МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.
«_____»_____2021 р.

ДИПЛОМНИЙ ПРОЄКТ

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ
"БАКАЛАВР"

Тема: Програмно-апаратний модуль управління безпекою в системі *Smart House*

Виконавець: Шудря В.Д.

Керівник: Марченко Н.Б.

Нормоконтролер: Тупота Є.В.

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет Кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп’ютеризованих систем управління
Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Литвиненко О.Є.
«_____»_____2021 р.

ЗАВДАННЯ
на виконання дипломної роботи (проєкту)
Шудря Віталій Дмитрович
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема дипломної роботи (проєкту): Програмно-апаратний модуль управління безпекою в системі *Smart House*
затверджена наказом ректора від «4» лютого 2021 р. № №135/ст
2. Термін виконання роботи (проєкту): з 17.05.2021 по 20.06.2021
3. Вихідні дані до роботи (проєкту): державні стандарти України, тема дипломного проєкту, технічне завдання
4. Зміст пояснювальної записки:
    1) аналіз предметної області та постановка задачі;
    2) опис технологій для розробки та проєктування системи;
    3) опис процесу розробки та основних модулів системи.
5. Перелік обов'язкового графічного (ілюстративного) матеріалу:
    1) архітектура системи;
    2) діаграма станів апаратного забезпечення;
    3) діаграма станів блоку управління;
    4) *use-case* діаграма мобільного додатку.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Проаналізувати літературу по темі дипломної роботи	17.05.2021-20.05.2021	
2	Провести аналіз існуючих рішень та розробити постановку завдання	21.05.2021-24.05.2021	
3	Спроєктувати систему та обрати необхідні для розробки технології	25.05.2021-28.05.2021	
4	Розробити систему за проєктом	29.05.2021-05.06.2021	
5	Оформити пояснювальну записку	06.06.2021-14.06.2021	
6	Оформити графічний та ілюстративний матеріал	15.06.2021-20.06.2021	

7. Дата видачі завдання: «17» травня 2021 р.

Керівник дипломної роботи (проєкту) _____ Марченко Н.Б.
(підпис керівника) (П.І.Б.)

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

РЕФЕРАТ

Пояснювальна записка до дипломного проєктування “Програмно-апаратний модуль управління безпекою в системі *Smart House*”: 60 с., 16 рис., 15 літературних джерел, 1 додаток.

Ключові слова: *SMART HOUSE, IOT*, БЕЗПЕКА, СИСТЕМА, ДАТЧИК, ОХОРОНА.

Об'єкт дослідження: процес створення системи управління безпекою.

Предмет дослідження: програмно-апаратний модуль безпеки в системі *Smart House*.

Мета дипломного проєктування: реалізація програмного та апаратного забезпечення модуля управління безпекою для підвищення якості охорони та безпеки життя людини.

Методи дослідження: *IDE Visual Studio* 2019, *C#* 8.0, *ASP.NET Core*, *ASP.NET Core MVC*, *Entity Framework Core*, *LINQ*, *JavaScript ES*6, *HTML* 5, *CSS* 3, *Ajax*, *JQuery*, *PostgreSQL*, *Arduino UNO*, *Arduino IDE*, *C++*, *Java*, *IDE Android Studio*, *HTTP*, *REST*, *WEB API*.

В даному дипломному проєкті був проведений аналіз існуючих систем управління безпекою та було спроєктовано і реалізовано програмно-апаратний модуль безпеки. Модуль надає користувачу простоту в конфігурації та масштабування системи, відмовостійкий та оперативний комплекс оповіщення і обробки даних навколишнього середовища за допомогою датчиків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

IOT (Internet of Things) – Інтернет речей

HTTP (Hyper Text Transfer Protocol) – Протокол передачі гіпертексту

БД – База Даних

СУБД – Система Управління Базами Даних

WEB – Всесвітня павутина

FCM (Firebase Cloud Messaging) – Модуль для відправки безкоштовних повідомлень

ВСТУП

Проблема охорони і безпеки життя людини є найгострішою в будь-якому проміжку історії людства. Вторгнення зловмисників на приватну територію, пожежі і затоплення в приміщеннях, витіки токсичних речовин та інші катастрофи – все це загроза життю або майну.

Всі ці сценарії відбуваються через відсутність оперативної інформації, не вчасного виконнання певного алгоритму дій та раніше не помічених деталей, таких яких:

– пошкоджена ізоляція провідника – пожежа;

– пошкодження в критичних точках сантехніки – затоплення будівлі;

– відсутність контролю закритості приватного сектора – вторгення;

– витіки вогненебезпечних речовин – вибух.

Більшість критичних ситуацій неможливо передбачити людиною, але завжди є можливість заздалегідь визначити причину або оперативно відреагувати на наслідки. Для цього існують охоронні служби та комплекси безпеки.

Охоронні служби дозволяють оперативно відреагувати на наслідки та виконати алгоритм дій для зменшення збитків. Але основною проблемою даних служб є те, що вони здатні тільки, переважно, реагувати на наслідки.

Для запобігання та виявлення причин критичних ситуацій існують комплекси аналізу і обробки даних – системи управління безпекою. Такі системи складаються з блоку оброки та збору даних – контролер, та приладів вимірювання – датчиків. За допомогою таких модулів з'являється можливість аналізу і збору даних всіх змінних середовища:

– рівень води в приміщенні;

– температура;

– рух в будівлі і відкриття дверей.

Мета дипломного проєкту – реалізація програмного та апаратного забезпечення модуля управління безпекою для підвищення якості охорони та безпеки життя людини.

Галузь застосування – побутова сфера життя людини, підприємства та виробництва, охоронні служби.

Структура та зміст теоретичної та практичної частини дипломної роботи. Дипломна робота складається зі вступу, трьох розділів та висновків до них.

У першому розділі проведено аналіз предметної області та теоретичних основ побудови систем безпеки, складові та загальні характеристики систем. Також був проведений аналіз існуючих рішень та були визначені основні недоліки.

Модуль безпеки в системі *Smart House* повинен мати наступні функціональні можливості:

– запобігання пожежі;

– запобігання затоплення будинку;

– реєстрація високого рівня токсичних газів (чадний газ, пропан, бутан);

– запобігання несанкціонованого доступу;

– надсилання швидких повідомлень на клієнт.

Архітектурно система повинна складатись з таких рівнів:

– набір вимірювальних приладів (датчики);

– прилад контролю и реєстрації даних (контролер *Arduino*);

– блок управління та обробки даних (*Rasbepry PI* або персональний комп’ютер);

– програмний модуль (*Web*–сервер);

– база даних (*SqlLite*);

– мобільний додаток.

Другий розділ представляє собою опис архітектури програмно-апаратного модуля та перелік і опис використовуваних технологій. Для усіх рівнів системи були обрані наступні технології розробки:

– апаратне забезпечення – *Arduino UNO*, *Arduino IDE*, *C++*;

– блок управління – *C#*, *ASP.NET Core*, *ASP.NET Core MVC*, *Entity Framework Core*, *SQLite*, *HTML* 5, *CSS* 3, *JavaSciprt*;

– мобільний додаток – *Android Studio*, *Java*, *FCM*.

Третій розділ присвячений процесу створення програмно-апаратного комплексу та пояснення роботи основних його компонентів.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ РІШЕНЬ У СФЕРІ УПРАВЛІННЯ БЕЗПЕКОЮ І ПОСТАНОВКА ЗАДАЧІ ДИПЛОМНОГО ПРОЄКТУВАННЯ

1.1. Система *Smart House*

Smart House - це автоматизована система управління приладами в будинку, які об'єднані в єдину екосистему (*IoT*). Система може без втручання користувача приймати рішення і виконувати заздалегіть задані алгоритми дій.

Система *Smart House* розпізнає події, які відбуваються в будинку, реєструє та відповідним чином реагує. Кожна з підсистем може контролювати поведінку інших за допомогою заздалегідь розробленими алгоритмами. Перевагою *Smart House* є об'єднання інших підсистем в єдину систему управління.

Система управління дозволяє створювати будь–які процедури функціонування, тому що всі елементи цієї системи можуть бути синхронізовані. Основні типи реалізацій:

– управління безпекою;

– контролю всіх змінних систем та реєстрація їх критичних змін;

– управління будинком за домопомогою віддаленого доступу.

Набір обладнання, що входить в розумний будинок, залежить від моделі і призначення системи. Але є кілька елементів, які присутні в будь-якій комплектації. Контролер – центр управління, який з'єднує всі частини системи один з одним. Датчики – приймають сигнали з навколишнього середовища і спрацьовують, коли відбувається певна подія.

1.2. Модуль управління безпеки в системі *Smart House*

Комплекс приладів сигналізації – основа модуля безпеки, прилади вимірювання та спостереження дозволяють відреагувати на вторгнення у будівлю та критичне значення датчиків, при цьому спрацює звукова сирена, яка сповістить користувача повідомленням на мобільний додаток.

До модуля безпеки можна підключити такі прилади вимірювання та спостереження:

– інфрачервоний датчик руху – виявляє рух в приміщенні за допомогою інфрачервоного сигналу;

– віброзвуковий датчик – реєструє розбиття скла та ішні джерела шуму;

– датчик відкриття дверей – реєструє проникнення в будівлю;

– прилад вимірювання рівня води – реагує на затоплення в будинку;

– датчик токсичних газів – реєструє витіки газів (*H*, *CH4*, *C3H8*, *C4H10*, *C2H6O*);

– датчик диму – реєструє чадний газ (*CO*) при задимленні приміщення внаслідок пожежі;

– інфрачервоний бар'єр – встановлюються по периметру території.

1.3. Аналіз існуючих систем управління безпекою

Багато систем безпеки пропонують компоненти, які працюють разом у домашньому середовищі та ними можна керувати за допомогою індивідуальних правил.

Майже всі професійні системи дозволяють використовувати ваш смартфон як командний центр для постановки та зняття системи з охорони, створення правил, додавання та видалення компонентів та отримання *push*–сповіщень, коли спрацьовують сигнали тривоги.

Системи зарубіжних виробників: *Vivint*, *SimpliSafe*, *Frontpoint*, *ADT*, *Blue by ADT*, *Abode*, *Cove*.

Системою українських виробників є *Ajax*.

1.3.1. *Vivint*

На рис. 1.1 представленний загальний вигляд системи *Vivint.*

Рис. 1.1. Система *Vivint*

Vivint відомий своєю технологією домашньої автоматизації, але окремої уваги заслуговують інноваційні функції домашньої безпеки від *Vivint*.

Поєднання технологій штучного інтелекту та високотехнологічного обладнання для безпеки, виводить *Vivint* поза межі середньої системи домашньої безпеки для вирішення найгостріших проблем безпеки.

Vivint має на меті виявляти зловмисників заздалегідь, перш ніж вони потраплять до ваших вхідних дверей – це виводить домашню безпеку на новий рівень.

