвідношення дозволяють моделі належати більш ніж одній моделі на одній асоціації. Наприклад, у вас може бути фотомодель, яка належить або моделі персоналу, або моделі замовлення:

```
class Photo extends Model {
    public function imageable()
    {
        return $this->morphTo();
    }
}
class Staff extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
class Order extends Model {
    public function photos()
    {
        return $this->morphMany('App\Photo', 'imageable');
    }
}
```

– *Many To Many Polymorphic Relations.* На додаток до традиційних поліморфних відносин, також можна вказати *Many To Many Polymorphic Relations*. Наприклад, модель публікації та відео в блозі може поділяти поліморфне відношення до моделі тегу:

```
class Post extends Model {
    public function tags()
    {
```

```
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

2.2.5. *Nginx*

Nginx – це програмне забезпечення з відкритим кодом для веб-сервісу, зворотного проксі-сервера, кешування, балансування навантаження, потокового передавання медіа тощо. Він розпочався як веб-сервер, розроблений для максимальної продуктивності та стабільності. На додаток до можливостей сервера *HTTP, Nginx* може також функціонувати як проксі-сервер для електронної пошти (*IMAP, POP3* та *SMTP*) та зворотний проксі-сервер та балансування навантаження для серверів *HTTP, TCP* та *UDP*.

Метою *Nginx* було створити найшвидший веб-сервер, і підтримка такої досконалості досі є центральною метою проекту. *Nginx* послідовно перемагає *Apache* та інші сервери в тестах, що вимірюють продуктивність веб-серверів. З моменту первісного випуску *Nginx*, веб-сайти розширилися від простих *HTML*-сторінок до динамічного багатогранного вмісту. *Nginx* виріс разом із ним і тепер підтримує всі компоненти сучасного Інтернету, включаючи *WebSocket, HTTP* / 2 та потокове передавання декількох відеоформатів (*HDS, HLS, RTMP* та інші).

Хоча *Nginx* прославився як найшвидший веб-сервер, масштабована основна архітектура виявилася ідеальною для багатьох веб-завдань, окрім обслуговування контенту. Оскільки *Nginx* може обробляти великий обсяг з'єднань, він зазвичай використовується як зворотний проксі-сервер і балансувальник навантаження для управління вхідним трафіком і розподілу його на повільніші висхідні сервери – від старих серверів баз даних до мікросервісів.

Nginx створений для забезпечення низького обсягу використання пам'яті та високої паралельності. Замість того, щоб створювати нові процеси для кожного веб-запиту, *Nginx* використовує асинхронний підхід, керований подіями, де запити обробляються в одному потоці.

За допомогою *Nginx* один головний процес може керувати кількома робочими процесами. Він підтримує робочі процеси, тоді як робітники виконують фактичну обробку. Оскільки *Nginx* асинхронний, кожен запит може виконуватися одночасно, не блокуючи інші запити.

Серед загальних особливостей *Nginx*:

– Зворотний проксі з кешуванням;

– *IPv6;*

– Балансування навантаження;

– Підтримка *FastCGI* з кешуванням;

– *WebSockets;*

– Обробка статичних файлів, файлів індексу та автоматичне індексування;

– *TLS* / *SSL* із *SNI.*

Apache – ще один популярний веб-сервер з відкритим кодом. За даними *W3Techs, Apache* – найпопулярніший існуючий веб-сервер, який використовується 43,6% (порівняно з 47% у 2018 році) серед усіх веб-сайтів з відомим веб-сервером. *Nginx* займає друге місце – 41,8%.

Компанія *Netcraft* провела опитування серед 233 мільйонів доменів і виявила, що використання *Apache* становить 31,54%, а використання *Nginx* – 26,20%.

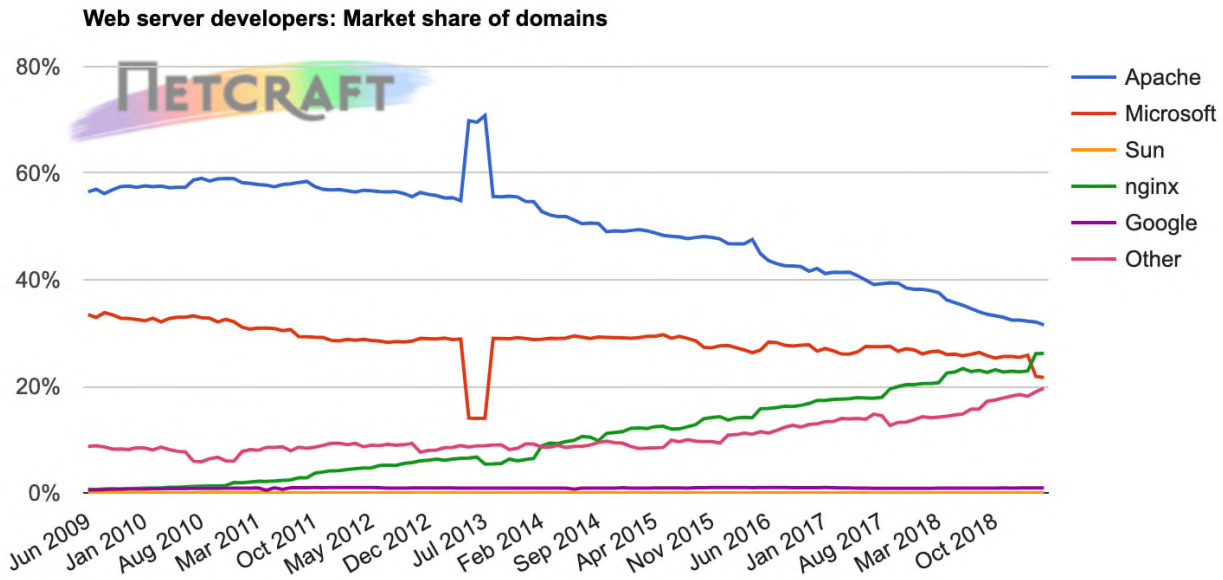

Рис. 2.5. Найпопулярніші світові веб-сервери

Хоча *Apache* – найпопулярніший загальний варіант, *Nginx* – насправді найпопулярніший веб-сервер серед веб-сайтів із високим трафіком.

Коли розробники розбивають показники використання за трафіком, *Nginx* забезпечує:

– 60,9% із 100 000 найпопулярніших сайтів (з 56,1% у 2020 році);

– 67,1% з 10 000 найпопулярніших сайтів (проти 63,2% у 2020 році);

– 62,1% з 1000 найпопулярніших сайтів (з 57% у 2020 році).

Насправді *Nginx* використовується деякими найбільш ресурсоємними веб-сайтами, що існують, зокрема *Netflix*, *NASA* та навіть *WordPress.com*.

З іншого боку, використання *Apache* зменшується, коли збільшується трафік на сайті:

24,0% із 100 000 найпопулярніших сайтів (порівняно з 27,1% у 2020 році);

18,8% з 10 000 найпопулярніших сайтів (з 21,5% у 2020 році);

16,6% з 1000 найпопулярніших сайтів (з 16,2% у 2020 році).

Якщо ми поглянемо на пошукові терміни в *Google* з 2004 року, то побачимо, що кількість запитів з *Apache* поступово падають, тоді як *Nginx* спостерігає незначне зростання.

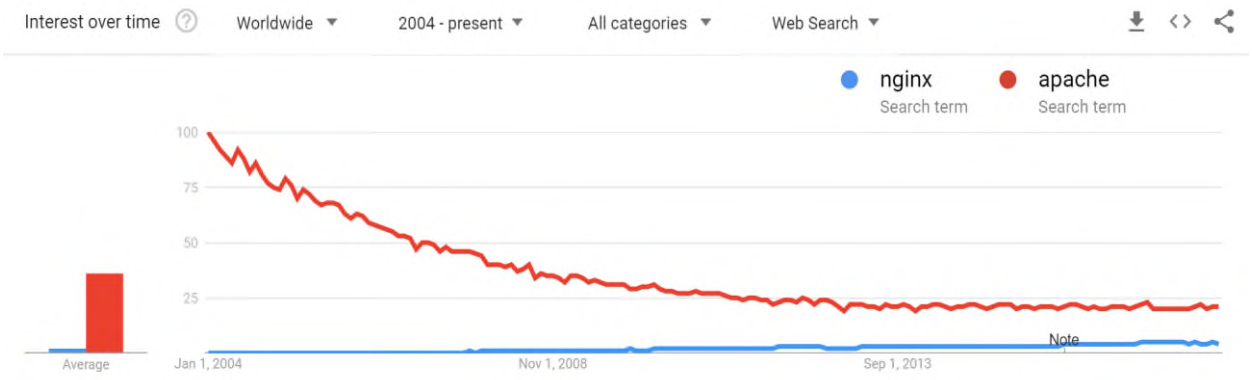

Рис. 2.6. *Apache* проти *Nginx*

2.3. *React Native*

React Native – це фреймворк, який створює ієрархію компонентів інтерфейсу для побудови коду *JavaScript*. Він має набір компонентів як для платформ *iOS*, так і для *Android* для створення мобільного додатка з власним виглядом та стилем. З іншого боку, *ReactJS* – це бібліотека *JavaScript* з відкритим кодом для створення користувальницьких інтерфейсів. Однак і *React Native*, і *ReactJS* розробляються *Facebook*, використовуючи однакові принципи дизайну, за винятком проектування інтерфейсів.

React Native допомагає створювати справжні та захоплюючі мобільні додатки лише за допомогою *JavaScript*, який підтримується як для платформ *Android*, так і для *iOS*. Знайшовши велику популярність, а також за підтримки *Facebook*, *REACT Native*, сьогодні має величезну підтримку спільноти. *React Native* побудований на версії *ReactJS*, що створило величезну конкуренцію давно улюбленому *AngularJS*.

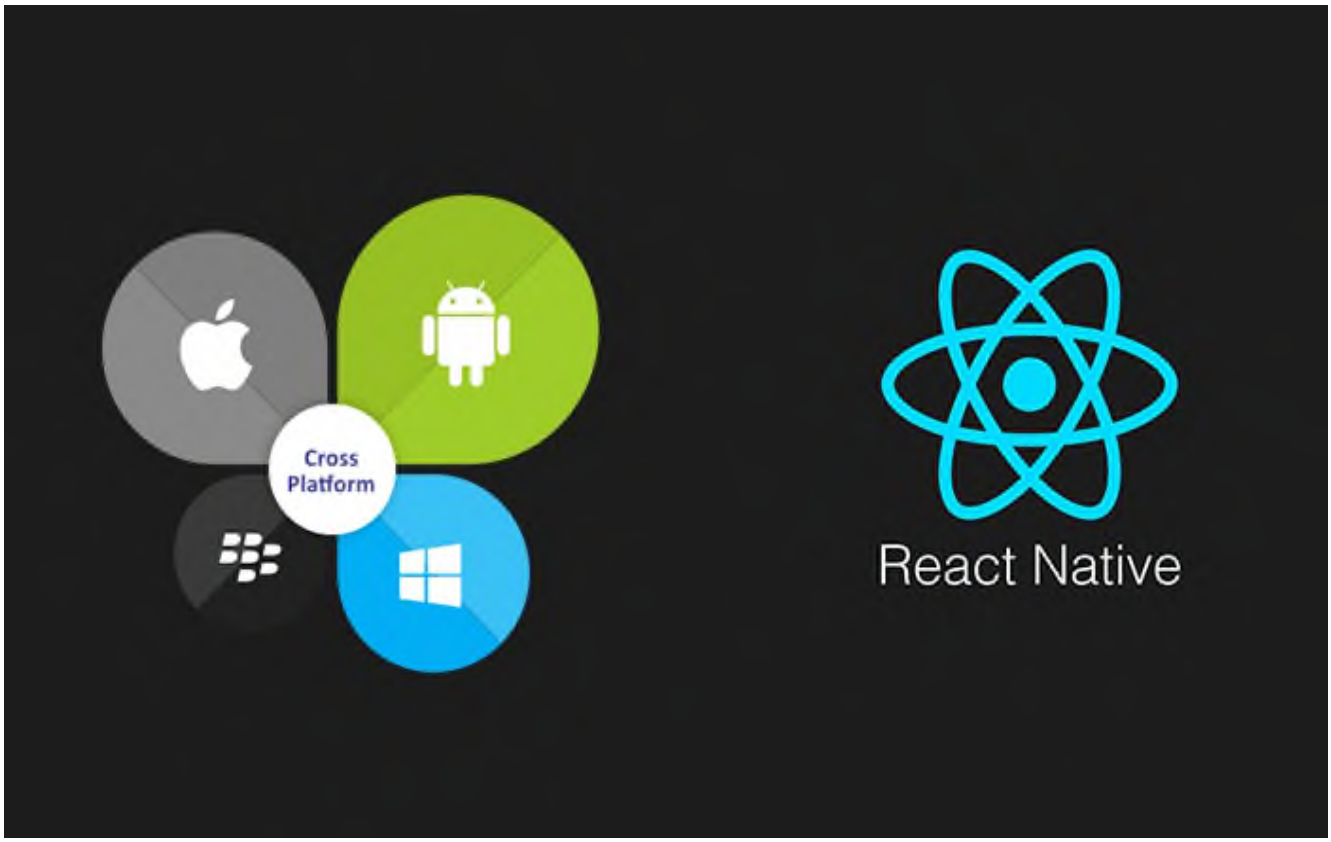

Рис. 2.7. *React Native*

За допомогою *React Native Framework* розробники можуть відображати інтерфейс для платформ *iOS* і *Android*.

Це фреймворк з відкритим кодом, який найближчим часом може бути сумісним з іншими платформами, такими як *Windows* або *tvOS*.

Оскільки компоненти *React Native* мають відповідні права, програмісти можуть використовувати їх повторно для створення додатків для *Android* та *iOS*.

Розробник можете або включити рідні компоненти *REACT* в код існуючої програми, або повторно використати код на основі Кордови за допомогою плагіна. Однак ваш існуючий додаток повинен бути побудований з використанням кордови та іонічного коду.

Даний фреймворк порівняно простий, швидкий та ефективний.

React Native – чудовий вибір для тих розробників, які мають досвід роботи з *JavaScript*, оскільки немає необхідності вивчати спеціальну *Java*-програму для *Android* або *Swift* для *iOS*.

React Native – це орієнтоване на користувальницький інтерфейс, що дозволяє швидко завантажувати програми та надає більш плавне відчуття.

2.4. *Docker* та *Docker Compose*

Docker – це відкрита платформа для розробки, розгортання та запуску додатків. *Docker* дозволяє відокремити ваші програми від інфраструктури, щоб розробники могли швидко реалізувати програмне забезпечення. За допомогою *Docker* можна керувати інфраструктурою так само, як і програмами. Скориставшись методологіями *Docker* для швидкої реалізації, тестування та розгортання коду, можна значно скоротити затримку між написанням коду та його запуском у виробництво.

Docker надає можливість упаковувати та запускати додаток у вільно ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на даному хості. Контейнери легкі, оскільки їм не потрібно додаткове навантаження гіпервізора, але вони

працюють безпосередньо в ядрі хост-машини. Це означає, що розробник може запускати більше контейнерів на певній апаратній комбінації, ніж якби використовували віртуальні машини. Існує навіть можливість запускати контейнери *Docker* у хост-машинах, які насправді є віртуальними машинами!

Docker Engine – це програма клієнт-сервер з такими основними компонентами:

– *REST API*, який визначає інтерфейси, які програми можуть використовувати для спілкування з демоном та вказівки йому, що робити.

– Клієнт інтерфейсу командного рядка (*CLI*) (команда *docker*).

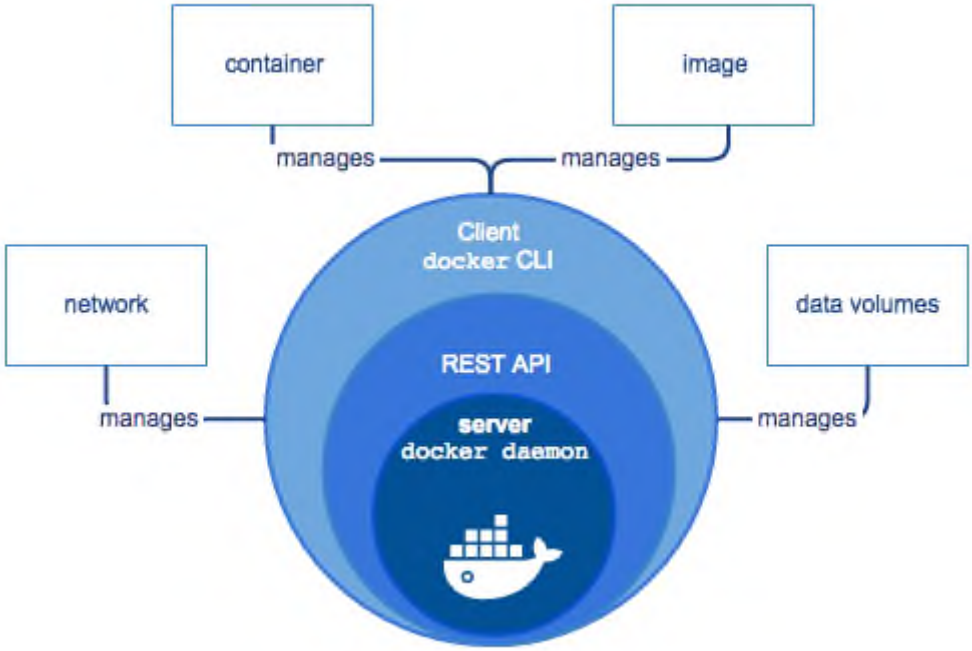

Рис. 2.8. Структура *Docker*

CLI використовує *Docker REST API* для управління або взаємодії з демоном *Docker* за допомогою сценаріїв або прямих команд. Багато інших програм *Docker* використовують базовий *API* та *CLI*.

Демон створює та керує об'єктами *Docker*, такими як зображення, контейнери, мережі та томи.

Docker впорядковує життєвий цикл розробки, дозволяючи розробникам працювати в стандартизованих середовищах, використовуючи локальні

контейнери, які надають ваші програми та послуги. Контейнери чудово підходять для безперервної інтеграції та безперервної доставки робочих процесів (*CI* / *CD*).

Платформа і засоби контейнерної віртуалізації можуть бути корисні в таких випадках:

– пакування вашого додатку (і так само використовуваних компонент) в *docker* контейнери;

– роздача і доставка цих контейнерів вашим командам для розробки і тестування;

– викладання цих контейнерів на ваші продакшени, як в дата-центри так і в хмари.

Docker легкий і швидкий. Він надає стійку, рентабельну альтернативу віртуальним машинам на основі гіпервізора. Він особливо корисний в умовах високих навантажень, наприклад, при створення власного хмари або платформа-як-сервіс (*platform-as-service*). Але він так само корисний для маленьких і середніх додатків, коли вам хочеться отримувати більше з наявних ресурсів.

Щоб розуміти, з чого складається *docker*, вам потрібно знати про його три компоненти:

– образи (*images*);

– реєстр (*registries*);

– контейнери.

Docker-образ – це *read-only* шаблон. Наприклад, образ може містити операційну систему *Ubuntu* с *Apache* і додатком на ній. Образи використовуються для створення контейнерів. *Docker* дозволяє легко створювати нові образи, оновлювати існуючі, або можна завантажити образи створені іншими людьми. Образи – це компонента збірки *docker*-а.

Docker-реєстр зберігає образи. Є публічні і приватні реєстри, з яких можна скачати або завантажити образи. Публічний *Docker*-реєстр – це *Docker Hub*. Там зберігається величезна колекція образів. Реєстри – це компонента поширення.

Контейнери схожі на директорії. У контейнерах міститься все, що потрібно для роботи програми. Кожен контейнер створюється з образу. Контейнери можуть

бути створені, запущені, зупинені, перенесені або видалені. Кожен контейнер ізольований і є безпечною платформою для додатка. Контейнери – це компонента роботи.

Команда *compose* – це інструмент для *Docker*, який використовується для визначення і запуску декількох додатків-контейнерів, в яких файл *compose* використовується для визначення необхідних для застосування сервісів.

При такому налаштуванні додатку всі сервіси можна запустити за допомогою однієї команди. Завдяки цьому додатки можуть бути створені, якщо виконати чотири прості кроки:

Налаштувати *Dockerfile* для визначення середовища додатки.

Додати файл вимог для завантаження пакетів *Laravel*.

Створити файл *docker-compose.yml* для визначення сервісів, які потрібні додатком. Вони будуть працювати разом в ізольованому середовищі.

Запустити збірку *docker-compose*, щоб створити додаток, і *docker-compose*, щоб запустити цей самий додаток.

Dockerfile

