

Politechnika Wrocławskaw

Wydział Elektroniki, Fotoniki i Mikrosystemów

Kierunek: Automatyka i robotyka

Specjalność: Przemysł 4.0

Praca dyplomowa inżynierska

Zdalne monitorowanie i sterowanie procesami za pomocą
aplikacji mobilnej

Remote monitoring and control of processes through a mobile
application

Autor:

Mateusz Klimek

Promotor:

Dr inż. Andrzej Jabłoński
(K28W04ND03)

Wrocław 2023

Spis treści

1 Wstęp	7
1.1 Telemetria w zdalnym sterowaniu procesami	7
1.2 Cel projektu	8
1.3 Zakres projektu	8
1.4 Koncepcja projektu	9
2 Projekt systemu telemetrycznego	10
2.1 Założenia projektowe	10
2.1.1 Wymagania funkcjonalne	10
2.1.2 Wymagania niefunkcjonalne	10
2.2 Projekt modelu demonstracyjnego	11
2.2.1 Algorytm działania	11
2.2.2 Dobór urządzeń	12
2.2.3 Czujniki DHT11 i DS18B20	13
2.2.4 Sygnalizacja diodami LED	14
2.2.5 BH1750 w magistrali I2C	15
2.2.6 Układ zasilania wentylatora	16
2.2.7 Projekt układu elektronicznego modelu szklarni	18
2.3 Projekt interfejsu aplikacji	20
2.4 Architektura aplikacji	22
2.5 Baza danych	22
3 Implementacja	24
3.1 Baza danych	24
3.2 Komunikacja mikrokomputera z bazą danych	28
3.3 Integracja czujników i odczytu danych w Pythonie	29
3.3.1 i2c_communication.py	29
3.3.2 read_sensors.py	31
3.3.3 main.py	32
3.4 Struktura aplikacji mobilnej	41

3.5	Moduł autentykacji	41
3.6	Ekran podglądu	45
3.7	Ekran sterowania	46
3.8	Ekran statystyk	47
3.9	Ekran ustawień	50
3.10	Komunikacja aplikacji mobilnej z bazą danych	51
3.11	Problemy implementacyjne	53
3.12	Wdrożenie i testy	53
4	Podsumowanie	55
4.1	Podsumowanie pracy	55
4.2	Dalszy rozwój projektu	55
5	Wykaz elementów	57
	Bibliografia	58

Spis rysункów

1.1	Schemat koncepcyjny projektu	9
2.1	Schemat elektroniczny podłączenia czujników DHT11 oraz DS18B20 . . .	13
2.2	Schemat elektroniczny modułu sygnalizującego	14
2.3	Schemat elektroniczny połączenia czujników natężenia oświetlenia w ma- gistralach dwukierunkowej	15
2.4	Schemat elektroniczny układu zasilania wentylatora	17
2.5	Schemat elektroniczny układu symulującego zautomatyzowaną szklarnię .	19
2.6	Schemat nawigacji części odpowiedzialnej za uwierzytelnianie	20
2.7	Schemat nawigacji części głównej	21
3.1	Struktura przechowywania pomiarów szklarni	25
3.2	Strukutra przechowywania pomiarów stanowisk	26
3.3	Przechowywanie szablonów nastaw wartości progowych	26
3.4	Przechowywanie aktualnie używany szablon	27
3.5	Przechowywanie danych do wyświetlania statystyk	28