Vivint забезпечує охорону з усією підтримкою та повним спектром послуг - професійна установка, консультації з домашньої охорони та цілодобовий професійний контроль. Але, як і будь-який професійний продукт, він поставляється за величезною ціною.

1.3.2. *SimpliSafe*

На рис. 1.2 представленний загальний вигляд системи *SimpliSafe.*

Рис. 1.2. Система *SimpliSafe*

SimpliSafe здійснює домашню безпеку на власних умовах. Це означає відмовитися від деяких речей, які відвертають клієнтів від охоронних компаній: зухвалі продавці та довгі контракти.

За допомогою *SimpliSafe* ви можете вибрати необхідне обладнання та рівень моніторингу. Він пропонує Інтернет-магазини та професійні послуги моніторингу.

1.3.3. *Frontpoint*

На рис. 1.3 представленний загальний вигляд системи *Frontpoint.*

Рис. 1.3. Система *Frontpoint*

Frontpoint найкраще підходить для тих, кому подобається безкомпромісний щодо варіантів обладнання або функцій підхід *DIY* та незалежність системи, *Frontpoint* відповідає всім необхідним.

На відміну від інших виробників *DIY* домашньої безпеки, які мають обмежені можливості (наприклад, жодної зовнішньої охоронної камери),

Frontpoint пропонує майже кожен компонент, який може існувати в системі безпеки. У базовій комплектації *Frontpoint* включає набір всіх можливих приладів вимірювання, наприклад моніторинг чадного газу (*CO)* та детектор диму.

Основною перевагою *Frontpoint* є відсутність будь-яких контрактів і ви маєте повний контроль над тим коли та як встановлюється система безпеки.

1.3.4. *ADT*

На рис. 1.4 представленний загальний вигляд системи *ADT*.

Рис. 1.4. Система *ADT*

ADT надає експертні консультації з питань безпеки та професійну установку. Крім того, *ADT* має найбільшу кількість центрів моніторингу, ніж будь-яка компанія з охорони будинку.

ADT пропонує угоди, які допомагають скоротити витрати на обладнання, що значно зменшує кількість загальних витрат на 36-місячний контракт.

Що стосується моніторингу домашньої безпеки, ніхто не може похвалитися досвідом як у *ADT*. Ця компанія, що займається охороною житла впродовж майже 145 років, забезпечує першокласний захист житла та захищає понад 8000000 людей по всій планеті.

1.3.5. *Blue by ADT*

На рис. 1.5 представленний загальний вигляд системи *Blue by ADT*.

Рис. 1.5. Система *Blue by ADT*

Blue by ADT пропонує найкраще з обох боків – лекгу *DIY* безпеку підтриману надійним професійним моніторингом від *ADT*.

Blue by ADT – останній випуск лідера безпеки *ADT*. На відміну від традиційних пропозицій *ADT*, додалась можливість отримати послугу без довгострокового контракту.

Ця система була створена у 2020 році завдяки розумним камерам, додатковому самоконтролю та безкоштовному мобільному додатку.

1.3.6. *Adobe*

На рис. 1.6 представленний загальний вигляд системи *Adobe*.

Рис. 1.6. Система *Adobe*

Abode – це єдина система безпеки, яка працює з *Amazon Alexa*, *Google Assistant*, *Apple HomeKit*, *Z-Wave* та *Zigbee*.

Abode має можливість налаштування домашньої безпеки за допомогою голосового управління через *Amazon Alexa*, *Google Assistant* або *Apple HomeKit*.

Крім голосового управління є також можливість додавання розумного освітлення, розумних замків та розумних детекторів диму та *CO*.

1.3.7. *Cove*

На рис. 1.7 представленний загальний вигляд системи *Cove*.

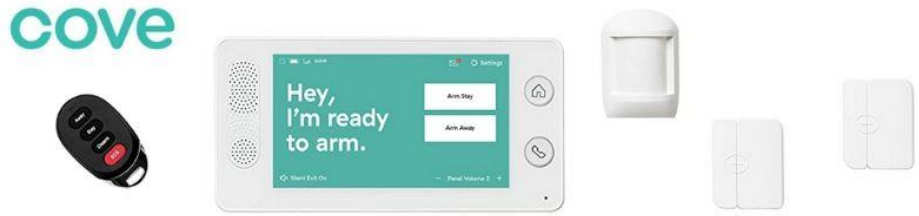

Рис. 1.7. Система *Cove*

План моніторингу *Cove* дозволяє керувати системою безпеки за допомогою мобільного додатка, включаючи відео моніторинг та інтеграцію смарт-пристроїв. Крім того, *Cove* дає довічну гарантію на свою систему.

1.3.8. *Ajax Systems*

На рис. 1.8 представленний загальний вигляд системи *Ajax Systems*.

Рис. 1.8. Система *Ajax Systems*.

Ajax Systems – міжнародна технологічна компанія, головний офіс і виробничі потужності якої розташовані в Києві.

Основний продукт – професійна бездротова система безпеки *Ajax*. Станом на 2020 рік система складається з 27 пристроїв для захисту від пограбування, пожежі і затоплення, а також пристроїв для управління електроживленням. Датчики працюють до 7 років від комплектних батарей. Для зв'язку пристрої системи використовують розроблений компанією пропріетарний двосторонній радіопротокол *Jeweller*.

Jeweller – технологія двостороннього радіозв'язку, яка працює на частотах 868,0-868,6 *МГц* з дальністю зв'язку до 2000 метрів.

Принцип роботи системи – хаб є керівним пристроєм системи - централлю. Він працює від електромережі і облаштований резервним акумулятором – це забезпечує до 16 годин автономної роботи. До хабу по радіопротоколу *Jeweller* можна під'єднати до 150 будь-яких пристроїв *Ajax*: охоронні, протипожежні датчики і датчики затоплення, сирени, клавіатури і брелоки, пристрої управління електроживленням.

Після отримання сигналу тривоги від датчика хаб включає сирени і миттєво сповіщає користувачів і охоронну компанію. Крім того, хаб реагує на відключення живлення, спроби глушіння радіоефіру, демонтаж або втрату зв'язку з пристроєм.

1.4. Аналіз недоліків існуючих рішень

Проаналізувавши реалізації систем *Smart House* та систем безпеки, можна виділити декілька загальних недоліків більшості реалізацій – проблема масштабування та складний інтерфейс конфігурації:

– проблема масштабування – більшість систем важко масштабувати через відсутність єдиного і універсального інтерфейсу взаємодії, підключення нового

модуля або датчика може бути неможливим по причині монолітної архітектури системи;

– проблема інтерфейса конфігурації – у більшості систем звичайний користувач не має ніякої можливості впливати на склад системи, для цього йому потрібні певні технічні знання або фахівець в даній області для виконання конфігурації системи.

До інших поширених недоліків можна віднести:

– дороге обладнання та обслуговування системи;

– обов'язкове заключення довгострокового контракту з виробником або охоронними фірмами;

– системи різних виробників не синхронизуються між собою;

– застарілий канал сповіщення (*GSM* та *SMS*), відсутність мобільного додатку.

1.5. Постановка задачі

Модуль безпеки в системі *Smart House* повинен мати наступні функціональні можливості:

– запобігання пожежі;

– запобігання затоплення будинку;

– сигналізація та реєстрація високого рівня токсичних газів (чадний газ, пропан, бутан);

– запобігання несанкціонованого доступу.

Система повинна проводити своєчасні сповіщення, які повинні надходити на телефон та електронну пошту користувача, користувач має змогу самостійно підключити модуль, підключивши лише живлення та інтернет з'єднання.

Система повинна бути універсальною та розширюваною і сприймати будь-який тип пристрою і датчика.

Основний список підтримуваних пристроїв:

– датчик широкого спектра газів *MQ-2* – визначає концентрацію вуглеводневих газів (пропан, метан, н-бутан), диму (зважених часток, які є результатом горіння) і водню в навколишньому середовищі;

– датчик горючих газів *MQ-5* – визначає концентрацію скрапленого вуглеводневого газу, метану та коксового газу в навколишньому середовищі;

– інфрачервоний датчик руху *HC-SR501* – складається з чутливого елемента (циліндр з кристалом в центрі), який спрацьовує на інфрачервоне випромінювання;

– глибиномір *T1592* – датчик рівня води для контролю рівня в різних ємностях або сигналізації затоплення;

– датчик температури *DS18B20* є одним з найбільш популярних температурних датчиків, часто він використовується в водонепроникному корпусі для вимірювання температури води або інших рідин.

Система також повинна визначати будь-який інший пристрій, який може реєструвати, вимірювати і видавати аналогове (безперервне) або цифрове (1 або 0) значення.

Відмовостійкість – система повинна в будь-який момент часу відправляти дані і відповідати на запити користувача. Навіть якщо користувач не знаходиться в мережі і не запитував нові дані, система повинна автоматично готувати набір актуальних даних для оперативної відправки при надходженні запиту. Так само незалежно чи знаходиться користувач в мережі чи ні, в критичних ситуаціях система повинна відправляти оповіщення і виконувати заздалегідь записаний алгоритм дій.

Користувач повинен мати доступ до системи за допомогою мобільного додатку. Додаток має автоматично опитувати систему і отримувати актуальні дані і відображати їх. Відображення даних має відбуватися за допомогою формуванні графіків та статусу приладів.

Архітектурно система повинна складатись з таких рівнів:

– набір вимірювальних приладів (датчики);

– прилад контролю и реєстрації даних (контролер *Arduino*);

– блок управління та обробки даних (*Rasbepry PI* або персональний комп’ютер);

– програмний модуль (*Web*–сервер);

– база даних (*SqlLite*);

– мобільний додаток.

Схема архітектури представлена на рис 1.9.