```
FROM microsoft/dotnet:sdk AS build-env
WORKDIR /Docker

# Copy csproj and restore as distinct layers

RUN dotnet restore

# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o out


# Build runtime image
FROM microsoft/dotnet:aspnetcore-runtime
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

Рис. 2.9. Приклад *Dockerfile*

2.5. Архітектура *Restful Api*

RESTful API – це архітектурний стиль інтерфейсу прикладних програм (*API*), який використовує *HTTP*-запити для доступу та використання даних. Ці дані можуть бути використані для отримання, збереження, оновлення та видалення типів даних, що стосується однойменних операцій для виділених ресурсів.

API для веб-сайту – це код, який дозволяє двом програмним системам взаємодіяти між собою.

RESTful API – також званий *RESTful* веб-сервісом або *REST API* – заснований на репрезентативній передачі стану (*REST*), що є архітектурним стилем та підходом до комунікацій, часто використовуваних при розробці веб-служб.

REST, що використовується браузерами, можна сприймати як мову Інтернету. Зі збільшенням використання хмарних споживачів *API* використовують хмарні споживачі для встановлення та організації доступу до веб-сервісів. *REST* є логічним вибором для побудови *API*, які дозволяють користувачам гнучко підключатися до хмарних служб та взаємодіяти з ними, у розподіленому середовищі. *API RESTful* використовуються такими сайтами, як *Amazon, Google*, *LinkedIn* і *Twitter*.

API RESTful розбиває транзакцію, створюючи серію невеликих модулів. Кожен модуль звертається до основної частини транзакції. Ця модульність надає розробникам велику гнучкість, але також і складність у розробці свого *REST API* з нуля. В даний час кілька компаній пропонують моделі для використання розробниками; моделі, що надаються *Amazon S3, Cloud Data Management Interface* (*CDMI*) та *OpenStack Swift*, є найбільш популярними.

API RESTful використовує команди для отримання ресурсів. Стан ресурсу в будь-який даний момент часу називається поданням ресурсу. *API RESTful* використовує існуючі методології *HTTP*, визначені протоколом *RFC 2616*, а саме:

– *GET* отримати ресурс;

– *PUT* для зміни стану або оновлення ресурсу, який може бути об'єктом, файлом або блоком;

– *POST* для створення цього ресурсу;

– *PATCH* для зміни стану або оновлення частини ресурсу, який може бути об'єктом, файлом або блоком;

– *DELETE*, щоб видалити його.

Формати даних, які підтримує *API REST*, включають:

– *application/json;*

– *application/xml;*

– *application/x-wbe+xml;*

– *application/x-www-form-urlencoded;*

– *multipart/form-data.*

Дизайн *RESTful API* був визначений доктором Роєм Філдінгом у його докторській дисертації 2000 року. Щоб бути справжнім *RESTful API*, веб-служба повинна дотримуватися наступних шести архітектурних обмежень *REST*:

– Використання єдиного інтерфейсу (*UI*). Ресурси мають бути однозначно ідентифіковані за однією *URL*-адресою, і лише за допомогою основних методів мережевого протоколу, таких як *DELETE, PUT* та *GET* з *HTTP*, має бути можливим маніпулювання ресурсом;

– Клієнт-сервер. Між клієнтом і сервером має бути чітке розмежування. Проблеми щодо користувацького інтерфейсу та збору запитів – це домен клієнта. Доступ до даних, управління навантаженням та безпека – це домен сервера. Це вільне з'єднання клієнта та сервера дозволяє розробити та вдосконалити кожен, незалежно від іншого;

– Операції без стану. Усі операції клієнт-сервер повинні бути без стану, і будь-яке необхідне управління станом повинно відбуватися на клієнті, а не на сервері;

– Кешування ресурсів *RESTful*. Усі ресурси повинні дозволяти кешування, якщо прямо не вказано, що кешування неможливо;

– Багатошарова система. *REST* дозволяє створити архітектуру, що складається з декількох шарів серверів;

– Код на вимогу. Здебільшого сервер надсилає назад статичні подання ресурсів у вигляді *XML* або *JSON*. Однак при необхідності сервери можуть надсилати виконуваний код клієнту.

Окрім дизайнерських та архітектурних обмежень, людям доведеться стикатися з деякими проблемами при роботі з *REST API*. Деякі поняття, які можуть викликати ускладнення, можуть включати:

– Послідовність кінцевих точок – шляхи до кінцевих точок повинні узгоджуватися, дотримуючись загальноприйнятих веб-стандартів, якими важко керувати;

– Версія *API* – версії *URL*-адрес кінцевих точок не повинні бути недійсними при внутрішньому використанні або з іншими програмами;

– Довгий час відгуку та занадто багато даних – кількість повернутих ресурсів може збільшуватися в часі, збільшуючи час навантаження та відгуку;

– Шляхи навігації та місця введення користувачем – оскільки *REST* використовує *URL*-адреси для вхідних параметрів, визначення просторів *URL* може бути складним завданням;

– Безпека. Доволі непростим завданням є розробка надійно захищеного *API*;

– Аутентифікація – використовуйте загальнодоступні методи автентифікації, такі як базова аутентифікація *HTTP* (що дозволяє кодувати ім’я користувача *base64*: рядок пароля), ключі *API*, веб-маркери *JSON* та інші маркери доступу. Наприклад, *OAuth* 2.0 добре підходить для контролю доступу;

– Запити та дані – можуть містити більше даних та метаданих, ніж потрібно, або для отримання всіх даних може знадобитися більше запитів. Для цього можна налаштувати *API*.

Тестування *API* – це довгий процес налаштування та запуску. Кожна частина процесу може бути тривалою або складною. Тестування також можна виконати в командному рядку за допомогою утиліти *Curl*.

2.6. Реляційна база даних *Mysql*

MySQL – це система управління реляційними базами даних (*RDBMS*) із підтримкою *Oracle* з відкритим кодом, заснована на мові структурованих запитів (*SQL*). *MySQL* працює практично на всіх платформах, включаючи *Linux, UNIX* та *Windows*. Хоча його можна використовувати в широкому діапазоні програм, *MySQL* найчастіше асоціюється з веб-додатками та публікацією в Інтернеті.

MySQL є важливим компонентом корпоративного стеку з відкритим кодом, який називається *LAMP*. *LAMP* – це платформа веб-розробки, яка використовує *Linux* як операційну систему, Apache як веб-сервер, *MySQL* як реляційну систему управління базами даних та *PHP* як об'єктно-орієнтовану мову сценаріїв. (Іноді замість *PHP* використовується *Perl* або *Python*.)

Спочатку задуманий шведською компанією *MySQL AB, MySQL* був придбаний *Sun Microsystems* у 2008 році, а потім *Oracle*, коли він придбав *Sun* у 2010 році. Розробники можуть використовувати *MySQL* під загальною публічною ліцензією *GNU* (*GPL*), але підприємства повинні отримати комерційну ліцензію від *Oracle*.

Сьогодні *MySQL* є СУБД, що стоїть за багатьма провідними веб-сайтами у світі та незліченними корпоративними та споживчими веб-додатками, включаючи *Facebook, Twitter та YouTube*.

MySQL базується на моделі клієнт-сервер. Ядром *MySQL* є сервер *MySQL*, який обробляє всі інструкції бази даних (або команди). Сервер *MySQL* доступний як окрема програма для використання в мережевому середовищі клієнт-сервер та як бібліотека, яка може бути вбудована в окремі програми.

Технологія працює разом з декількома утилітами, які підтримують адміністрування баз даних *MySQL*. Команди надсилаються на *MySQL Server* через клієнт *MySQL*, який встановлений на комп'ютері.

Спочатку *MySQL* був розроблений для швидкої обробки великих баз даних. Хоча *MySQL* зазвичай встановлюється лише на одній машині, він може надсилати базу даних у різні місця, оскільки користувачі мають доступ до неї через різні

клієнтські інтерфейси. Ці інтерфейси надсилають на сервер оператори *SQL*, а потім відображають результати.

MySQL дозволяє зберігати дані та отримувати доступ до них на декількох механізмах зберігання, включаючи *InnoDB, CSV* та *NDB*. *MySQL* також здатний тиражувати дані та таблиці розділів для кращої продуктивності та довговічності. Користувачам *MySQL* не потрібно вивчати нові команди; вони можуть отримати доступ до своїх даних за допомогою стандартних команд *SQL*.

MySQL написаний на *C* і *C++*, доступний і доступний на понад 20 платформах, включаючи *Mac, Windows, Linux* та *Unix*. СУБД підтримує великі бази даних з мільйонами записів і підтримує багато типів даних, включаючи підписані або беззнакові цілі числа 1, 2, 3, 4 і 8 байт та просторові типи *OpenGIS*. Також підтримуються типи рядків із фіксованою та змінною довжиною.

З міркувань безпеки *MySQL* використовує привілеї доступу та зашифровану систему паролів, що забезпечує перевірку на основі хосту. Клієнти *MySQL* можуть підключатися до *MySQL Server* за допомогою декількох протоколів, включаючи сокети *TCP / IP* на будь-якій платформі. *MySQL* також підтримує ряд клієнтських та службових програм, програм командного рядка та інструментів адміністрування, таких як *MySQL Workbench*.

До 2016 року основною відмінністю *MySQL* від *SQL* було те, що першу можна було використовувати на декількох платформах, тоді як другу можна було використовувати лише в *Windows*. З тих пір *Microsoft* розширила *SQL* для підтримки *Linux*, зміна, яка набула чинності в 2017 році. Коли *MySQL* встановлюється через *Linux*, її система управління пакетами вимагає власної конфігурації для налаштування параметрів безпеки та оптимізації.

MySQL також дозволяє користувачам вибрати найбільш ефективний механізм зберігання для будь-якої даної таблиці, оскільки програма може використовувати декілька механізмів зберігання для окремих таблиць. Одним з двигунів *MySQL* є *InnoDB*. *InnoDB* був розроблений для високої доступності. Через це він не такий швидкий, як інші двигуни. *SQL* використовує власну систему

зберігання, але вона підтримує безліч засобів захисту від втрати даних. Обидві системи можуть працювати в кластерах для високої доступності.

SQL Server пропонує широкий спектр інструментів для аналізу даних та звітування. Служби звітування *SQL Server* – найпопулярніший і доступний для безкоштовного завантаження. Існують подібні інструменти аналізу для *MySQL* від сторонніх програмних компаній, такі як *Crystal Reports XI* та *Actuate BIRT*.

MySQL був розроблений для сумісності з іншими системами. Він підтримує розгортання у віртуальних середовищах, таких як *Amazon RDS* та *Amazon Aurora*. Користувачі можуть передавати свої дані в базу даних *SQL Server* за допомогою інструментів міграції баз даних, таких як *AWS Schema Conversion Tool* та *AWS Database Migration Service*.

Семантика об'єктів бази даних між *SQL Server* і *MySQL* схожа, але не однакова. Існують архітектурні відмінності, які слід враховувати при переході з *SQL Server* на *MySQL*. У *MySQL* немає різниці між базою даних та схемою, тоді як *SQL Server* розглядає ці два як окремі сутності.

2.7. Висновки до розділу

Згідно з усіма необхідними вимогами до створення веб-додатку для керування електроприладами було обрано для реалізації трирівневу архітектуру.

Для реалізації рівня презентації було вирішено використовувати технологію *React Native* через простоту реалізації, швидкість та плавність роботи системи при використанні даної технології, а також можливість використовувати один код для кількох основних платформ, а саме : *Android, IOS*.

Для реалізації рівня додатку системи , було обрано *PHP* фреймворк *Laravel*.

Laravel – це фреймворк з відкритим кодом *PHP*, розроблений для спрощення та швидкої розробки веб-програм завдяки вбудованим функціям. Даний фреймворк є дуже популярним нині, має чудову документацію, підтримується і оновлюється, взаємодіє з такими потужними технологіями, як *composer, npm, homestead, docker*, має вбудований *ORM Eloquent*, для спрощення взаємодії з базою даних. З

допомогою даного фреймворку, код легко масштабується, розробляється та структурується, оскільки він підтримує паттерн проектування *MVC*. Також слід зазначити, що дана технологія чудово підходить для реалізації архітектурного стилю *REST API*.

Для локальної розробки, як систему віртуалізації було обрано *Docker*. *Docker* – це відкрита платформа для розробки, розгортання та запуску додатків. *Docker* дозволяє відокремити ваші програми від інфраструктури, щоб розробники могли швидко реалізувати програмне забезпечення. За допомогою *Docker* можна керувати інфраструктурою так само, як і програмами.

Docker надає можливість упаковувати та запускати додаток у вільно ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на даному хості. Контейнери легкі, оскільки їм не потрібно додаткове навантаження гіпервізора, але вони працюють безпосередньо в ядрі хост-машини.

Для реалізації рівня даних було обрано реляційну базу даних *MySQL*. *MySQL* – це система управління реляційними базами даних (*RDBMS*) із підтримкою *Oracle* з відкритим кодом, заснована на мові структурованих запитів (*SQL*). *MySQL* працює практично на всіх платформах, включаючи *Linux, UNIX та Windows*. Хоча його можна використовувати в широкому діапазоні програм, *MySQL* найчастіше асоціюється з веб-додатками та публікацією в Інтернеті.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВЕБ-ДОДАТКУ

3.1. Опис та встановлення необхідних програмних компонентів

Фреймворк *Laravel* має кілька системних вимог. Усі ці вимоги задовольняє віртуальна машина *Laravel Homestead*, тому рекомендується використовувати *Homestead* як місцеве середовище розробки *Laravel*.

Однак, якщо не використовувати *Homestead*, то потрібно переконатися, що налаштований сервер відповідає таким вимогам:

– *PHP >= 7.3*;
– *BCMath PHP Extension*;
– *Ctype PHP Extension*;
– *Fileinfo PHP Extension*;
– *JSON PHP Extension*;
– *Mbstring PHP Extension*;
– *OpenSSL PHP Extension*;
– *PDO PHP Extension*;
– *Tokenizer PHP Extension*;
– *XML PHP Extension*;
– *XML PHP Extension*.

Laravel використовує *Composer* для управління залежностями. Отже, перед використанням *Laravel* слід переконатися, що на обраному комп'ютері встановлено *Composer*.

Для встановлення *Composer* у *Linux* терміналі слід ввести наступні команди:

sudo apt install curl php-cli php-mbstring git unzip

curl -sS https://getcomposer.org/installer -o composer-setup.php

composer

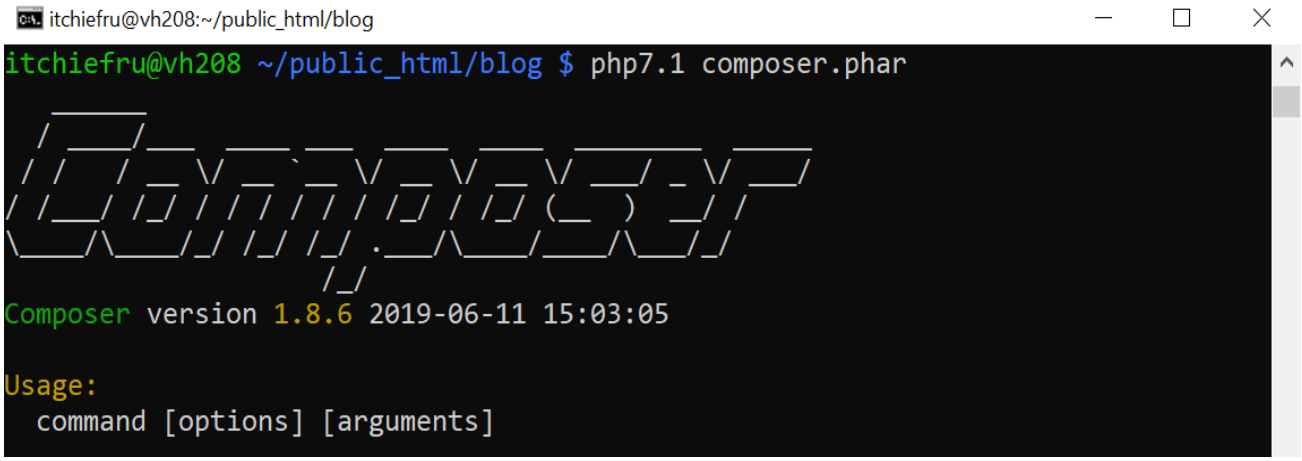

Рис. 3.1. *Composer*

Спочатку завантажте інсталятор *Laravel* за допомогою *Composer*:

composer global require laravel/installer

Варто помістити загальносистемний каталог коду постачальників компонентів у системний *$PATH*, щоб виконуваний файл *laravel* міг знаходити потрібну систему. Цей каталог існує в різних місцях залежно від використовуваної операційної системи.