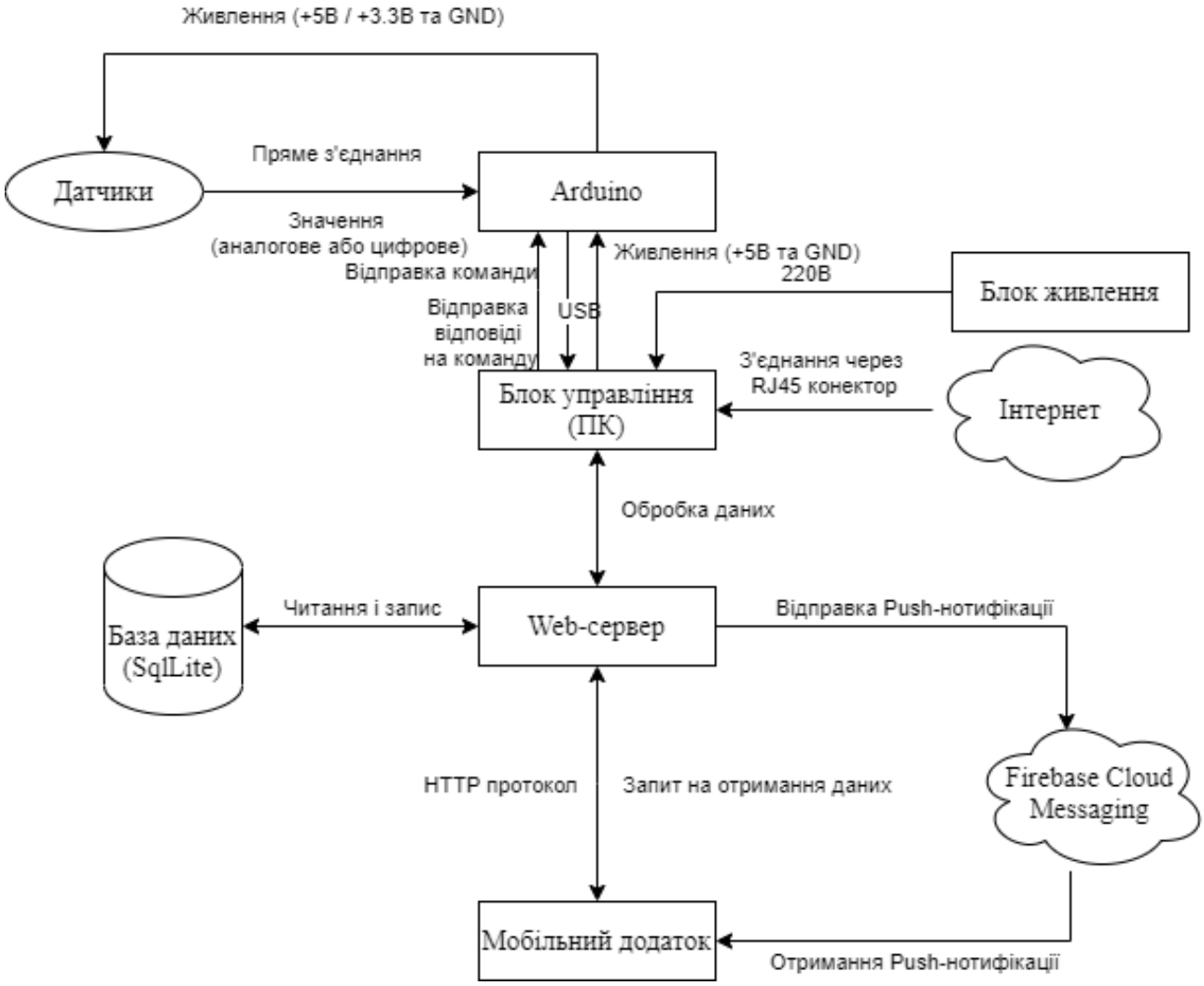

Рис 1.9. Архітектура системи

1.6. Висновки до розділу

В даному розділі була проаналізована предметна область дипломного проєкту, а саме існуючі методи управління безпекою в системах розумних

будинків. Були розглянуті переваги та недоліки найбільш популярних на сьогодні рішень: *Vivint*, *SimpliSafe*, *Frontpoint*, *ADT*, *Blue by ADT*, *Abode*, *Cove* та *Ajax*.

Серед переваг проаналізованих систем слід виділити наступні:

– використання смартфону як пульту управління та контролю за системою;

– об’єднання приладів та датчиків в екосистему *IOT*, що дає змогу виконувати всі обчислення на віддаленому сервері.

Також слід навести недоліки, що були виявлені в цих системах:

– проблема масштабування;

– проблема інтерфейса конфігурації;

– дороге обладнання та обслуговування системи.

На основі наведених вище переваг та недоліків було сформульовано постановку задачі для розроблюваного модулю управління безпекою у системах *Smart House*, функціональні та архітектурні вимоги.

РОЗДІЛ 2
ВИКОРИСТАНІ ТЕХНОЛОГІЇ ТА ПРОЄКТУВАННЯ ПРОГРАМНО-АПАРАТНОГО МОДУЛЯ БЕЗПЕКИ

2.1. Визначення необхідний технологій для розробки

Програмно-апаратний модуль управління безпекою є комплексною структурою, яка складається з трьох основних компонент:

– мікроконтролер та датчики;

– *web*-сервер обробки даних;

– мобільний додаток.

Для цих компонентів було визначено необхідний набір технологій.

2.1.1. Мікроконтролер *Arduino UNO*

Arduino UNO - це мікроконтролер, який заснована на 8-бітному процесорі *ATmega328P*. Також він складається з інших компонентів такі як кварцовий генератор, послідовний канал зв'язку та регулятор напруги. *Arduino UNO* має 14 цифрових *input*/*output* пінів, 6 аналогових входів (які також можна використовувати як цифрові) та *USB*-з'єднання.

Arduino можна використовувати для спілкування з комп’ютером або іншими мікроконтролерами. На рис. 2.1 представлено мікроконтролер *Arduino UNO R3*

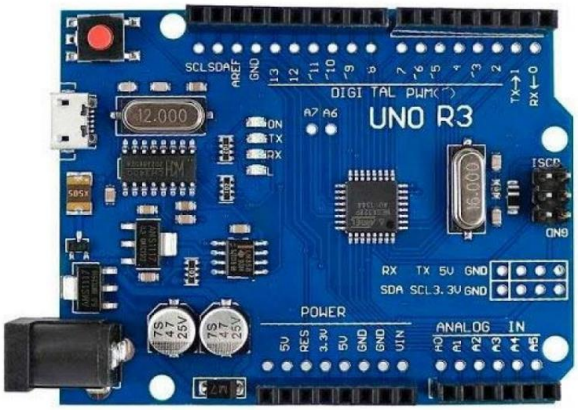

Рис. 2.1. Мікроконтролер *Arduino UNO R3*

2.1.2. Мова програмування *C++*

Код для *Arduino* створюється за допомогою мови програмування *C++* та додаванням спеціальних методів та функцій (бібліотек).

C++ – це об’єктно-орієнтована комп’ютерна мова програмування, створена як нащадок сімейства мов *C*. Вона була розроблена як крос-платформне вдосконалення *C*, щоб забезпечити розробникам вищий ступінь контролю над пам'яттю та системними ресурсами.

C++ – високорівнева мова програмування, так як дозволяє використовувати об'єтно-орієнтований підхід розробки та створення абстракцій в виді базових класів для подальшого наслідування та реалізації і утилізації коду.

В об'єктно-орієнтованому програмуванні об'єкт - це тип даних, який має як дані, так і функції. Ідентифікація об’єктів із вбудованими даними та функціями призвела до нового способу упаковки та автоматизації роботи коду.

Інтегроване середовище розробки *Arduino* (*IDE*) – основна програма написання коду, що використовується для програмування *Arduino*.

2.1.3. Протокол *HTTP*

Програмний модуль *Web*-сервера обробки даних повивен працювати по протоколу *HTTP* для спілкування з мобільним додатком.

Hyper Text Transfer Protocol – протокол для гіпермедійних систем, який знаходиться на рівні програмного забезпечення.

Версія *HTTP*-0.9 була протоколом передачі простих даних через канал зв'язку. Версія *HTTP*-1.0 вже покращила протокол, створивши формат повідомлень типу *Multipurpose Internet Mail Extension*, що містить інформацію про відправленні дані.

Протокол *HTTP* здійснюється обмін ресурсами сервера та клієнта через інтернет. Клієнтські пристрої надсилають запити на сервери щодо ресурсів, необхідних для завантаження веб-сторінки після чого сервери надсилають відповіді клієнту. Запити та відповіді обмінюються піддокументами - такими як дані про зображення, текст, текстові макети тощо – які з'єднуються клієнтським веб-браузером для відображення певної веб-сторінки.

Кожен *HTTP*-запит містить закодовані дані з такою інформацією, як:

- конкретна версія *HTTP*;
- *URL*-адреса, що вказує на веб-ресурс;
- метод *HTTP* – вказує на тип дії (створення, видалення, відправка, тощо), яку сервер повивен виконати;
- заголовки *HTTP*-запитів – включають в себе тип браузера, який використовується та які дані запит очікує від сервера;
- тіло *HTTP* – необов'язкова інформація, яка потрібна серверу. Тіло може включати в себе будь-яку інформацію, наприклад логіни для входу, файли, тощо.

Відповіді *HTTP* зазвичай містять такі дані:

– код стану *HTTP* відповіді – вказує на стан запиту до клієнтського пристрою. Можуть свідчити про успіх виконання запиту або про помилки на стороні сервера чи клієнта;

– заголовки відповіді *HTTP* – включають в себе інформацію про сервер та запитувані ресурси.

2.1.4. Структура *Web*-серверу

Мова розмітки гіпертексту (*HTML*) - це основний мовний стандарт, що використовується для впорядкування та форматування веб-сторінок та інших документів у Всесвітній павутині. Вона часто використовується в поєднанні з каскадними таблицями стилів (*CSS*) та *JavaScript* для створення повністю адаптивних веб-сторінок, які відображаються правильно на різних розмірах екранів.

HTML визначає, які частини тексту є абзацами, заголовками, гіперпосиланнями, а *CSS* визначає, як ці частини візуально виглядають. *JavaScript*, навпаки, додає на сторінку динамічні елементи, такі як спливаючі вікна, анімована графіка, прокрутка банерів, тощо.

Каскадні таблиці стилів (*CSS*) забезпечують центральне розташування інформація про те, як різні шрифти, кольори переднього плану, кольори фону та ін. слід застосовувати до різних елементів *HTML* на веб-сторінці.

Javascript (*JS*) - це мова сценаріїв, яка в основному використовується в для розробки веб-додатків. Вона використовується для вдосконалення *HTML*-сторінок і зазвичай вже вбудована у *HTML*-код.

JavaScript може відображати веб-сторінки інтерактивно та динамічно незалежно від веб-сервера. Це дозволяє сторінкам реагувати на події, демонструвати спеціальні ефекти, перевіряти дані на стороні клієнта, створювати файли *cookie*, визначати браузер користувача тощо.

2.1.5. Мова програмування *C#*

C# є похідною мовою від мови програмування *C* і подібна до мови *C++*. *C#* використовує ті ж самі оператори, що і *C++*, є об’єктно-орієнтованим, чутливим до регістру та має майже однаковий синтаксис як у *C++*.

Але найважливішою відмінністю *C#* від *C++* є те, що програма написана на мові *C#* працює в керованому середовищі виконання - це означає, що програма не має прямого доступу до пам'яті і їй не потрібно стежити за нею, створювати або видаляти об'єкти. Це дозволяє уникнути помилок звернень до пам'яті та своєчасної очистки від сміття.

У керованому середовищі компіляція проводиться в 2 етапи:

– компілятор переводить *C#* код в *IL* код;

– для виконання програми потрібно перевести *IL* код в машинний код процесора, що вимагає додаткової динамічної пам'яті і часу.