Після встановлення команда *new laravel* створить нову установку *Laravel* у вказаному каталозі. Наприклад, *laravel new project* створить каталог із назвою *project*, що містить свіжу інсталяцію *Laravel* з усіма вже встановленими залежностями:

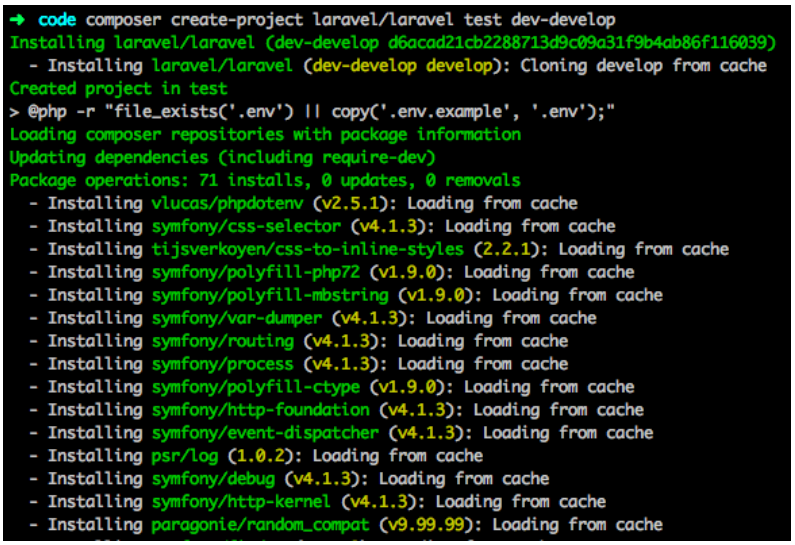

Рис. 3.2. Запуск команди створення проекту

Наступним кроком необхідно створити файл з змінними оточення проекту – *.env*. Даний файл необхідний для правильного налаштування проекту, та підключення основних компонентів, таких як база даних та пошта отримувача:

APP_READ_ONLY=false

APP_NAME="My Project"

APP_ENV=local

APP_KEY=base64:wM9hxud0ltD1eSS14k1PYOhv3VZFCTZT0yxGILm5pMw=

APP_DEBUG=true

APP_URL=http://localhost:8084

APP_LOCALE=en

APP_FALLBACK_LOCALE=en

APP_LOCALE_PHP=en_US

APP_TIMEZONE=UTC

LOG_CHANNEL=daily

DEBUGBAR_ENABLED=false

SINGLE_LOGIN=false

DB_CONNECTION=mysql

DB_HOST=db

DB_PORT=3306

DB_DATABASE=pr

DB_USERNAME=pr

DB_PASSWORD=pr

BROADCAST_DRIVER=log

CACHE_DRIVER=file

QUEUE_CONNECTION=database

SESSION_DRIVER=file

SESSION_LIFETIME=120

SESSION_ENCRYPT=false

MAIL_DRIVER=smtp

MAIL_HOST=localhost

MAIL_PORT=1025

MAIL_USERNAME=null

MAIL_PASSWORD=null

MAIL_ENCRYPTION=null

MAIL_FROM_ADDRESS=hello@example.com

MAIL_FROM_NAME="${APP_NAME}"

 # Access

ENABLE_REGISTRATION=true

CHANGE_EMAIL=false

PASSWORD_HISTORY=3

PASSWORD_EXPIRES_DAYS=30

 # This should be one or the other, or neither

REQUIRES_APPROVAL=false

CONFIRM_EMAIL=true

Після створення файлу оточення, слід запустити наступну команду: *php artisan key:generate*. Вона згенерує ключ для вашого проекту.

Якщо на комп'ютері встановлений *PHP* локально, і розробник хоче використовувати вбудований сервер розробки *PHP* для обслуговування своєї програми, можна використовувати команду *serve Artisan*. Ця команда запустить сервер розробки за адресою http: // localhost: 8084:

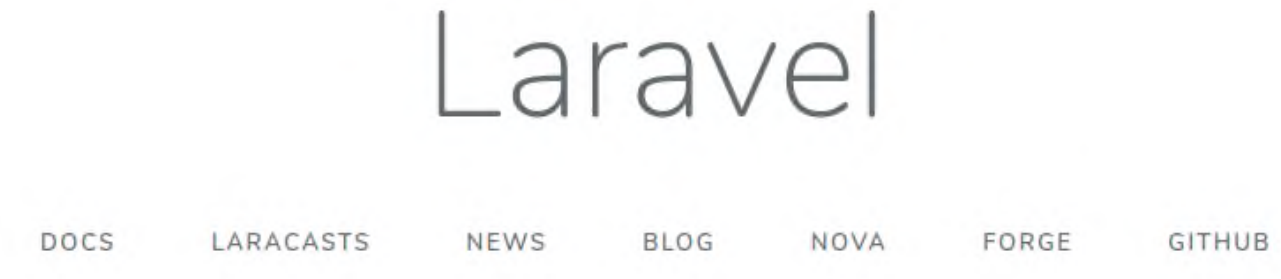

Рис. 3.3. Домашня сторінка згенерованого проекту

Якщо розробник використовує *Nginx*, наступна директива у конфігурації сайту направить всі запити на фронт-контролер *index.php*:

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

При використанні *Homestead* або *Valet URL*-адреси будуть автоматично налаштовані.

3.1.1. Встановлення *Docker* та *Docker Compose*. Налаштування *Laravel* з *Docker*

Завантажити *Docker Engine* можна різними способами, залежно від потреб розробника:

Більшість користувачів налаштовують сховища *Docker* та завантажують із них для зручності встановлення та оновлення. Це рекомендований підхід.

Деякі користувачі завантажують пакет *DEB* та встановлюють його вручну та повністю вручну керують оновленнями. Це корисно в таких ситуаціях, як встановлення *Docker* на локальних системах, що не мають доступу до Інтернету.

У середовищах тестування та розробки деякі користувачі вирішують використовувати автоматизовані зручні сценарії для встановлення *Docker*.

Перш ніж встановити *Docker Engine* на новій машині, потрібно налаштувати сховище *Docker*. Після цього розробник зможе встановити та оновити *Docker* зі сховища.

Кроки для налаштування репозиторію:

Слід оновити індекс пакета apt та встановити пакети, щоб дозволити *apt* використовувати сховище через *HTTPS*:

```
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
```

gnupg-agent \

software-properties-common

Додати офіційний ключ *GPG Docker*:

$curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Наступна команда використовується для налаштування стабільного сховища. Щоб додати нічний або тестовий репозиторій, додайте слово *nightly* або *test* (або обидва) після слова *stable* в командах нижче:

$ sudo add-apt-repository \ "deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb_release -cs) \

stable"

Наступні команди встановлять останню версію *Docker Engine* та *containerd*:

$ sudo apt-get update

$ sudo apt-get install docker-ce docker-ce-cli containerd.io

Щоб завантажити поточний стабільний випуск *Docker Compose* слід запустити наступну команду:

sudo curl -L

"https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Побудова додатків за допомогою *Docker Compose* забезпечує процес налаштування та контролю версії у інфраструктурі системи. Щоб налаштувати додаток *Laravel*, слід створити файл *docker-compose* із визначенням службового веб-сервера, баз даних та додатків.

У файлі *docker-compose* визначаються три служби: додаток, веб-сервер та db. Приклад файлу *docker-compose*:

version: '3'

services:

#PHP Service

app:

build:

```
    context: .
    dockerfile: Dockerfile
  image: digitalocean.com/php
  container_name: app
  restart: unless-stopped
  tty: true
  environment:
    SERVICE_NAME: app
    SERVICE_TAGS: dev
  working_dir: /var/www
  networks:
    - app-network
#Nginx Service
webserver:
  image: nginx:alpine
  container_name: webserver
  restart: unless-stopped
  tty: true
  ports:
    - "80:80"
    - "443:443"
  networks:
    - app-network
#MySQL Service
db:
  image: mysql:5.7.22
  container_name: db
  restart: unless-stopped
  tty: true
  ports:
```

```
    - "3306:3306"
  environment:
    MYSQL_DATABASE: laravel
    MYSQL_ROOT_PASSWORD: your_mysql_root_password
    SERVICE_TAGS: dev
    SERVICE_NAME: mysql
  networks:
    - app-network
#Docker Networks
networks:
  app-network:
    driver: bridge
```

Додаток: це визначення служб містить додаток *Laravel* і запускає персоналізований образ *Docker*, *digitalocean.com/php*. Також оновлення встановлюється для параметрів *working_dir* у контейнері значення */ var / www.*

Веб-сервер: це визначення служб бере образ *nginx: alpine* з *Docker* і відкриває порти 80 і 443.

db: це визначення служб виводить образ *mysql*: 5.7.22 з *Docker* та визначає нові передові середовища, у тому числі базу даних *laravel* для ваших додатків та пароль користувача *root* для баз даних. Можна використовувати будь-яке ім'я баз даних, яке захочете, а також слід замінити *your_mysql_root_password* надійним паролем. Це визначення служб також забезпечує розміщення порт хоста 3306 та порт контейнера 3306.

Кожну властивість контейнера визначає ім'я контейнера, відповідне ім'я служби. Якщо не визначити це ім'я, *Docker* буде привласнювати кожному контейнеру ім'я, що складається з імен історичної особистості та випадкового слова, розділених символами підкреслення.

Для взаємодії взаємодії між контейнерами служби підключаються до з'єднаної мережі з ім'ям *app-network*. Єдина мережа використовує програмний міст, що дозволяє підключити один мережевий контейнер до взаємодії з іншим. Драйвер

моста автоматично встановлює правила хосту, щоб контейнери в різних сполучних мережах не могли бути використані для взаємодії з іншими. Це підвищує рівень безпеки додатків, оскільки взаємодіяти один з одним вони можуть лише через пов'язані служби. Це також означає, що можна задати різні мережі та служби, підключаючись до пов'язаних функцій: наприклад, клієнтські служби можуть використовувати мережу *frontend*, а сервер – мережу *backend*.

3.1.2. Пакет для взаємодії з приладами на основі технології *Z-Wave*.

LaravelWave – надає пакет *Laravel* для інтеграції з існуючим сервером *Z-Wave*.

Найпростіший спосіб додати пакет до проекту це, виконати дану команду:

composer require exposuresoftware/laravel-zway

Пакет можна налаштувати за допомогою змінних середовища. Щоб налаштувати його таким чином, додайте наступні рядки до *.env* у проекті, змінивши значення відповідно до необхідної конфігурації:

```
# The host URI for your Z-Way server.
# Defaults to http://localhost
# Include the schema and do not provide a trailing slash
ZWAY_HOST=http://localhost
# The port on which your Z-Way server is listening.
# Defaults to Z-Way default port 8083.
ZWAY_PORT=8083
# API User
# Defaults to the Z-Way default of admin
ZWAY_USER=admin
```

Існує ряд методів використання *SDK* у проекті. Кожен забезпечує доступ до всіх доступних методів, повний перелік яких міститься в цьому документі.

Весь зв’язок із сервером *Z-Wave*, крім входу в систему, вимагає аутентифікації при доступі через цей *SDK* на даний момент. Після успішного входу на сервер один раз маркер буде додатково збережений.

Таблиця 3.1.

Доступні методи для взаємодії з *api Z-Wave*

Метод	Тип даних, що повертає метод	Опис
hasToken()	*bool*	Метод повертає *true* або *false* в залежності від того має поточний екземпляр токен чи ні
login('admin', 'secret', true)	*bool*	Метод використовується для аутентифікації.
listDevices(true)	*Illuminate\Support\Collection*	Повертає колекцію всіх пристроїв, відомих серверу. Якщо передано значення *false*, вони не зберігатимуться в базі даних.
update(device)	*ExposureSoftware\LaravelWave\Device*	Повертає Пристрій з оновленими атрибутами для відображення поточного стану.
command(device, parameters)	*bool*	Запускає команду на даному пристрої із зазначеними параметрами.
В цій версії підтримується тільки *switchBinary*	-	-

3.2. Створення *UML*-діаграм додатку для керування електроприладами

Діаграма *UML* – це діаграма, заснована на *UML* (уніфікованій мові моделювання) з метою візуального представлення системи разом з її основними акторами, ролями, діями, артефактами або класами, з метою кращого розуміння, зміни, підтримки чи документування інформації про систему.

UML – це абревіатура, що розшифровується як *Unified Modeling Language*. Простіше кажучи, *UML* – це сучасний підхід до моделювання та документування програмного забезпечення. Насправді це один із найпопулярніших методів моделювання бізнес-процесів.

Він базується на схематичному зображенні програмних компонентів. Використовуючи візуальні подання, ми можемо краще зрозуміти можливі недоліки чи помилки програмного забезпечення або бізнес-процесів.

UML був створений в результаті хаосу, який відносився до розробки програмного забезпечення та документації. У 1990-х рр. існувало кілька різних способів представлення та документування програмних систем. Виникла потреба в більш уніфікованому способі візуального представлення цих систем, і в результаті в 1994-1996 роках *UML* був розроблений трьома інженерами-програмістами, що працювали в *Rational Software*. Пізніше він був прийнятий в якості стандарту в 1997 яким залишається і нині, отримавши лише кілька оновлень.

Переважно, *UML* використовується як мова загального призначення для моделювання в галузі програмного забезпечення. *UML*-діаграми забезпечують як більш стандартизований спосіб моделювання робочих процесів, так і ширший спектр функцій для поліпшення розуміння та ефективності.

Існує кілька типів діаграм *UML*, і кожна з них має різну мету, незалежно від того, розробляється вона до впровадження чи після (як частина документації).

Дві найбільш широкі категорії, які охоплюють усі інші типи, – це поведінкова діаграма *UML* та структурна діаграма *UML*. Як випливає з назви, деякі діаграми *UML* намагаються проаналізувати та зобразити структуру системи або процесу,

тоді як інші описують поведінку системи, її суб'єктів та її будівельних компонентів. Існує наступна класифікація діаграм.

Поведінкові *UML*-діаграми:

– Діаграма діяльності;
– Діаграма прецедентів;
– Діаграма огляду взаємодії;
– Часова діаграма;
– Діаграма станів;
– Комунікативна діаграма;
– Діаграма послідовності.

Структурні *UML*-діаграми:

– Діаграма класів;
– Діаграма об’єктів;
– Діаграма компонентів;
– Складена структурна діаграма;
– Діаграма розгортання;
– Діаграма профілю.

Не всі з 14 різних типів діаграм *UML* регулярно використовуються під час документування систем та / або архітектур. Найбільш часто використовуваними при розробці програмного забезпечення є: діаграми прецедентів, класів та діаграми послідовності.

Наріжним каменем системи є функціональні вимоги, яким вона відповідає. Діаграми прецедентів використовуються для аналізу загальних вимог до створюваної системи. Ці вимоги виражаються в різних варіантах використання. Можна виділити три основні компоненти цієї діаграми:

– Функціональні вимоги – представлені як варіанти використання.

– Актори – вони взаємодіють із системою; актор може бути людиною, організацією або внутрішнім чи зовнішнім додатком.

– Відносини між акторами та варіанти використання – представлені за допомогою прямих стрілок.

Діаграма прецедентів додатку для керування електроприладами зображена на рисунку 3.4.

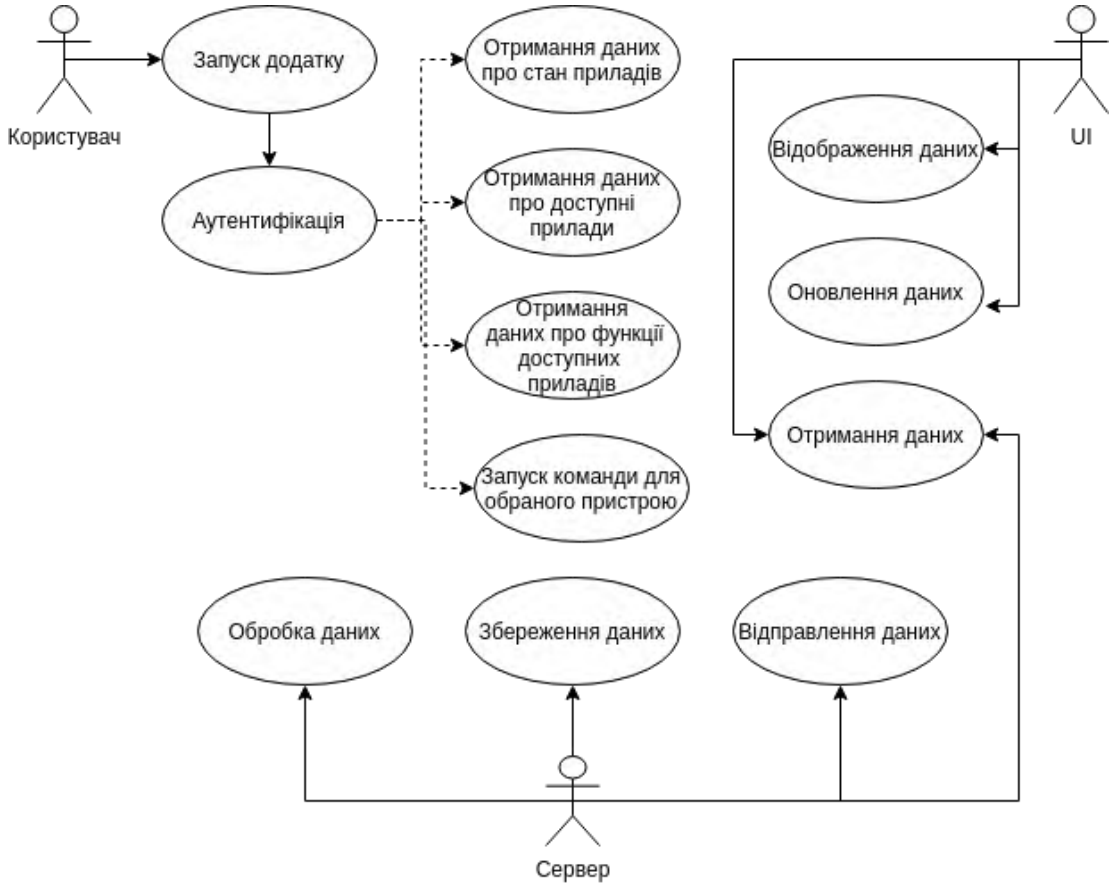

Рис. 3.4. Діаграма прецедентів додатку для керування електроприладами

Діаграми послідовності – це, мабуть, найважливіші діаграми *UML* серед не тільки програмування, а й як моделі дизайну для розробки бізнес-додатків. Останнім часом вони стали популярними у змалюванні бізнес-процесів через їх абсолютно зрозумілу візуальну природу.

Як випливає з назви, схеми послідовностей описують послідовність повідомлень та взаємодій, що відбуваються між акторами та об’єктами. Актори або об'єкти можуть бути активними лише тоді, коли це необхідно або коли інший об'єкт хоче спілкуватися з ними. Вся комунікація представлена в хронологічному порядку.

Структурні схеми використовуються для зображення структури системи. Більш конкретно, вони застосовуються при розробці програмного забезпечення для представлення архітектури системи та того, як різні компоненти взаємопов’язані.

Діаграма послідовності додатку для керування електроприладами зображена на рисунку 3.5.

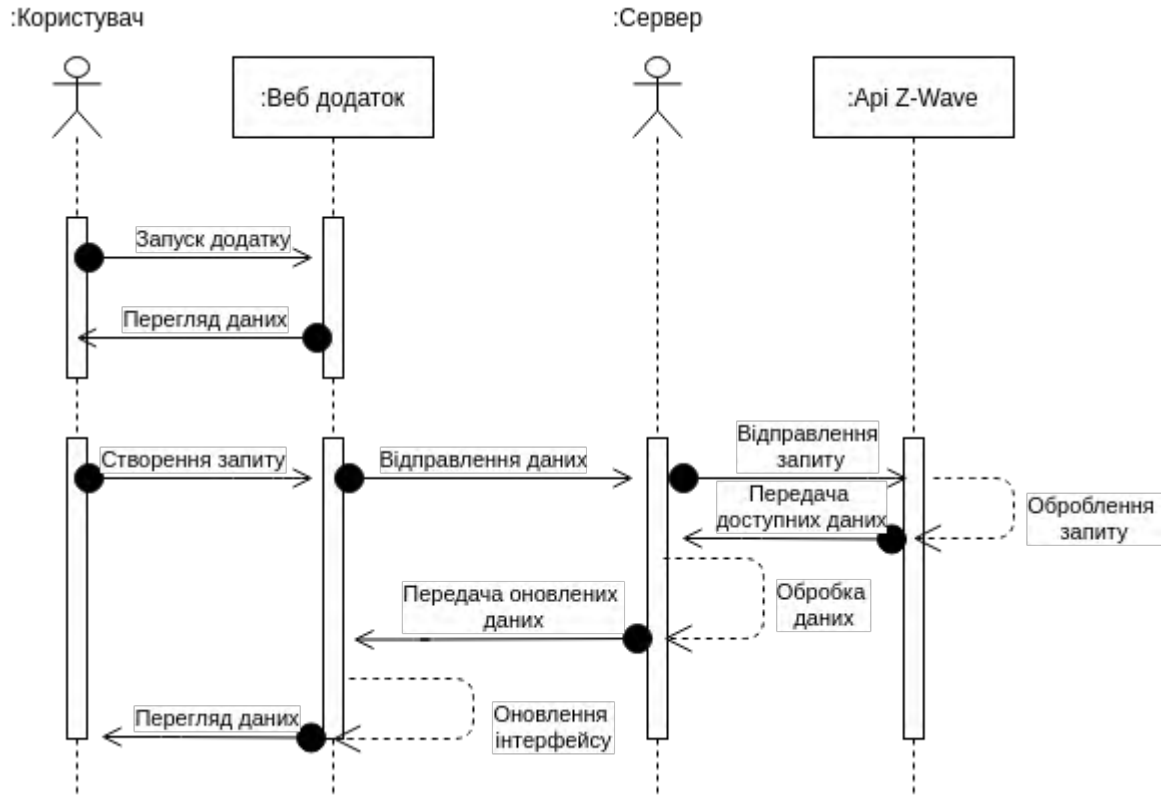

Рис. 3.5. Діаграма послідовності додатку для керування електроприладами

Використання діаграми *UML* для документування процесів та систем може бути дуже корисним. Недоліком є те, що спочатку це може здатися складним. Потрібно вивчити синтаксис, потрібно вибрати, яка діаграма з 14 різних типів є найбільш ефективною для роботи, тощо. Однак, як тільки розробник почне мислити за стандартами *UML*, то він отримає краще розуміння системи яку розробляє.

Зрештою, це може допомогти розробникам виявити недоліки або можливі оптимізації, про які вони, можливо, не думали раніше.

3.3. Опис основних компонентів системи

Для розробки коду веб-додатку для керування електроприладами було обрано *IDE PHPStorm.*

PhpStorm – комерційне крос-платформне інтегроване середовище розробки для *PHP*. Розробляється компаній *JetBrains* на основі платформ *IntelliJ IDEA.*

В основному *PhpStorm* охоплює все, що потрібно для легкої розробки веб-додатків, від якості коду до інтегрованої підтримки *phpunit* та *xdebug*, повних інструментів інтерфейсу, тестування та налагодження, до налаштування середовища локальної розробки за допомогою докера або *vagrant* і однієї з найкращих *vcs* інтеграцій.

Головні переваги:

– Глибоке розуміння всієї бази коду, включаючи завершення коду, рефакторинг, запобігання помилкам та інтегровані інструменти якості;

– Повна інтеграція всіх основних фреймворків та бібліотек *PHP* та *Frontend: Symfony, Laravel, WordPress, Zend 2, React, Angular, JQuery*;

– Вбудовані інструменти якості коду для всього, включаючи підтримку конфігурації редактора;

– Відмінна підтримка композитора;

– Місцева історія *Git*;

– Дійсно легке тестування та налагодження за допомогою вбудованого візуального налагоджувача, який підтримує *PHPUnit, XDebug, Jest, Mocha* тощо;

– Керування та адміністрування баз даних;

– Відмінна підтримка *VCS*. Плагін *Gitlens* є чудовим доповненням за замовчуванням;

– Підтримка *REST Client, Docker, Vagrant*;

– Включає розгортання в локальних та реальних серверних середовищах.

Всі *Jetbrains IDES* – це, по суті, одне і ті ж: *Intellij IDEA*, але з відповідними плагінами, встановленими та повністю налаштованими для певної мови. Таким

чином, *PHPStorm* не має проблем із редагуванням *JAVA*, *Python* або баз даних. Також варто зазначити, що для студентів доступна безкоштовна версія даної *IDE*.

Як вже зазначалось у попередніх розділах, додаток для керування електроприладами буде складатися з таких основних компонентів:

– *User Interface* – код будет створено на базі фреймворку *React*;

– *Server – Laravel Framework*;

– *Database – Mysql*.

Нижче наведено схему алгоритму роботи додатку:

Рис. 3.6. Схема алгоритму роботи додатку

3.3.1. Створення інтерфейсу користувача на базі *React Native*

Для новачків у розробці мобільних пристроїв, найпростіший спосіб розпочати роботу з *React Native* – це *Expo CLI*. *Expo* – це набір інструментів, побудованих навколо *React Native*, і, хоча він має безліч функцій, зараз найбільш актуальною функцією є те, що він може допомогти розробнику написати програму *React Native* за лічені хвилини. Йому знадобиться лише остання версія *Node.js* та телефон або емулятор. Також є можливість спробувати *React Native* безпосередньо у своєму веб-браузері за допомогою технології *Snack*, перш ніж встановлювати будь-які інструменти для розробки.