2.1.6. *ASP.NET Core* та *ASP.NET Core MVC*

ASP.NET Core - платформа з відкритим кодом, яка призначена для створення хмарних додатків, таких як веб-додатки, *IoT* додатки та мобільні серверні системи. *ASP.NET Core* призначений для роботи як у хмарних сервісах, так і в локальній мережі.

ASP.NET Core працює поверх протоколу *HTTP* і використовує команди та політики *HTTP* для встановлення двостороннього зв'язку між браузером і сервером. *ASP.NET Core* дозволяє виконувати і обробляти будь-які запити, завантажувати і зберігати файли, спілкуватися з базою даних, спілкуватися з іншими веб-додатками. *ASP.NET Core* є основою, яку можна розширювати різним функціоналом, в тому числі і власноручно створеним.

На рис. 2.2 детально зображено життєвий цикл запиту в *ASP.NET Core.*

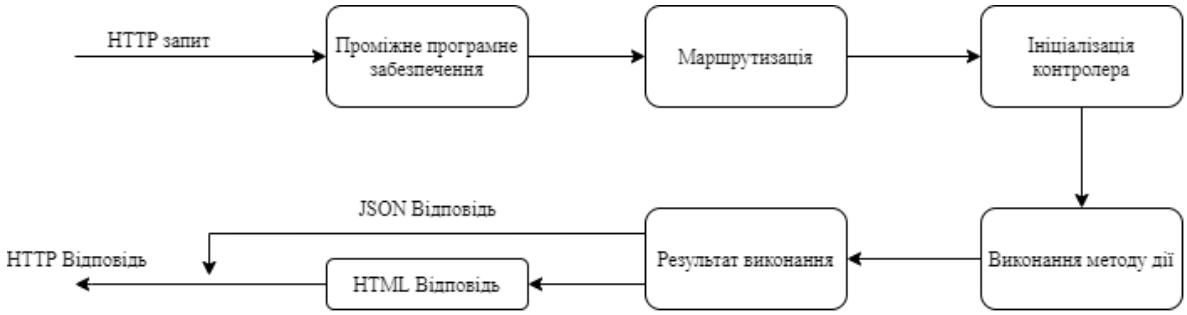

Рис. 2.2. Життєвий цикл *HTTP* запиту в *ASP.NET Core*

ASP.NET Core MVC - це фреймворк для створення веб-додатків та *API* за допомогою шаблону проєктування *Model-View-Controller*.

Архітектурний шаблон *Model-View-Controller* (*MVC*) розділяє додаток на три основні групи компонентів: моделі, представлення та контролери.

За допомогою цього шаблону запити користувачів направляються до контролера, який відповідає за роботу з моделлю данних для виконання дій користувача. Контролер обирає необхідне представлення (*HTML* файл), яке потрібно відобразити користувачеві і надає йому потрібну модель даних.

Модель – представляє стан програми та будь-яку бізнес-логіку або операції, які вона повинна виконувати.

Представлення – відповідають за представлення даних через інтерфейс користувача.

Контролери – компоненти, які обробляють запити користувача, працюють із моделлю та, зрештою, вибирають подання для візуалізації.

2.1.7. СУБД та *Entity framework*

В даному проєктуванні була обрана СУБД *SQLite* так як вона є невеликою за розміром і вбудовується в проєкт. Так само в БД будуть відсутні зв'язку між сутностями, так як в БД планується зберігати тільки значення датчиків.

На відміну від систем управління базами даних клієнт-сервер, движок *SQLite* не має автономних процесів, з якими прикладна програма взаємодіє.

Натомість бібліотека *SQLite* вбудовується і, таким чином, стає невід'ємною частиною програмного забезпечення.

Програмне забезпечення використовує функціональні можливості *SQLite* за допомогою простих викликів функцій, які зменшують затримку доступу до бази даних: виклики функцій у межах одного процесу ефективніші, ніж міжпроцесовий зв'язок.

SQLite зберігає всю базу даних (таблиці, індекси та самі дані) як єдиний файл на хост-машині. Він реалізує цю просту конструкцію, блокуючи весь файл бази даних під час запису. Операції читання *SQLite* можуть бути багатопотоковими, на відміну від операцій запису, вони можуть виконуватися лише послідовно.

Завдяки безсерверному дизайну, *SQLite* вимагає меншої конфігурації, ніж бази даних клієнт-серверної архітектури. Контроль доступу здійснюється за допомогою дозволів файлової системи, що надаються самому файлу бази даних.

У реалізації веб-сервера є вимога уникнути використання і написання прямих запитів мови *SQL*, такі як оператори *SELECT, CREATE, DELETE*. Для цього потрібно використовувати технологію *Entity framework.*

Entity Framework (*EF*) - це об'єктно-реляційне відображення (*ORM*) з відкритим вихідним кодом (рис. 2.3).

Entity Framework дозволяє розробникам працювати з даними у вигляді об'єктів та властивостей, що стосуються домену, таких як клієнти та адреси клієнтів, тощо, без необхідності працювати з реальними таблицями та стовпцями в базі даних. За допомогою *Entity Framework* розробники можуть працювати на вищому рівні абстракції, коли мають справу лише з даними.

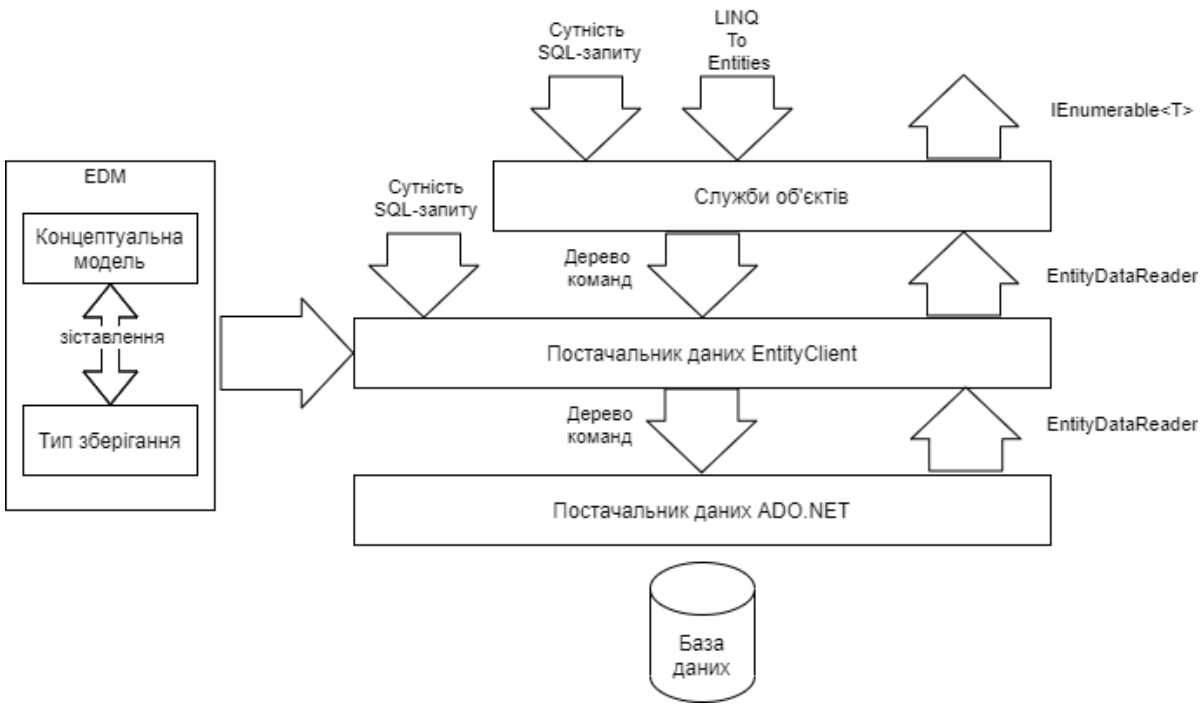

Рис. 2.3. Архітетура *Entity framework*

2.1.8. Мобільна *OS Android* та мова програмування *Java*

Операційна система *Andriod* була обрана після проведення аналізу та оцінки витрат часу на реалізацію, так як для написання та створення мобільного додатку для операційної системи *Android* не потрібно дороге обладнання.

Операційна система *Android* - це мобільна операційна система, розроблена *Google*, яка в основному використовується для сенсорних пристроїв, мобільних телефонів та планшетів. Його конструкція дозволяє користувачам інтуїтивно маніпулювати мобільними пристроями за допомогою рухів пальців, що відображають загальні рухи, такі як пощипування, проведення та натискання. *Google* також використовує програмне забезпечення *Android* для телевізорів, автомобілів та наручних годинників – кожен з яких має унікальний користувальницький інтерфейс.

Розробка програмного забезпечення *Android* - це процес, за допомогою якого створюються додатки для пристроїв під управлінням операційної системи Android. *Google* заявляє, що додатки для *Android* можна писати за допомогою мов

Kotlin, Java та *C++*, використовуючи набір для розробки програмного забезпечення *Android* (*SDK*).

Набір для розробки програмного забезпечення для *Android* (*SDK*) включає повний набір засобів розробки. Сюди входять налагоджувач, бібліотеки, емулятор слухавки на основі *QEMU* та документація.

Java - це об'єктно-орієнтована мова програмування загального призначення, заснована на класах, призначена для зменшення залежностей реалізації. Це обчислювальна платформа для розробки додатків. *Java* швидка, безпечна та надійна. Вона широко використовується для розробки програм для ноутбуків, персональних комп'ютерів та для мобільних пристроїв.

Існує три основні компоненти мови програмування *Java*:

– віртуальна машина *Java* (*JVM*) – це механізм, який забезпечує середовище виконання для керування кодом або програмами, вона є центром мови програмування і виконує операцію перетворення байт-коду *Java* у машинну мову;

– середовище виконання *Java* (*JRE*) – це середовище виконання, яке потрібно для запуску програм *Java*, якщо користувач хоче запустити програму *Java* на своїй машині, у нього повинна бути встановлена *JRE*, що залежить від платформи, тобто *JRE* повинно бути сумісним з операційною системою та архітектурою локальної машини користувача;

– набір для розробки *Java* (*JDK*) – є основним компонентом середовища *Java* та містить *JRE* разом із компілятором *Java*, налагоджувачем *Java* та іншими класами, він використовується для розробки додатків, компіляції та налагодження.

2.1.9. *Firebase Cloud Messaging*

Firebase Cloud Messaging (*FCM*) - це рішення для обміну повідомленнями між платформами, яке дозволяє надійно надсилати повідомлення безкоштовно.

FCM надає можливість повідомити клієнтський додаток про те, що нові дані доступні для синхронізації. *FCM* може надсилати сповіщення, щоб стимулювати повторне залучення та утримання користувачів або для екстреного сповіщення. Для таких випадків використовується миттєвий обмін повідомленнями.