Якщо користувач вже знайомі з розробкою мобільних пристроїв, можна скористатися *React Native CLI*. Для початку потрібен *Xcode* або *Android Studio*. Якщо на комп'ютері вже встановлено один із цих інструментів, можна запустити проект за кілька хвилин. Якщо інтегроване середовище розробки не встановлене, слід витратити близько 20 хвилин на його встановлення та налаштування.

Для розробки знадобляться *Node*, інтерфейс командного рядка *React Native*, *JDK* та *Android Studio*.

Хоча користувач може використовувати будь-який обраний ним редактор для розробки своєї програми, потрібно буде встановити *Android Studio*, щоб налаштувати необхідні інструменти для створення програми *React Native* для *Android*.

React Native вимагає принаймні 8 версію *Java SE Development Kit (JDK)*. Розробник може завантажити та встановити *OpenJDK* з *AdoptOpenJDK* або із системного пакувальника. Також є можливість завантажити та встановити *Oracle JDK* 14 за бажанням.

Для конфігурації середовища розробки *Android* слід виконати наступні кроки:

Завантажити та встановити *Android Studio*. Перебуваючи в майстрі встановлення *Android Studio*, варто переконатися, що встановлені прапорці поруч із усіма наведеними нижче елементами:

– *Android SDK*;

– *Android SDK Platform*;

– *Android Virtual Device*.

Встановіть *Android SDK*. *Android Studio* за замовчуванням встановлює найновіший *Android SDK*. Однак для створення програми *React Native* з власним кодом потрібен *SDK* для *Android 10* (*Q*). Додаткові *SDK* для *Android* можна встановити через диспетчер *SDK* в *Android Studio*.

Наступним кроком потрібно налаштувати змінну середовища *ANDROID_HOME*. Інструменти *React Native* вимагають налаштування деяких змінних середовища для створення додатків із власним кодом:

export ANDROID_HOME=$HOME/Android/Sdk

export PATH=$PATH:$ANDROID_HOME/emulator

export PATH=$PATH:$ANDROID_HOME/tools

export PATH=$PATH:$ANDROID_HOME/tools/bin

export PATH=$PATH:$ANDROID_HOME/

React Native має вбудований інтерфейс командного рядка, який розробник може використовувати для створення нового проекту. Користувач може отримати до нього доступ, не встановлюючи нічого глобально, використовуючи *npx*, який постачається з *Node.js*. Наступна команда створить новий проект *React Native* під назвою "*Project*":

npx react-native init Project

Для запуску програми *React Native* для *Android* знадобиться пристрій *Android*. Це може бути або фізичний пристрій *Android*, або можна використовувати віртуальний пристрій *Android*, який дозволяє емулювати девайс на вашому комп’ютері.

У будь-якому випадку користувачу потрібно буде підготувати пристрій до запуску програм *Android* для розробки.

Розробнику потрібно буде запустити *Metro*, пакет *JavaScript*, який постачається з *React Native*. Metro "бере вхідний файл та різні параметри та повертає один файл *JavaScript*, що включає весь ваш код та його залежності". – *Metro Docs*

Щоб запустити *Metro*, всередині папки проекту *React Native* виконується команда:

npx react-native start

3.3.2. Розроблення *Api* для взаємодії між сервером та інтерфейсом користувача

Один з найважливіших аспектів розробки якісного та надійного *Api* – це його безпека. *Laravel* надає декілька технологій для захисту маршрутів додатку, однією з яких є технологія *Sanctum*.

Laravel Sanctum забезпечує легку систему аутентифікації для *SPA* (односторінкових додатків), мобільних додатків та простих *API* на основі токенів. *Sanctum* дозволяє кожному користувачеві додатку генерувати декілька маркерів *API* для свого облікового запису. Цим маркерам можуть бути надані можливості, які визначають, які дії маркерам дозволено виконувати.

Sanctum – це простий пакет, який розробник може використовувати для видачі маркерів *API* своїм користувачам без ускладнення *OAuth*. Ця функція натхнена *GitHub* та іншими програмами, які видають "токен особистого доступу". *Sanctum* можна використовувати для створення та управління токенами. Зазвичай вони мають дуже тривалий термін дії (роки), але користувач може в будь-який час скасувати їх вручну.

Laravel Sanctum пропонує цю функцію, зберігаючи маркери користувацьких *API* в одній таблиці бази даних та аутентифікуючи вхідні *HTTP*-запити через заголовок *Authorization*, який повинен містити дійсний маркер *API*.

Sanctum може запропонувати простий спосіб автентифікації односторінкових програм (*SPA*), які потребують зв’язку з *API*, що працює на основі *Laravel*. Ці *SPA* можуть існувати в тому ж сховищі, що і програма *Laravel*, або можуть бути абсолютно окремими сховищами, наприклад створеними за допомогою *Vue CLI* або додатку *Next.js*.

Дана технологія використовує вбудовані служби аутентифікації сеансів на основі файлів *cookie Laravel*. Зазвичай для цього *Sanctum* використовує захист веб-

аутентифікації *Laravel*. Це забезпечує переваги захисту *CSRF*, а також захищає від витоку облікових даних через *XSS*.

Sanctum намагатиметься автентифікуватись за допомогою файлів *cookie* лише тоді, коли вхідний запит надходить із користувацького інтерфейсу *SPA*. Коли він перевіряє вхідний *HTTP*-запит, то спочатку переглядає наявність файлу *cookie* для ідентифікації, а якщо такого немає, *Sanctum* перевіряє заголовок авторизації на наявність дійсного маркера *API*.

Laravel Sanctum можна встановити за допомогою менеджера пакетів *Composer*:

composer require laravel/sanctum

Для публікації конфігурацій слід запустити наступну команду:

Php artisan vendor:publish --provider = "Laravel\Sanctum\SanctumServiceProvider"

Для створення в базі даних таблиці з токенами користувачу слід запустити:

php artisan migrate

Далі, якщо розробник планує використовувати *Sanctum* для аутентифікації *SPA*, йому слід додати проміжне програмне забезпечення *Sanctum* до його групи проміжних програм *api* у файлі програми / *Http* / *Kernel.php* додатка.

Наступним кроком можна створити систему маршрутів *api*. Для додатку керування електроприладами вона буде виглядати наступним чином:

```
Route::post('login', 'API\UserController@login');
Route::group(['middleware' => 'auth:sanctum'], function(){
   Route::post('details', 'API\UserController@details');
   Route::prefix('v1')->group(function(){
      Route::get('getDevices', 'API\UserController@getDevices');
      Route::post('sendCommand', 'API\UserController@sendCommand');
      Route::get('getState', 'API\UserController@getState');
   });
});
```

Маршрути, що відносяться до групи *'middleware' => 'auth:sanctum'* захищені токеном.

Нижче описано приклад методу *api login*:

```
public function login(){
    $client = User::where('email', request('email'))
                where(password, encrypt(request(password)))->first();
    if($client){
     $success['token'] =  $client->createToken('Customer')->plainTextToken;
  return response()->json(['token' => $success['token']], $this->successStatus);
    }
    else{
       return response()->json(['error'=>'Unauthorised'], 401);
    }
  }
```

Для перевірки роботи *Api* було обрано веб-додаток *Insomnia REST Client*.

Insomnia – прекрасний міжплатформенний додаток для організації, запуску та налагодження *HTTP*-запитів.

Переваги:

– Легко перемикається між середовищами з відокремленими змінними;

– Підтримується створення фрагментів коду майже для будь-якої мови, яку захоче користувач;

– Необмежена кількість установок з безкоштовним тарифом;

– Доступно для додатків *Mac / Windows / Linux* та розширення *Google Chrome*;

– Відкрите джерело.

Відправлення запиту на логін користувача:

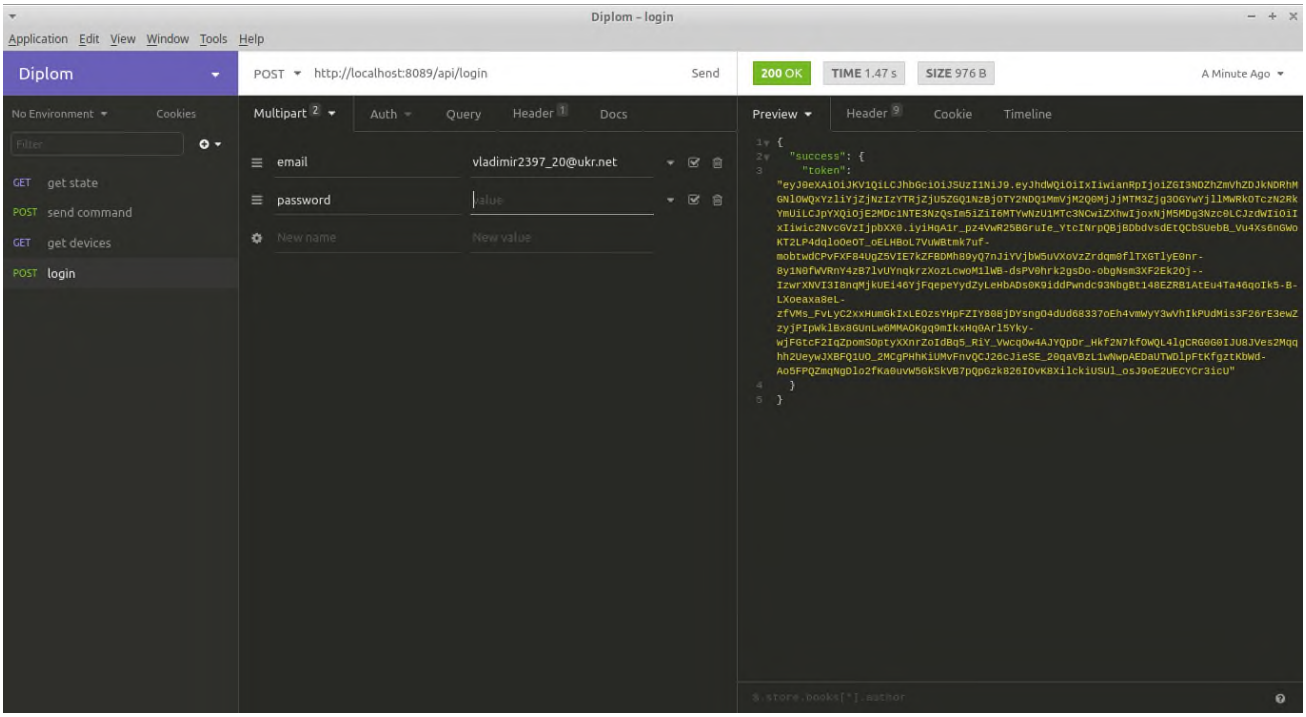

Рис. 3.7. Результати роботи методу *api login*

Метод *login*, якщо відправлені дані вірні, повертає токен, за допомогою якого можна отримати доступ до інших методів *api*.