Реалізація *FCM* включає два основні компоненти для відправки та отримання повідомлень:

– довірене середовище, таке як сервер додатків, на якому можна створювати та надсилати повідомлення;

– клієнтська програма для *iOS*, *Android* або веб-браузер (*JavaScript*), яка отримує повідомлення через відповідний канал зв'язку.

FCM спирається на наступний набір компонентів, які створюють, передають та отримують повідомлення (рис. 2.4).

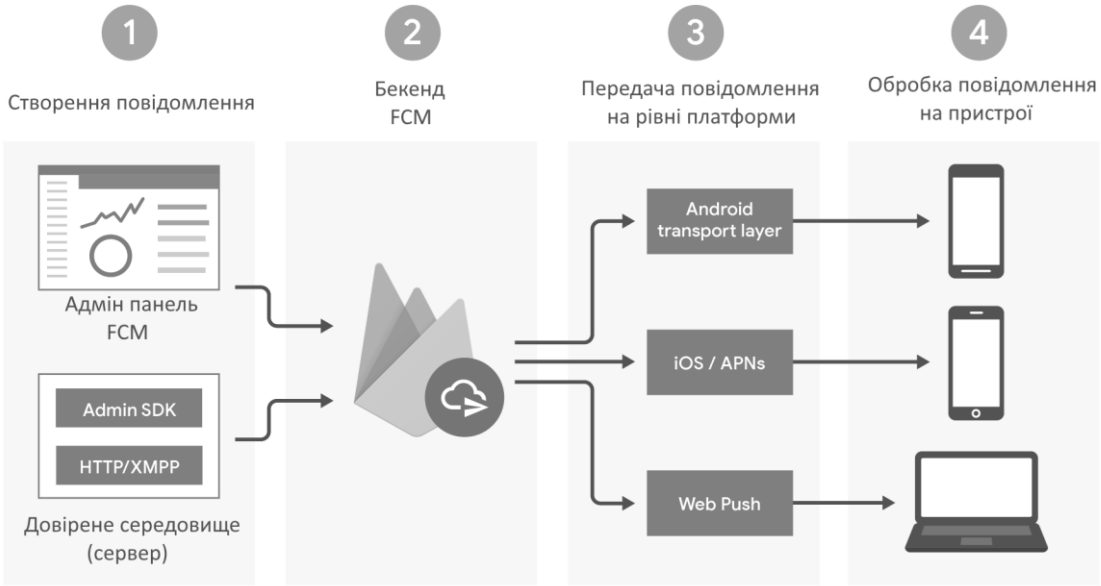

Рис. 2.4. Архітектура *FCM*

2.2. Проєктування функціональної частини системи

Система управління безпекою складається з трьох основних компонентів:

– апаратне забезпечення для реєстрації вхідних сигналів у вигляді мікроконтролера *Arduino UNO* та датчиків;

– блок управління та обробки даних (*web*-сервер);
– користувацький додаток у вигляді *android* застосунку.

2.2.1. Діаграма станів апаратного забезпечення

Апаратне забезпечення складається з мікроконтролера реєстрації сигналів та датчиків. На рис. 2.5 приведена діаграма станів апаратного збезпечення.

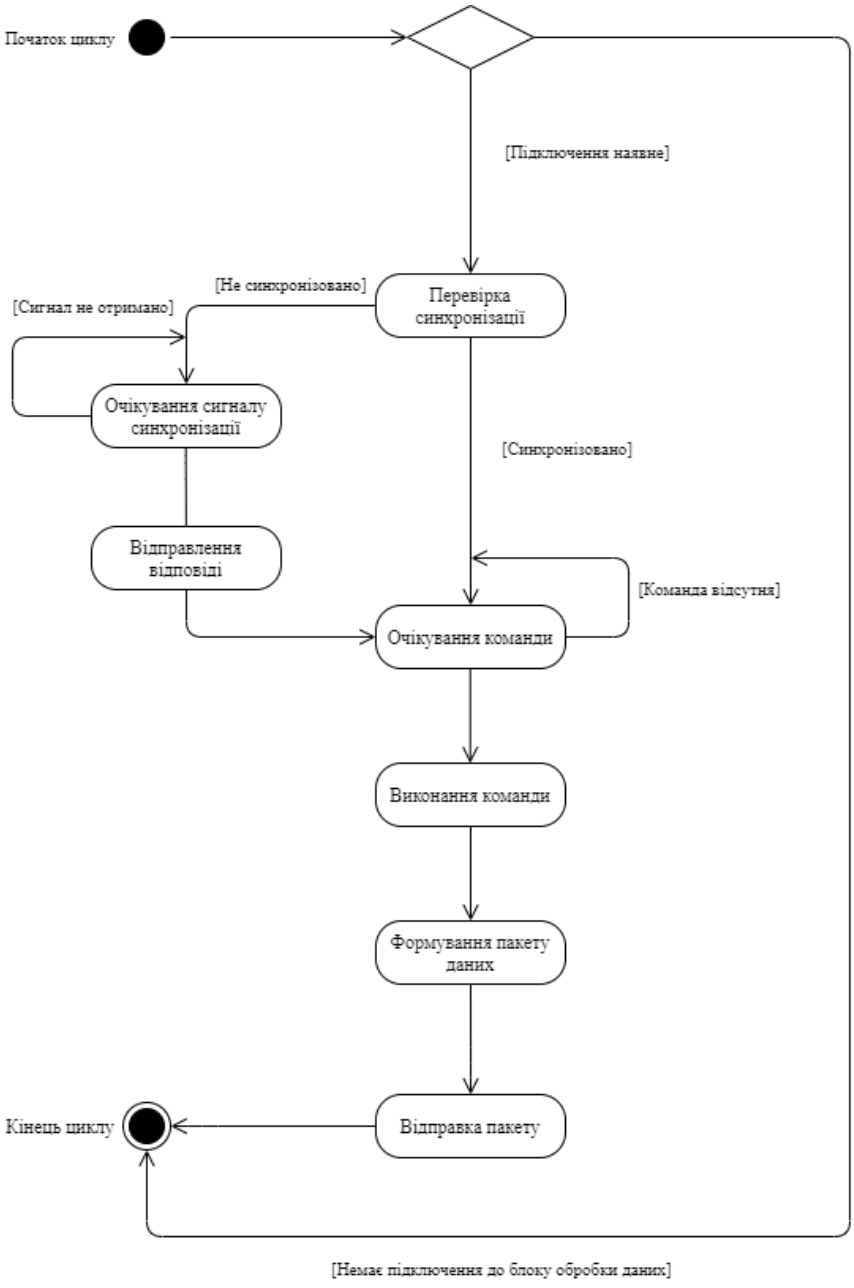

Рис. 2.5. Діаграма станів апаратного забезпечення

Мікроконтролер чекає підключення до блоку управління і синхронізуючого сигналу. Після синхронізації контролер починає слухати порт і очікувати команди від блоку управління, після отримання команди, контролер починає вичитувати сигналі датчиків і відправляти на блок управління в вигляді пакета даних. Далі цикл повторюється.

2.2.2. Діаграма станів блоку управління (*web*-сервера)

Блок управління складається з програмного забезпечення (*web*-сервер), яке після розгортання і запуску автоматично починає шукати всі доступні послідовні порти (*COM*). Після знаходження всіх активних портів, веб-сервер починає відправляти кожному з них запит на синхронізацію, за допомогою ключа, якщо в тривалий проміжок часу немає відповіді, запит на синхронізацію відправляється наступному порту.

Після синхронізації з апаратним забезпеченням, *web*-сервер починає посилати команди на отримання даних з датчиків, згідно конфігурації. Команда складається з номера порту підключеного датчика і типу значення (аналогове або цифрове). Після отримання пакету даних, сервер насамперед записує значення в базу даних і після цього виконує обробку. Під обробкою мається на увазі заздалегідь заданий алгоритм перевірки критичного значення, якщо значення перевищує критичне, то користувачам негайно відправляється сповіщення з назвою датчика і поточним значенням. Новий цикл починається після обробки даних, слід зауважити, що цикл працює в окремому потоці виконання.

У будь-який момент часу користувач має можливість запросити актуальні дані, після отримання такого запиту, *web*-сервер готує пакет даних для відправки, записуючи значення датчиків з бази даних в пакет.

На рис. 2.6 приведена діаграма станів блоку управління.

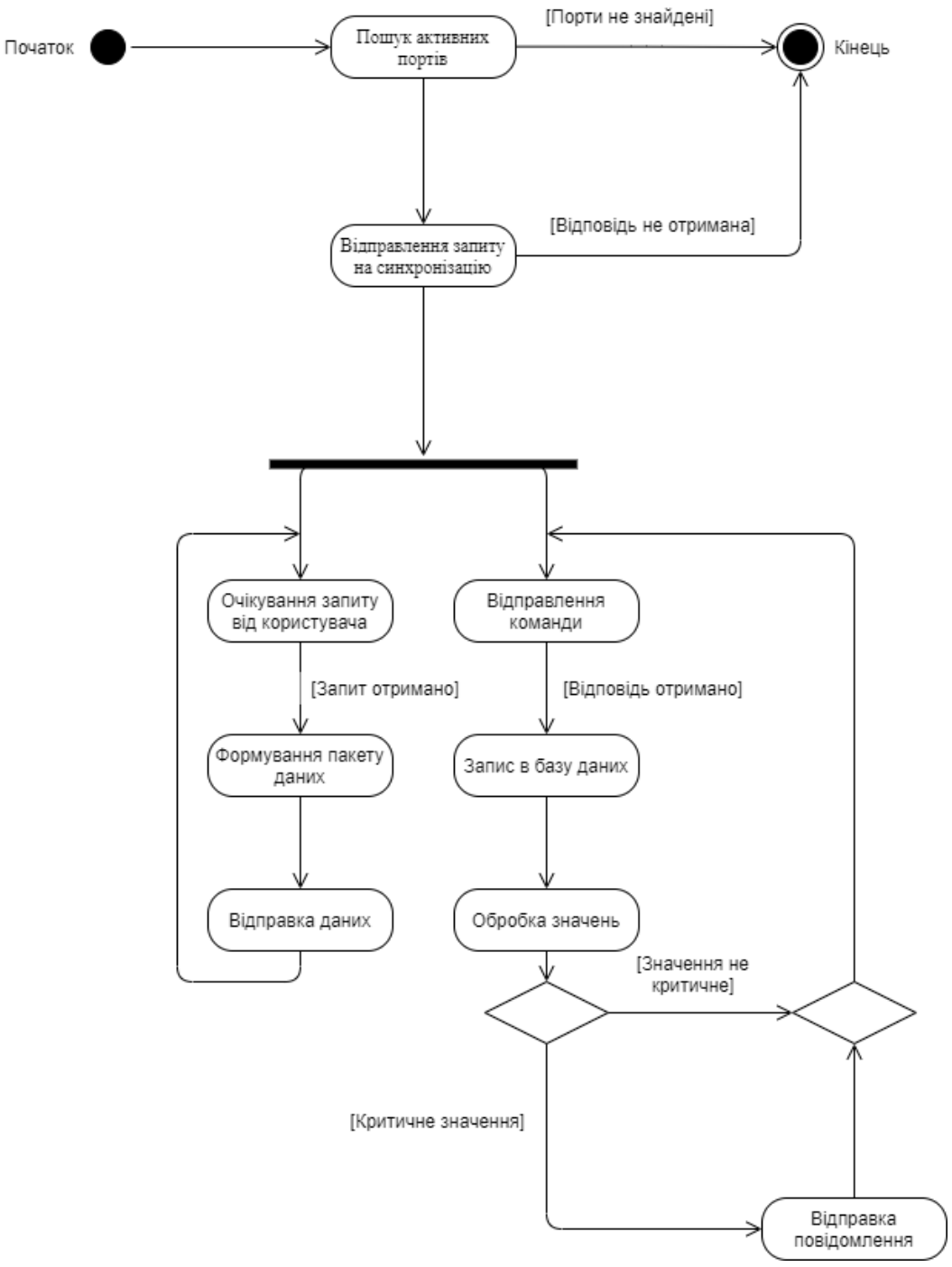

Рис. 2.6. Діаграма станів блоку управління

2.2.3. Клієнтський додаток

Функціонально клієнтську програму має тільки можливість на отримання актуальних даних. Додаток відображає поточний стан підключення контролера і список підключених датчиків. По кожному датчику можна отримати Побачити графік значень.