Відправлення запиту на доступні девайси користувача:

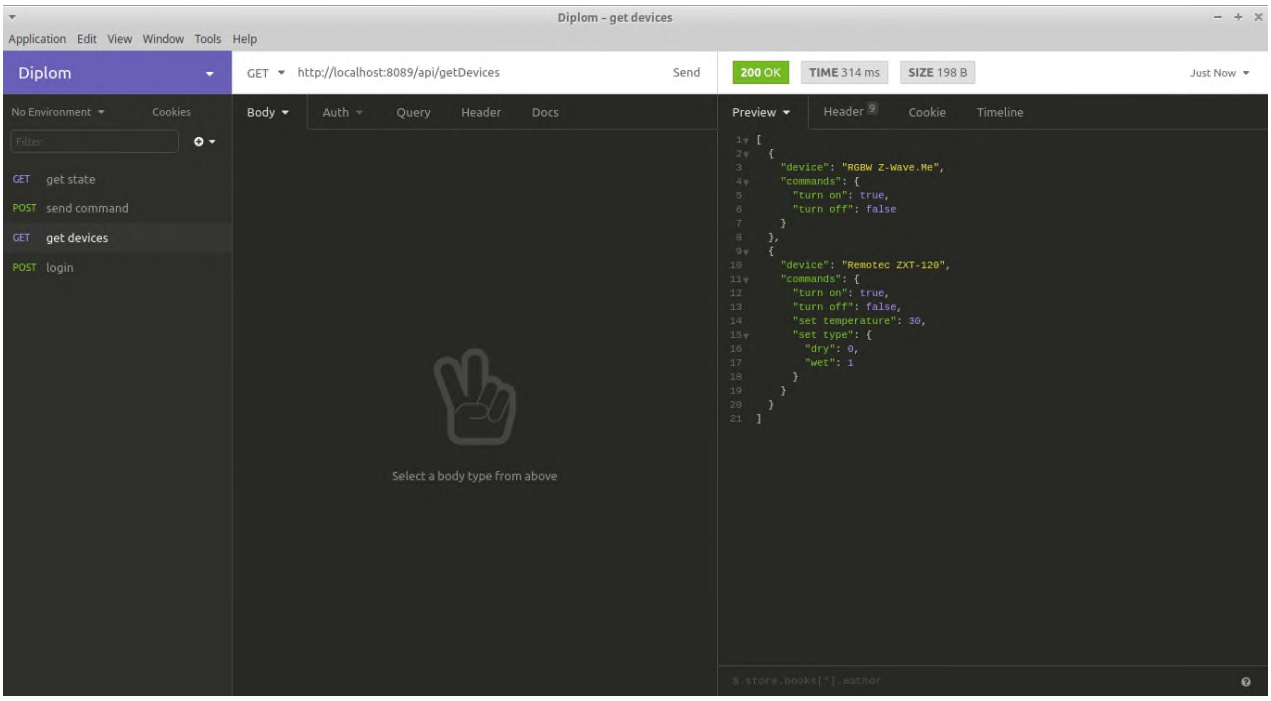

Рис. 3.8. Результати роботи методу *api getDevices*

Отримавши доступні для користувача девайси з командами, з’являється можливість відправляти команди з параметрами, для керування приладами. Так, наприклад, прилад *RGBW Z-Wave.Me* можна увімкнути або вимкнути (команди *turn on/turn off*).

Відправлення запиту на відправлення команд користувача:

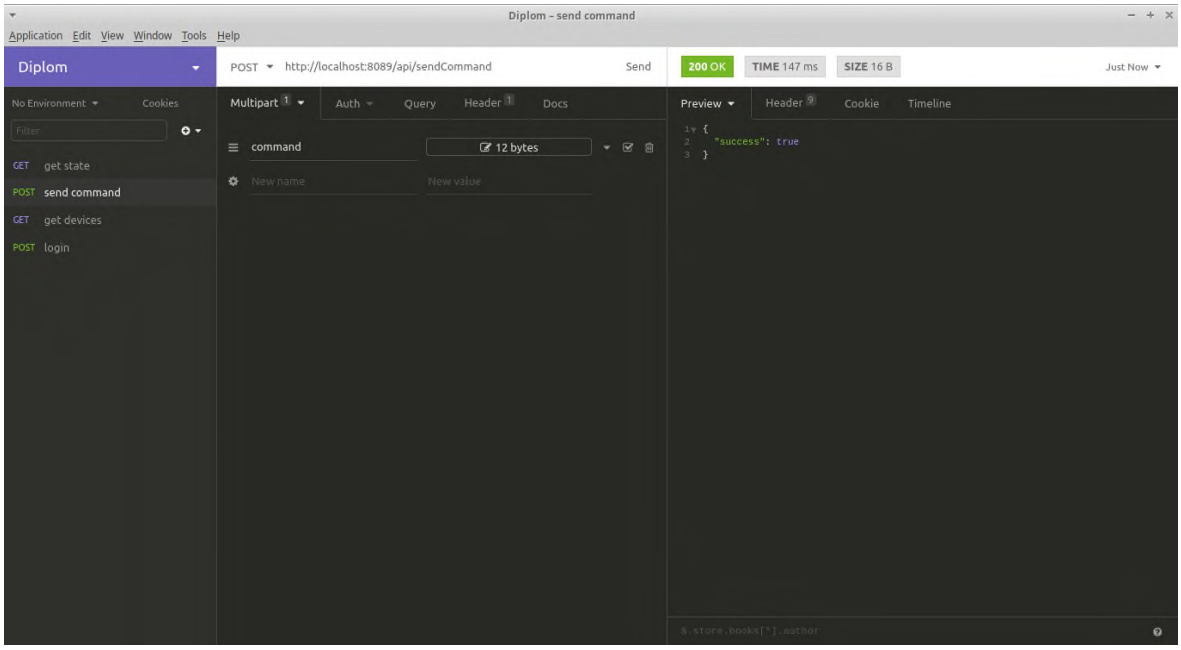

Рис. 3.9. Результати роботи методу *api sendCommand*

3.3.3. Тестування системи

Юніт-тест – це спосіб тестування одиниці – найменшого шматка коду, який можна логічно ізолювати у системі. У більшості мов програмування це функція, підпрограма, метод або властивість. У своїй книзі "Ефективна робота зі застарілим кодом" автор Майкл Фезерс стверджує, що тести не є модульними тестами, коли вони покладаються на зовнішні системи: "Якщо тести спілкуються з базою даних, вони спілкується через мережу, зачіпають файлову систему, конфігурації системи. Такі тести неможливо запустити одночасно з будь-якими іншими."

PHPUnit – це один із найстаріших і найвідоміших пакетів модульного тестування для *PHP*. Він призначений в першу чергу для модульного тестування, що означає тестування коду на найменших можливих компонентах, але він також неймовірно гнучкий і може використовуватися для набагато більших речей, ніж просто модульного тестування.

PHPUnit включає в себе безліч простих і гнучких тверджень, які дозволяють легко протестувати код, який працює дуже добре, коли ви тестуєте певні компоненти. Однак це означає, що тестування більш досконалого коду, такого як контролери та перевірка подання форми, може бути набагато складнішим.

Щоб полегшити роботу розробникам, фреймворк *Laravel PHP* включає колекцію помічників тестування додатків, які дозволяють писати дуже прості тести *PHPUnit* для тестування складних частин додатку.

Першим кроком при використанні *PHPUnit* є створення нового тестового класу. Домовленість для тестових класів полягає в тому, що вони зберігаються в межах ./*tests*/ у каталозі вашої програми. Усередині цієї папки кожен тестовий клас називається <*name*> *Test.php*. Цей формат дозволяє *PHPUnit* знаходити кожен тестовий клас – він буде ігнорувати все, що не закінчується на *Test.php*.

У розгорнутому додатку *Laravel* можна помітити два файли в каталозі ./*tests*/:

Feature / ExampleTest.php ma Unit / ExampleTest.php: ExampleTest.php – це приклад тестового класу, який включає базовий тестовий приклад із використанням помічників тестування додатків.

TestCase.php: файл *TestCase.php* є завантажувальним файлом для налаштування середовища *Laravel* в рамках тестів. Це дозволяє використовувати фасади Laravel і забезпечує основу для помічників тестування.

Для створення нового тестового класу можна створити новий файл вручну або запустити корисну команду *Artisan make: test*, надану *Laravel*.

Для того, щоб створити тестовий клас під назвою *BasicTest*, користувачу потрібно просто виконати ці команди:

$ cd phpunit-test

$ php artisan make:test BasicTest

Перш ніж вперше запустити тестовий пакет, варто вказати дані у файлі phpunit.xml, який за замовчуванням надає *Laravel*. *PHPUnit* автоматично шукатиме файл із назвою *phpunit.xml* або *phpunit.xml.dist* у поточному каталозі під час його запуску. Тут користувач налаштовує конкретні параметри для своїх тестів.

У цьому файлі є багато інформації, однак, найважливішим розділом, є визначення каталогу *testsuite*:

```
<?xml version="1.0" encoding="UTF-8"?>
. . .
    <testsuites>
        <testsuite name="Unit">
            <directory suffix="Test.php">./tests/Unit</directory>
        </testsuite>
        <testsuite name="Feature">
            <directory suffix="Test.php">./tests/Feature</directory>
        </testsuite>
    </testsuites>
. . .
</phpunit>
```

Ця конфігурація повідомляє *PHPUnit* про те, що слід запускати тести, знайдені в каталогах ./*tests*/*Unit* та ./*tests*/*Feature*.

Приклад тесту створеного для додатку керування електроприладами:

```
class CustomerTest extends TestCase
{
    use RefreshDatabase;
    public function testCreateCustomer()
    {
        factory(Customer::class)->create();
        $this->assertCount(1, Customer::all());
    }
    public function testDeleteCustomer()
    {
        $customer = factory(Customer::class)->create();
        $customer->delete();
        $this->assertDeleted($customer);
```

```
    }
}
```

3.3.4. Результати роботи

Результати роботи додатку для керування електроприладами відобрежено на рисунках 3.10 – 3.16.

Панель авторизації користувача:

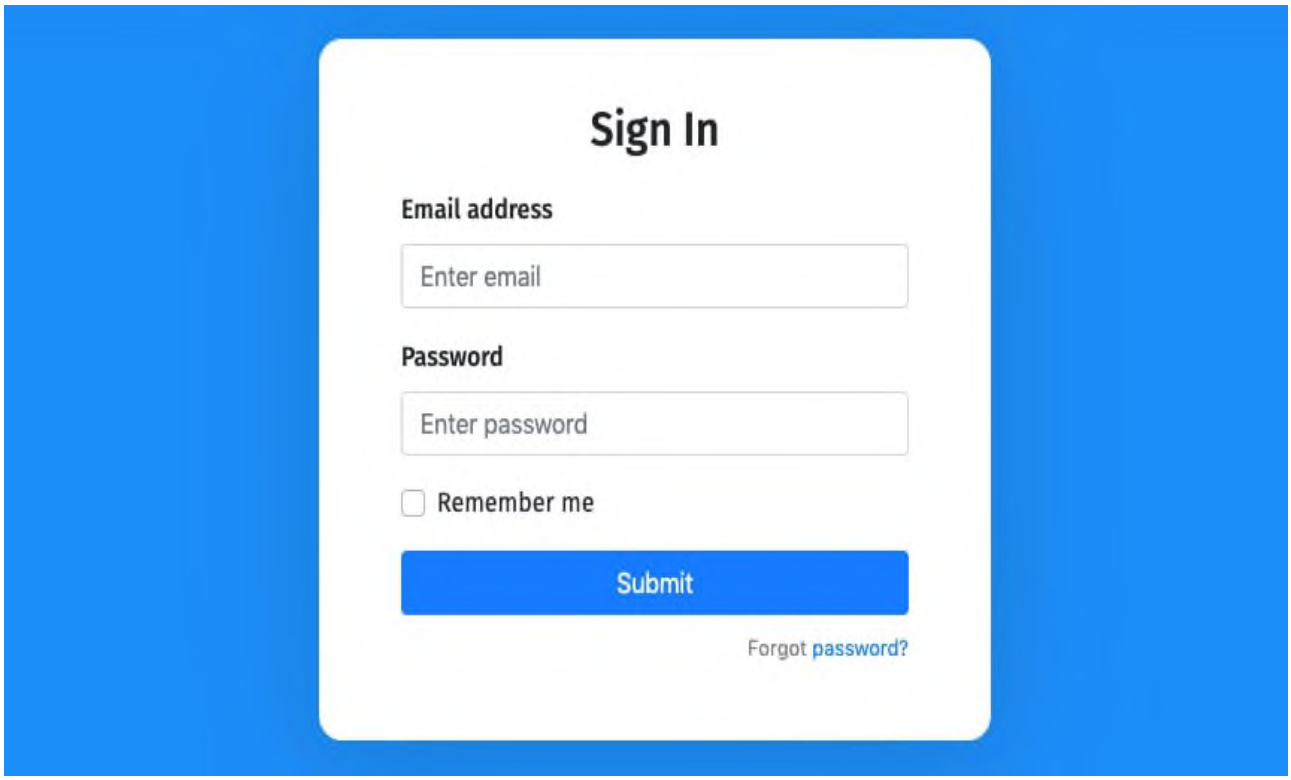

Рис. 3.10. Панель авторизації

Головна панель додатку, на якій виводяться всі доступні прилади та їх функції:

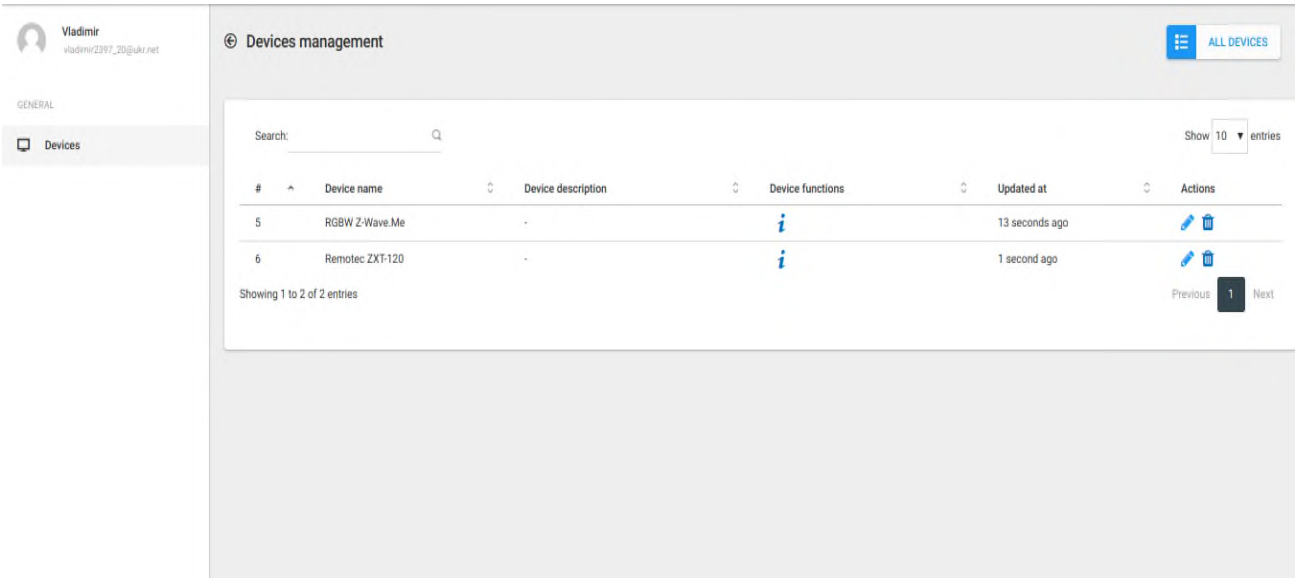

Рис. 3.11. Панель приладів додатку

На рисунках 3.12,3.13 зображено функції приладів:

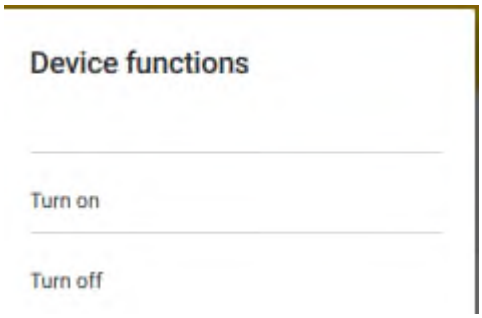

Рис. 3.12. Функції приладу *RGBW Z-Wave.ME*

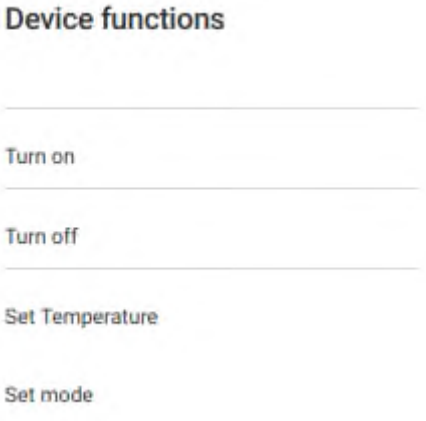

Рис. 3.13. Функції прилада *Remotec ZXT-120*

Вікно видалення даних:

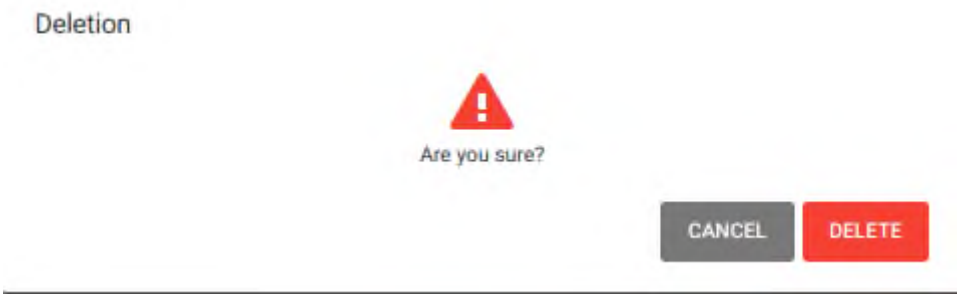

Рис. 3.14. Вікно видалення даних

Панелі зміни функцій зображено на малюнках 3.15, 3.16:

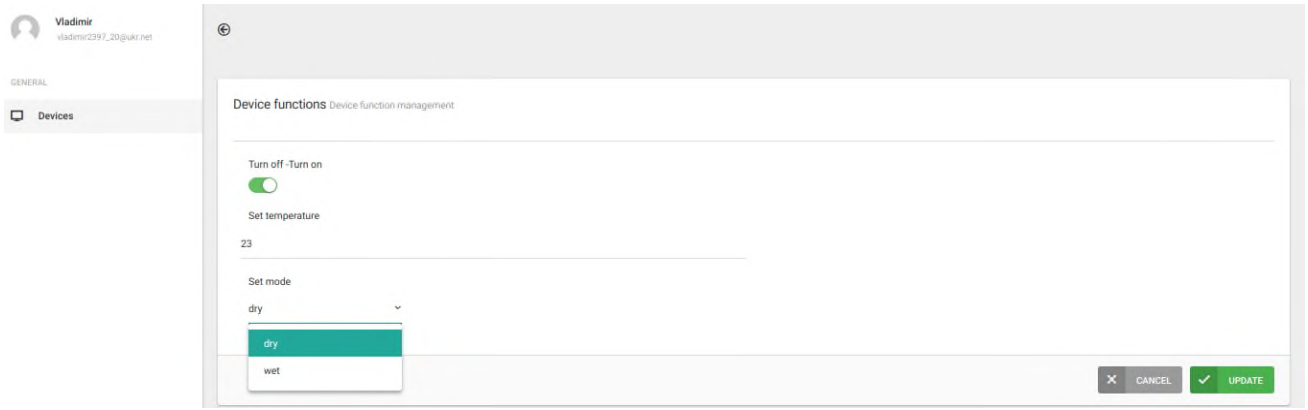

Рис. 3.15. Панель зміни функцій приладу *RGBW Z-Wave.ME*

Рис. 3.16. Панель зміни функцій приладу *Remotec ZXT-120*

3.4. Висновки до розділу

В даному розділі описано процес розробки додатку для керування електроприладами на базі фреймворку *Laravel, React Native* у середі розробки PhpStorm. Розглянуто основні компоненти розроблюваної системи для визначення критеріїв продуктивності: *Composer, Docker, Mysql, RestApi*. Створено *UML*-діаграми, описано основні класи та алгоритм роботи системи.

Розроблено та наведено фрагменти коду програмних компонентів системи для отримання та відображення даних щодо отримання даних по доступним для системи приладам та їх функціям. Створено зручний, простий та зрозумілий *UI*. Створено можливість динамічно змінювати стан приладів. Усі зміни зберігаються до бази даних. Також систему протестовано за допомогою технології *PhpUnit*

ВИСНОВКИ

Під час виконання дипломної роботи було здійснено аналіз існуючих додатків і систем для дистанційного керування електроприладами. Було розглянуто основні сучасні технології для домашньої автоматизації, їх можливості, а також описано основні варіанти їх використання. Для розробки веб-додатку у дипломній роботі було обрано трирівневу клієнт-серверну архітектуру. Для серверної частини було обрано використовувати популярний нині *PHP*-фреймворк *Laravel*. Для клієнтської сторони було обрано прогресивний *js*-фреймворк *React Native*. Для роботи з даними було обрано реляційну базу даних *Mysql*.

Було досліджено технологію автоматизації *Z-Wave* – на сьогоднішній день найбільш надійну і зручну бездротову технологію для будівництва розумного будинку. Також для розробки було використано спеціальну *Laravel*-бібліотеку – *LaravelWay* для взаємодії з *api Z-Wave*.

Були розглянуто технології контейнеризації – *Docker* та *Docker Compose* для локальної розробки, розгортання та запуску додатку.

Було розглянуто функціональні особливості та алгоритми роботи додатку для дистанційного керування електроприладами.

Було досліджено технології *RestfulApi* для взаємозв'язку між сервером та клієнтською частиною розроблюваного додатку тексту.

Було проведено тестування додатку за допомогою технології *PhpUnitTesting*.

Було розроблено схему алгоритму роботи додатку, а також *UML*-діаграми прецедентів та послідовності.

Було продемонстровано *user interface* додатку: його основні панелі, дизайн кожної сторінки.

Запропоноване рішення веб-додатку відрізняється від вже існуючих тим, що є абсолютно безкоштовним, може бути використане на будь-якій платформі, та є відносно дешевим у своїй реалізації .

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. «*Getting started with Laravel – collection of programming examples*». 2016. – [Електронний ресурс]. – Режим доступу: *http://creative-punch.net/articles/php-articles/laraveltutorials/*

2. *Frank Zametti «React Native in Action». Apress,* – 2018. – 327 *с.*

3. ДСТУ ГОСТ 3008-95 «Документація. Звіти у сфері науки і техніки. Структура і правила оформлення.»

4. *Nader Dabit «React Native in Action». Manning Shelter Island,* – 2019. – 309 *с.*

5. *Bonnie Eisenman «Learning React Native». Apress,* – 2018. – 227 *с.*

6. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.

7. Файлер М., Скотт К. «*UML*. Основы». –М.: Вильяме, – 2002. – 192 с.

8. *Ian Molyneaux*. «*The Art of Application Performance Testing*». *O'Reilly Media, – 2014. – 241 с.*

9. Філдінг Р. *Architectural Styles and the Design of Network-based Software Architectures*. / Р. Філдінг. – К.: «Саміт», 2018. – 520 с.

10. ГОСТ 2.106–96 ЕСКД “Текстовые документы”.

11. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.

12. ГОСТ 19.701–90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

13. *Conway R.W., Maxwell W.L., Miller L.W. Theory of Scheduling. Addison-Wesley, Reading, MA*. 1967.

14. «*Getting started with React»*. 2018. – [Електронний ресурс]. – Режим доступу: *https://reactjs.org/docs/getting-started.html*

15. «*What is Rest?».* 2018– [Електронний ресурс]. – Режим доступу*: https://restfulapi.net/*

16. *Dr. Christian Paetz «Z-Wave Basics: Remote Control in Smart Homes». CreateSpace Independent Publishing Platform* , – 2013. – 300 *c.*

17. *Gantt H.L., ASME Transactions,* 1903, 24, P. 1322–1336.

18. Дронов В., «*Laravel*. Быстрая разработка современных динамических *Web*-сайтов на *PHP, MySQL, HTML* и *CSS»*. –БХВ Петербург, – 2016. – 768 с.

19. Меєр Е., «*CSS*. Карманный справочник*»*. –Диалектика, – 2020. – 208 с.

20. Роббинс Д., «*HTML* 5. Карманный справочник*»*. –Диалектика, – 2020. – 192 с.

21. Фленаган Д., «*JavaScript*. Карманный справочник*»*. –Диалектика, – 2020. – 320 с.

ДОДАТОК А

Лістинг коду програмного модулю

app/Http/UserController:

```
<?php

namespace App\Http\Controllers\API;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use ExposureSoftware\LaravelWave\Zwave\Zwave;
use Illuminate\Support\Facades\Auth;
use Validator;

class UserController extends Controller
{
    public $successStatus = 200;
    public $errorStatus = 500;

    /** @var Zwave */
    private $zwave;

    public function __construct(Zwave $zwave) {
        $this->zwave = $zwave;
    }

    public function login(){
        if(Auth::attempt(['email' => request('email'), 'password' => request('password')])){
            $user = Auth::user();
            $success['token'] =  $user->createToken('User')->accessToken;
```

```
            $success['z-wave'] =   $this->zwave->login();
            return response()->json(['success' => $success], $this->successStatus);
        }
        else{
            return response()->json(['error'=>'Unauthorised'], 401);
        }
    }

    public function getDevices(){
        $devices = $this->zwave->listDevices();

        return response()->json(['success' => $devices], $this->successStatus);
    }

    public function sendCommand(Request $request){
        try{
            $this->zwave->command($request->device, $request->command,$request->parameters);
            return response()->json('success', $this->successStatus);
        }
        catch(\Exception $e){
            return response()->json($e->getMessage(), $this->errorStatus);
        }
    }

    public function getState(Request $request){
        try{
            $this->zwave->update($request->device);
            return response()->json('success', $this->successStatus);
        }
```

```
        catch(\Exception $e){
            return response()->json($e->getMessage(), $this->errorStatus);
        }
    }

    public function details()
    {
        $user = Auth::user();
        return response()->json(['success' => $user], $this-> successStatus);
    }
}
```