Додаток автоматично синхронізується з веб-сервером обробки даних, для цього користувач повинен знаходиться разом з веб-сервером в одній локальній мережі.

На рис. 2.7 представлена *Use-Case* діаграма клієнтського додатку.

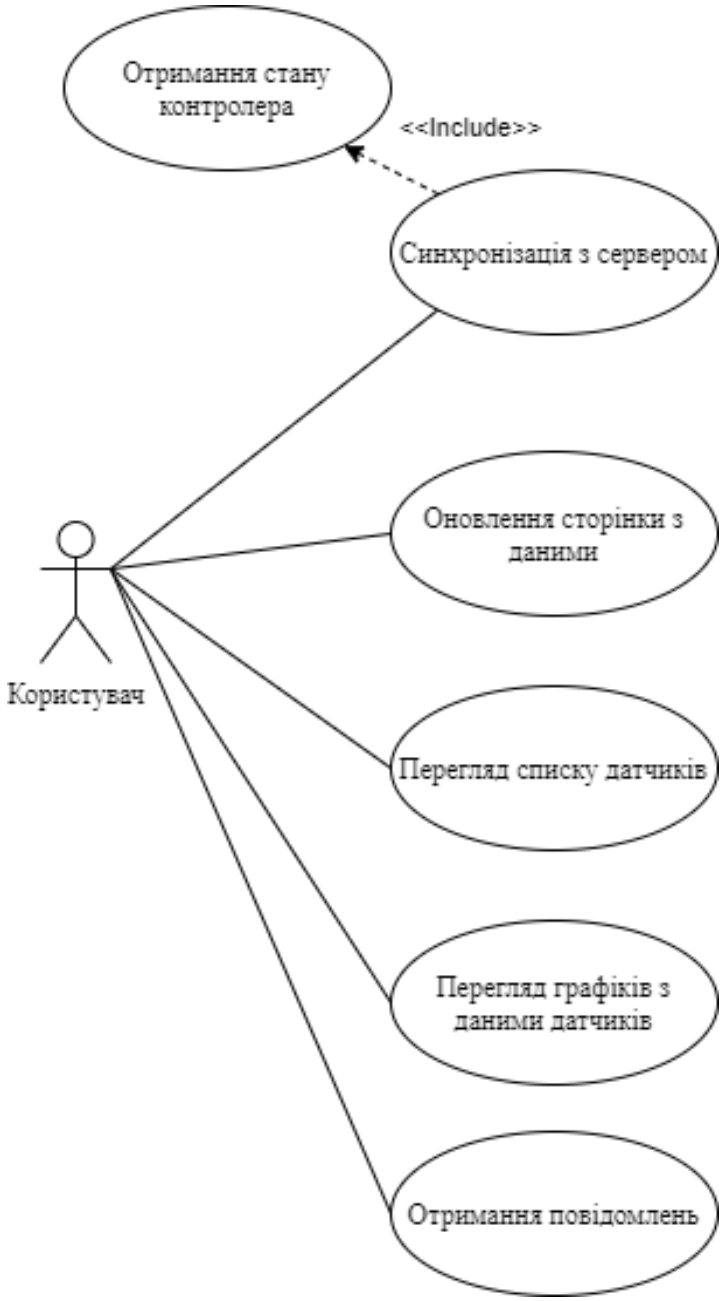

Рис. 2.7. *Use-Case* діаграма клієнтського додатку

2.3. Висновки до розділу

В даному розділі були проаналізовані та обрані всі необхідні технології для проєктування системи управління безпекою.

Розроблюваний модуль складається з трьох основних елементів, для кожного з яких були визначені необхідні технології:

– апаратне забезпечення – *Arduino UNO, Arduino IDE, C++*;

– блок управління – *C#, ASP.NET Core, ASP.NET Core MVC, Entity Framework Core, SQLite, HTML* 5, *CSS* 3, *JavaSciprt*;

– мобільний додаток – *Android Studio, Java, FCM*.

Для кожного основного елемента були описані і створені діаграма станів та *use-case* діаграма.

РОЗДІЛ 3
РОЗРОБКА МОДУЛЮ УПРАВЛІННЯ БЕЗПЕКОЮ

3.1. Розробка прошивки апаратного забезпечення

Для того щоб контролер мав можливість зчитувати дані з датчиків, синхронізуватися і спілкуватися з сервером йому потрібен набір команд (програма), яка буде виконувати записаний алгоритм дій. Для *Arduino UNO* цією програмою є прошивка, написана на мові *C++*.

Основною складовою прошивки *Arduino* є два заздалегідь вбудовані методи *setup ()* і *loop ()*.

У методі *setup* відбувається первинна конфігурація контролера, конфігурація портів, тощо. Цей метод викликається тільки раз, при запуску контролера.

Метод *loop* є циклом програми, посуті цей метод викликається постійно в безперервному циклі, поки контролер не буде вимкнено.

У прошивці також дозволено оголошувати власні методи, константи, перерахування.

Для того щоб сервер міг спілкуватися з контролером по *USB* з'єднанню, контролеру потрібно відкрити послідовний канал зв'язку:

```
void setup()
{
        Serial.begin(9600);
}
```

Метод *begin()* класу *Serial* відкриває послідовний канал зв'язку зі швидкістю передачи даних 9600 *біт/с* (*бод*).

У методі *loop()* знаходиться основний цикл програми, на самому початку якого викликається метод очікування команд від сервера, в тому числі і команда на синхронізацію:

```
void loop()
{
  JSONVar request = awaitCommand();

  Serial.println(AwaitCommandMarker);

  int cmd = request["command"];

  Command command = cmd;

  JSONVar response;

  switch(command)
  {
   case Ping:
    if(request["data"] == PingRequest)
    {
     response["success"] = true;
     response["data"] = PingResponse;
    }
    break;
   case ReadPin:
     response["success"] = true;
     response["data"] = PinRead(request["data"]);
    break;
   default:
    response["success"] = true;
```

```
    response["data"] = "Unknown command";
  }

  Serial.flush();

  Serial.println(JSON.stringify(response));
}
```

У методі *awaitCommand()* відбувається очікування отримання будь-яких байтів з послідовного порту, якщо щось було передано в порт, то цей метод виходить з циклу і повертає отримані байти:

```
JSONVar awaitCommand()
{
  do{
    delay(100);
  }while(Serial.available() == 0);

  String request = Serial.readString();

  JSONVar requestObj =  JSON.parse(request);

  return requestObj;
}
```

Структура команди складається з двох полів *"command"* та *"data"*. В поле *"command"* записується тип команди. Для визначення типів команд було створено перерахування *enum Command*, яке містить константи *Ping* та *ReadPin*:

```
enum Command
{
  Ping,
  ReadPin
};
```

Тип *Ping* визначає команду на синхронізацію з сервером. Для синхронізації присутні дві константи ключів, в контексті контролера, перша константа *const String PingRequest = "WEBSERVERPING"* позначає запит на синхронізації від сервера. Друга константа *const String PingResponse = "CONTROLLERRESPONCE"* позначає відповідь до сервера, що контролер синхронізований.

Якщо контролер отримав таку команду, то першим ділом перевіряється, що це дійсно запит на синхронізацію за допомогою порівняння вхідного значення в полі *"data"* і константи ключа *PingRequest = "WEBSERVERPING":*

```
case Ping:
 if(request["data"] == PingRequest)
 {
  response["success"] = true;
  response["data"] = PingResponse;
 }
```

Якщо ключ запиту збігається з константою, то контролер відправляє у відповіді серверу константу *PingResponse*.

Другим типом команди є команда на вичитку даних з датчиків. У запиті команди знаходяться дані про те якого типу датчик (аналоговий або цифровий) та номер порту, до якого підключений датчик:

```
case ReadPin:
  response["success"] = true;
  response["data"] = PinRead(request["data"]);
 break;
```

В методі *PinRead()* витягуються дані з датчиків:

```
int PinRead(JSONVar data)

{

 int pin = data["pin"];
```

```
if(data["isAnalog"])
{
  return analogRead(pin);
}
else
{
  pinMode(pin, INPUT);
  return digitalRead(pin);
}
}
```

Через те, що спілкування сервера і контролера відбувається по послідовному порту, то процеси читання і запису в ньому ніяк не синхронізовані. Для синхронізації цих двох дій, щоб сервер чекав відповіді від контролера, в послідовний порт контролер записує маркер *const String AwaitCommandMarker = "AWAITCOMMAND"* про те що він прийняв команду в обробку:

```
Serial.println(AwaitCommandMarker);
```

В кінці циклу контролер очищає послідовний порт і записує в нього відповідь:

```
Serial.flush();
Serial.println(JSON.stringify(response));
```

3.2. Розробка програмного забезпечення для блоку обробки даних

Так як сервер повинен мати можливість не тільки спілкуватися з контролером локально, але також передавати дані користувачам по мережі і мати зручний інтерфейс конфігурації, для цієї мети підійшла технологія *ASP.NET Core*.

ASP.NET Core дозволяє взаємодіяти з операційною системою, в тому числі з послідовним портом, взаємодіяти з мережею і відправляти запит по протоколу *HTTP*. Для відображення інтерфейсу конфігурації є можливість використовувати технологію *ASP.NET Core MVC* для формування, відображення сторінок і створення бізнес-логік моделей.

3.2.1. Конфігурація *web*-сервера

ASP.NET Core за своєю суттю є звичайним консольним додатком, точка входу якого знаходиться в класі *Program* в методі *Main()*:

```
public class Program
  {
    public static void Main(string[] args)
    {
      CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
          webBuilder.UseStartup<Startup>();
          webBuilder.UseKestrel(serverOptions =>
          {
       serverOptions.Listen(IPAddress.Parse(GetLocalIPAddress()), 5001);
          });
```

```
        });
```

Щоб додаток мав можливість спілкуватися по мережі, йому треба надати конфігурацію. Метод *CreateHostBuilder* починає конфігураційний процес і запускає екземпляр веб-сервера:

```
Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
        webBuilder.UseKestrel(serverOptions =>
        {
        serverOptions.Listen(IPAddress.Parse(GetLocalIPAddress()), 5001);
        });
    }
```

Вся конфігурація веб-сервера знаходиться в класі *Startup*, в якому описуються які модуля потрібно підключитися до веб-сервера, правила обробки запитів, проміжні програмні забезпечення.

У методі *public void ConfigureServices(IServiceCollection services)* оголошуються і додаються всі доступні для виклику сервіси. Наприклад додається сервіс для роботи модуля *ASP.NET Core MVC*:

```
services.AddControllersWithViews().AddJsonOptions(opts =>
        {
    opts.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());
        });
```

Або ж додається модуль *EntityFramework*, із зазначенням того, що використовується СУБД *SQLite*:

services.AddEntityFrameworkSqlite().AddDbContext<DatabaseContext>();

Конфігурація відбувається в методі *public void Configure()*, який виконується після оголошення сервісів.

public void Configure(IApplicationBuilder app, IWebHostEnvironment env, IServiceProvider serviceProvider)

```
{
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
    }
}
```

В ньому вказується, що додатку потрібно використовувати каталог для зберігання статичного контенту (зображення, тощо), авторизацію і маршрутизацію, для того щоб можна було виконувати отримання і переходи на сторінки по *URL*-адресі.

3.2.2. Структура *web*-сервера

Сервер складається з двох основних сторінок, головна сторінка та сторінка конфігурації. Нижче преведен уток коду контролера, який відповідає за головну сторінку:

```
public class HomeController : Controller{
    public IActionResult Index()
    {
        return View();
    }
    public IActionResult GetControllerState()
    {
        StateModel state = this._controllerProvider.CheckStatus();
        return PartialView("ControllerState", state);
    }
    public async Task<IActionResult> GetSensorData(string key, int take)
    {
     IEnumerable<SensorData> data = await this._sensorProvider.GetSensorData(key, take);
    return Json(data
            .OrderBy(x => x.Date).Select(x => new { Data = x.Data, Date = x.Date.ToString("hh:mm:ss"), IsAnalog = x.IsAnalog }));
    }
}
```

Метод *GetControllerState()* повертає поточний стан контролера, чи підключений він до послідовного порту у вигляді часткової розмітці *HTML*.

Метод *GetSensorData()* отримує по ключу з бази даних всі значення датчика та повертає їх у вигляді списку об'єктів.

Головна сторінка відображається за допомогою методу *Index()*, як тільки ми потрапляємо в інтерфейс *web*-сервера.

```
public IActionResult Index()
{
    return View();
}
```

Рядок *return View()* повертає подання, яке було задано для цього контролера у вигляді розмітці *HTML*:

```
@inject DatabaseContext Context
@{
   ViewData["Title"] = "Home Page";
}
<div class="text-center controller">
   <div class="controller-state">
      <h1>Controller state:</h1>
      <div class="lds
ring"><div></div><div></div><div></div><div></div></div>
   </div>
   @await Html.PartialAsync("SensorList", Context.Sensors)
</div>
```

```
@section Scripts
{
    <script>
        $.ajax({
            url: '@Url.Action("GetControllerState", "Home")',
            type: 'GET',
            success: function (data) {
                $('.controller-state').children().remove();
                $('.controller-state').append(data);
            }
        });
    </script>
}
```

За допомогою методу *@await Html.PartialAsync("SensorList", Context.Sensors)* ми можемо зробити запит на отримання часткової розмітки, яка містить таблицю з списком активних датчиків:

```
@model IEnumerable<SensorModel>
<div class="container">
    <table class="table table-hover">
        <thead>
            <tr>
                <th scope="col">Sensor</th>
                <th scope="col">Pin</th>
                <th scope="col">Type</th>
```

```
                <th scope="col"></th>
            </tr>
        </thead>
        <tbody>
            @foreach (SensorModel sensor in Model)
            {
                string pinType = sensor.IsAnalog ? "A" : "D";
                string pin = $"{pinType}{sensor.Pin}";
                <tr class="sensor" data-url="@Url.Action("GetSensorData", "Home")" id="@(sensor.Pin.ToString() + sensor.IsAnalog.ToString())">
                    <td>@sensor.DisplayName</td>
                    <td>@pin</td>
                    <td>@sensor.Type.ToString()</td>
                    <td class="remove-sensor" data-url="@Url.Action("RemoveSensor", "Setting")" data-pin="@sensor.Pin" data-is-analog="@sensor.IsAnalog">X</td>
                </tr>
            }
        </tbody>
    </table>
    <div style="height: 300px">
        <canvas id="sensorChart"></canvas>
    </div>
</div>
```

Дана розмітка після отримання буде автоматично вбудована в місце виклику методу *Html.PartialAsync()*.

Другою основною сторінкою є сторінка конфігурації, на якій є можливість додавання нових датчиків із зазначенням їх назви, тип значення (аналогове або цифрове), критичне значення, тип датчика (газовий, руху, води, тощо):

```
@model SensorModel

<div class="add-sensor-form">

    @using (Html.BeginForm("AddSensor", "Setting", FormMethod.Post, new { id = "add-sensor" }))

    {

        <div>

            <div class="validation" asp-validation-summary="ModelOnly"></div>

            <div class="form-group">

                <label asp-for="DisplayName">Sensor name</label>

                <input class="form-control" type="text" asp-for="DisplayName" />

                <span asp-validation-for="DisplayName"></span>

            </div>

            <div class="form-group">

                <label asp-for="Pin">Pin number</label>

                <input class="form-control" type="number" asp-for="Pin" />

                <span asp-validation-for="Pin"></span>

            </div>

            <div class="form-group">

                <label asp-for="Type">Sensor type</label>
```

```
                <select class="form-control" asp-for="Type" asp-
items="Html.GetEnumSelectList<SensorType>()"></select>
            </div>
            <div class="form-check">
            <input class="form-check-input" type="checkbox" asp-for="IsAnalog" />
             <label asp-for="IsAnalog">Is analog</label>
            </div>
            <div>
              <input type="submit" class="btn btn-primary" value="Add sensore" />
            </div>
        </div>
    }
</div>
```

3.2.4. Спілкування сервера з контролером

Перед відправкою команд до контролера на вичитку даних, сервер першим ділом намагається знайти і синхронізуватися з контролером:

```
private SerialPort PingAndConnectPort(string port)
    {
        SerialPort serialPort = new SerialPort(port, BaudRate);
        serialPort.ReadTimeout = ReadTimeOut;
        try
        {
```

```
serialPort.Open();
var message = JsonConvert.SerializeObject(new { Command = ControllerCommand.Ping, Data = PingRequest });

serialPort.WriteLine(message);
string responseJson = this.ReadCommand(serialPort);
if (string.IsNullOrEmpty(responseJson))
{
   serialPort.Close();
   return null;
}
ControllerResponse response = JsonConvert.DeserializeObject<ControllerResponse>(responseJson);
if (response.Success && string.Equals(response.Data, PingResponse, StringComparison.InvariantCultureIgnoreCase))
{
   return serialPort;
}
```

Після знаходження всіх активних портів, сервер відправляє в послідовний порт ключ константу, яку повинен перевірити контролер і відправити з свого боку свій ключ, який і повідомить серверу, що за поточним портом знаходиться потрібний контролер. Як тільки пристрої синхронізувалися і сервер отримав дані датчиків з контролера, відбувається їх обробка, в першу чергу значення записується в базу даних з прив'язкою до датчика:

```
SensorData sensorData = new SensorData
```

```
            {
                Data = data.Data,
                Pin = sensor.Pin,
                IsAnalog = sensor.IsAnalog,
                Date = DateTime.Now,
                SensorId = sensor.Id.ToString()
            };
        await this._context.SensorsData.AddAsync(sensorData);
        sensor.SensorState = this.GetSensorState(sensor);
        this._context.Sensors.Update(sensor);
        await this._context.SaveChangesAsync();
```

Після запису у базу, перевіряється чи є значення критичним, якщо так то користувачам відправляється повідомлення:

```
        bool isCritical = (sensor.IsAnalog
            ? Convert.ToDouble(data.Data) >= sensor.CriticalValue
            : Convert.ToInt32(data.Data) == sensor.CriticalValue) &&
!HasCache(sensor.Id.ToString());
        if (isCritical)
        {
            string message = $@"Значення датчика ""{sensor.DisplayName}"" перевищило критичне!";
            if (sensor.IsAnalog)
            {
                message += $"Поточне значення: {data.Data}";
```

```
            }
            await this._firebaseProvider.SendToAllNotification(message,
"КРИТИЧНЕ ЗНАЧЕННЯ");
            AddCache(sensor.Id.ToString(), new CriticalSensorValueTimeout
            {
               Id = sensor.Id.ToString(),
               Date = DateTime.Now
            }, TimeSpan.FromSeconds(5));
        }
```

3.2.5. Спілкування сервера з мобільним додатком

Щоб користувач на мобільному додатку отримувати актуальні дані в будь-який момент часу, на стороні сервера було реалізовано *Web API*. *Web API* – це інтерфейс прикладного програмування (*Application Programming Interfaces*, *API*) – це готові конструкції мови програмування (методи, функції), які можна викликати за допомогою *HTTP* протоколу.

Нижче наведено ділянку коду *APIController*, з набором методу, які викликаються з боку мобільного додатка

```
        [Route("api/[controller]")]
        [ApiController]
        public class DataController : ControllerBase
        {
    [HttpGet, Route("sensor-data")]
    public async Task<IActionResult> GetSensorsData()
```

```
    {
        IEnumerable<SensorData> sensorsData = await
this._sensorProvider.GetAllSensorsData();
        return Ok(sensorsData);
    }
    [HttpGet, Route("sensor-data/{sensorId}")]
    public async Task<IActionResult> GetSensorData(string sensorId)
    {
        IEnumerable<SensorData> sensorsData = await
this._sensorProvider.GetAllSensorsData(sensorId);
        return Ok(sensorsData);
    }
    [HttpGet, Route("sensors")]
    public async Task<IActionResult> GetSensors()
      {
        IEnumerable<SensorModel> sensors = await
this._sensorProvider.GetAllSensors();
        return Ok(sensors);
      }
      }
```

3.3. Розробка мобільного додатку

Функціонально мобільний додаток може тільки відображати дані, отримані від сервера – це відображає суть архітектури тонкого клієнта.

При першому запуску, додаток перевіряє синхронізацію з сервером, якщо синхронізація існує, то програма автоматично отображет акутальние дані у вигляді таблиці з датчиками. В іншому випадку користувачеві буде відображено повідомлення про те, що синхронізація відсутня і кнопку, при натисканні якої буде запущений процес синхронізації. Для того щоб додаток синхронізувати з сервером, він повинен знаходиться в одній локальній мережі. Після обміну даними, додаток запам'ятовує зовнішню *IP* адресу сервера і з'являється можливість виконувати запити поза локальної мережі.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        boolean isSync = false;

        try {
            isSync = SyncAdapter.CheckSync(this);
        }catch (Exception e){
            Log.e("MAIN", "error:" + e.getMessage());
        }

        if(isSync){
            Intent intent = new Intent(this, DataActivity.class);
            startActivity(intent);
        }
        else{
            setContentView(R.layout.activity_main);
        }
    }
}
```

Якщо пристрої синхронізовані, то будуть відображатися дані:

```
public class DataActivity extends AppCompatActivity {

    private SwipeRefreshLayout swipeRefreshLayout;
    private Context context;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_data);
        context = this;
        swipeRefreshLayout = findViewById(R.id.swipeRefreshLayout);
        AsyncTask.execute(() -> {
            DataAdapter.UpdateControllerState(context,
findViewById(R.id.controllerState), swipeRefreshLayout);
            DataAdapter.UpdateData(context,this, swipeRefreshLayout);
        });
        swipeRefreshLayout.setOnRefreshListener(() -> {
            AsyncTask.execute(() -> {
                DataAdapter.UpdateControllerState(context,
findViewById(R.id.controllerState), swipeRefreshLayout);
                DataAdapter.UpdateData(context,this, swipeRefreshLayout);
            });
        });
    }
```

Нижче представлена розмітка сторінки з таблицею даних датчиків:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    tools:context=".DataActivity">
    <TextView
```

```
<TextView
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
  <HorizontalScrollView
    <ScrollView
      <TableLayout
    </ScrollView>
  </HorizontalScrollView>
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

3.4. Висновки до розділу

В даному розділі були описані процеси розробки основних модулів системи та методи, що були застосовані для цього.

Для контролера *Arduino UNO* була розроблена власна прошивка за допомогою *Arduino IDE* та мови *C++*. Були описані принцип роботи контролера та основні ділянки коду.

Також був описаний процес створення програмного забезпечення для блоку управління та мобільного додатку, описано взаємодію між ними.

ВИСНОВКИ

В ході виконнаня допломного проєкту була розроблена система управління безпекою.

Архітектурно система складається з таких рівнів:

– набір вимірювальних приладів (датчики);

– прилад контролю и реєстрації даних (контролер *Arduino*);

– блок управління та обробки даних (персональний комп'ютер);

– програмний модуль (*Web*–сервер);

– база даних;

– мобільний додаток.

Щодо використаних для розробки технологій, то було застосовано наступний інтрументарій:

– апаратне забезпечення – *Arduino UNO, Arduino IDE, C++*;

– блок управління – *C#, ASP.NET Core, ASP.NET Core MVC, Entity Framework Core, SQLite, HTML* 5*, CSS* 3*, JavaSciprt*;

– мобільний додаток – *Android Studio, Java, FCM*.

Створена система відповідає таким критеріям:

– безперервно вичитує дані з контролера;

– надсилає своєчасні сповіщення на клієнтський додаток;

– система є універсальною та розширюваною і сприймає будь-який тип пристрою і датчиків.

В першому розділі було проведено аналіз предметної області дипломного проєкту, тобто існуючі методи управління безпекою в системах розумних будинків

На основі аналізу вже інуючих рішень, їх переваг та недоліків, було сформульовано постановку задачі для розроблюваного модулю управління безпекою у системах *Smart House*.

В другому розділі були проаналізовані та обрані всі необхідні технології для проєктування системи управління безпекою.

Для кожного основного елемента були описані і створені діаграма станів та *use-case* діаграма.

В тертьому розділі були описані процеси розробки основних модулів системи та методи, що були застосовані для цього. Також для контролера *Arduino UNO* була розроблена власна прошивка.

Результати виконання дипломної роботи слід використовувати при проектуванні та реалізації програмно-апартних систем на основі контролеру *Arduino UNO*, а розроблену систему як модуль управління безпекою у системах розумногодому.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.
2. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.
3. ГОСТ 2.106-96 ЕСКД “Текстовые документы”.
4. Дронов В. *HTML5, CSS3* и *Web2.0*. Разработка современных *Web*-сайтов. – БХВ-Петербург, 2011. – 414 с.
5. Рихтер Д. *CLR via C#*. Программирование на платформе *Microsoft .NET Framework* 4.5 на языке *C#*. – Питер 2017. – 896 с.
6. Флэнаган Д.. *JavaScript*. Подробное руководство (6-е издание). – СПб: Символ-Плюс, 2008. – 992 с.
7. Фримен А. *ASP.NET MVC5* с примерами на *C#* 5.0 для профессионалов. – Вильямс, 2015. - 736 с.
8. Фленов М. Библия *C#*. – СПб.: БХВ-Петербург, 2011. – 560 с.
9. Троелсен Э. Язык программирования *C#* 5.0 и платформа *.NET* 4.5. – Вильямс, 2015. – 1312 с.
10. Нейгел К. *C#* 2005 и платформа *.NET* 3.0 для профессионалов. / К. Нейгел, Б. Ивьен, Д. Глинн, М. Скиннер, К. Уотсон. – М.: Вильямс, 2008. – 1376 с.
11. Макфарланд Д. Большая книга *CSS3*. Питер, 2014. – 608 с.
12. Буров Є. Комп'ютерні мережі. Магнолія 2010. – 262 с.
13. Фримен А. *jQuery* для профессионалов. – Вильямс, 2012 – 960 с.
14. *Spurlock J. Bootstrap. Responsive Web-Development. – O'Reilly,* 2013. – 128 с.
15. Резиг Д., Фергюсон Р., Пакстон Д. *JavaScript* для профессионалов. Вильямс, 2016. – 231 с.

ДОДАТОК А

ВИХІДНИЙ КОД МОДУЛЮ КЕРУВАННЯ КОНТРОЛЕРОМ

```
public ControllerResponse SendCommand<T>(ControllerCommand command, T data)
where T : class
{
                SerialPort port = this.ConnectToSerialPort();
                if(port == null)
                {
                   this.Log("Cannot connect to controller", LogType.Error);
                   return new ControllerResponse { Success = false };
                }
                try
                {
                   var message = JsonConvert.SerializeObject(new { Command =
command, Data = data });
                   port.WriteLine(message);
                }
                catch (Exception e)
                {
                   this.Log(e.Message, LogType.Error);
                }

                try
                {
                   string responseJson = this.ReadCommand(port);

                   if (string.IsNullOrEmpty(responseJson))
                   {
```

```
                return new ControllerResponse { Success = false };
            }

            ControllerResponse response =
JsonConvert.DeserializeObject<ControllerResponse>(responseJson);
            return response;
        }
        catch (Exception)
        {
            return new  ControllerResponse { Success = false };
        }
    }

    public StateModel CheckStatus()
    {
        SerialPort port = this.ConnectToSerialPort();

        if (port == null)
        {
            return new StateModel { ControllerState =
ControllerState.Disconnected };
        }
        if (port.IsOpen)
        {
            return new StateModel { ControllerState = ControllerState.Connected,
Port = port.PortName };
        }
        else
        {
            this.RemoveCache(SerialPortCacheKey);
```

```
                return new StateModel { ControllerState =
ControllerState.Disconnected, Port = port.PortName };
            }
        }

        #region Private methods
        private string ReadCommand(SerialPort port)
        {
            string result = string.Empty;

            do
            {
                result = port.ReadLine()?.Trim('\r');
            } while (string.Equals(result, AwaitCommandMarker,
StringComparison.InvariantCultureIgnoreCase));

            return result;
        }

        private SerialPort ConnectToSerialPort()
        {
            SerialPort serialPort;

            if (this.Cache(SerialPortCacheKey, out serialPort))
            {
                return serialPort;
            }

            IEnumerable<string> ports = SerialPort.GetPortNames();
```

```
            foreach (string port in ports)
            {
                serialPort = this.PingAndConnectPort(port);

                if (serialPort == null)
                {
                    continue;
                }

                this.AddCache(SerialPortCacheKey, serialPort);

                return serialPort;
            }

            return null;
        }

        private SerialPort PingAndConnectPort(string port)
        {
            SerialPort serialPort = new SerialPort(port, BaudRate);

            serialPort.ReadTimeout = ReadTimeOut;

            try
            {
                serialPort.Open();

                var message = JsonConvert.SerializeObject(new { Command =
ControllerCommand.Ping, Data = PingRequest });
```

```
                serialPort.WriteLine(message);

                string responseJson = this.ReadCommand(serialPort);

                if (string.IsNullOrEmpty(responseJson))
                {
                    serialPort.Close();

                    return null;
                }

                ControllerResponse response =
JsonConvert.DeserializeObject<ControllerResponse>(responseJson);

                if (response.Success && string.Equals(response.Data, PingResponse,
StringComparison.InvariantCultureIgnoreCase))
                {
                    return serialPort;
                }
            }
            catch (Exception e)
            {
                this.Log(e.Message, LogType.Info);

                serialPort.Close();

                return null;
            }
```

```
            serialPort.Close();

            return null;
        }
        #endregion
    }
}
        private FirebaseApp App
        {
            get
            {
                if (Cache<FirebaseApp>(FirebaseAppCache, out FirebaseApp app))
                {
                    return app;
                }

                var newApp = FirebaseApp.Create(new AppOptions()
                {
                    Credential = GoogleCredential.FromFile("google-
service.json").CreateScoped("https://www.googleapis.com/auth/cloud-platform"),
                });

                AddCache(FirebaseAppCache, newApp);

                return newApp;
            }
        }

        private FirebaseMessaging Messaging
        {
```

```
            get
            {
                if (Cache<FirebaseMessaging>(FirebaseMessagingCache, out
FirebaseMessaging messaging))
                {
                    return messaging;
                }

                FirebaseMessaging newMessaging =
FirebaseMessaging.GetMessaging(App);

                AddCache(FirebaseMessagingCache, newMessaging);

                return newMessaging;
            }
        }
        public async Task SendNotification(string message, string subject, string
token)
        {
            Message fcmMessage = new Message()
            {
                Notification = new Notification()
                {
                    Title = subject,
                    Body = message,
                },
                Token = token,
            };
            await Messaging.SendAsync(fcmMessage);
        }
```