3.6 Inicializacja połączenia z bazą danych	28
3.7 Zawartość skryptu i2c_communication.py przedstawiającainicjalizacjema-	
gistral	29
3.8 Zawartość skryptu i2c_communication.py	30
3.9 Zawartość skryptu read_sensors.py przedstawiająca inicializację	31
3.10 Zawartość skryptu read_sensors.py przedstawiająca funkcję main()	32
3.11 Zawartość skryptu main.py przedstawiająca inicializację pinów GPIO . . .	32
3.12 Zawartość skryptu main.py przedstawiająca inicializację zmiennych . . .	33
3.13 Zawartość skryptu main.py przedstawiająca inicializację referencji do sek-	
torów bazy danych	33
3.14 Zawartość skryptu main.py przedstawiająca funkcję init_thresholds() . . .	34
3.15 Zawartość skryptu main.py przedstawiająca funkcje nasłuchujące	35
3.16 Zawartość skryptu main.py przedstawiająca funkcję porównującą czas . .	35
3.17 Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzają-	
cej logiką modelu demonstracyjnego	36
3.18 Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzają-	
cej logiką modelu demonstracyjnego	37
3.19 Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzają-	
cej logiką modelu demonstracyjnego	37
3.20 Zawartość skryptu main.py przedstawiająca funkcję przesyłającą pomiary	
do bazy danych	38
3.21 Zawartość skryptu main.py przedstawiająca funkcję przesyłającą średnie	
pomiarów do bazy danych	39
3.22 Zawartość skryptu main.py przedstawiająca główną pętle programu . . .	40
3.23 Zrzut ekranu terminalu ukazujący wykonanie dwóch cykli głównej pętli	
programu	40
3.24 Zrzut ekranu przedstawiający strukturę aplikacji mobilnej	41
3.25 Ekran logowania	42
3.26 Ekran rejestracji	42
3.27 Obsługa błędów logowania	43
3.28 Obsługa błędów rejestracji	43

3.29 Fragment skryptu LoginActivity.kt przedstawiający funkcje logowania	44
3.30 Fragment skryptu LoginActivity.kt przedstawiający funkcje onStart oraz updateUI	44
3.31 Ekran podglądu	45
3.32 Ekran sterowania	46
3.33 Ekran statystyk przedstawiający wykres wilgotności oraz natężenia światła w układzie godzinowym	48
3.34 Ekran statystyk przedstawiający wykres temperatury oraz natężenia światła słonecznego w układzie godzinowym.	48
3.35 Ekran statystyk przedstawiający wykres wilgotności oraz natężenia światła w układzie tygodniowym	49
3.36 Ekran statystyk przedstawiający wykres temperatury oraz natężenia światła słonecznego w układzie tygodniowym	49
3.37 Ekran ustawień z wybranym motywem ciemnym	50
3.38 Fragment kodu przedstawiający funkcję nasłuchującą zmiane wartości . .	51
3.39 Fragment kodu przedstawiający funkcję zapisującą wartości	52
3.40 Fragment kodu przedstawiający funkcję pobierającą wartości	52

1. Wstęp

1.1. Telemetria w zdalnym sterowaniu procesami

Rozwój przemysłu w historii charakteryzuje się trzema znaczącymi etapami. Pierwszy z nich, zapoczątkowany przez wynalezienie i upowszechnienie silnika parowego, inaugurował erę industrializacji. Kolejny przełom nastąpił dzięki elektryfikacji. Pojawienie się prądu elektrycznego przyspieszyło procesy produkcyjne, prowadząc do powstania pierwszych linii produkcyjnych i umożliwiając masową produkcję towarów. Trzecia faza, cyfryzacja, nastąpiła wraz ze wzrostem wydajności komputerów i systemów sterowania. Automatyzacja części lub całych procesów produkcyjnych w tym okresie doprowadziła do zmniejszenia kosztów i przyspieszenia produkcji.

Obecnie jesteśmy świadkami czwartej rewolucji przemysłowej. Rozprzestrzenienie internetu umożliwia rozbudowę procesu produkcyjnego poprzez stałe połączenie z siecią, co zapewnia ciągłą komunikację między urządzeniami bez względu na ich lokalizację.

Telemetria, czyli dziedzina zajmująca się zdalnym przesyłaniem wartości pomiarowych, odgrywa kluczową rolę w tym procesie. Dzięki rozwojowi technologii, możliwe jest przesyłanie danych na zarówno krótkie, jak i długie dystanse, wykorzystując do tego komunikację kablową, systemy radiowe czy internet. Taka komunikacja w czasie rzeczywistym pomiędzy urządzeniami pomiarowymi a centralą sterującą pozwala na szybsze i bardziej efektywne reagowanie na zmiany w procesie produkcyjnym. Systemy oparte na tej koncepcji mogą funkcjonować automatycznie, z ograniczonym udziałem człowieka, który pełni rolę nadzorującą lub decyzyjną w zakresie ustawień parametrów procesu.

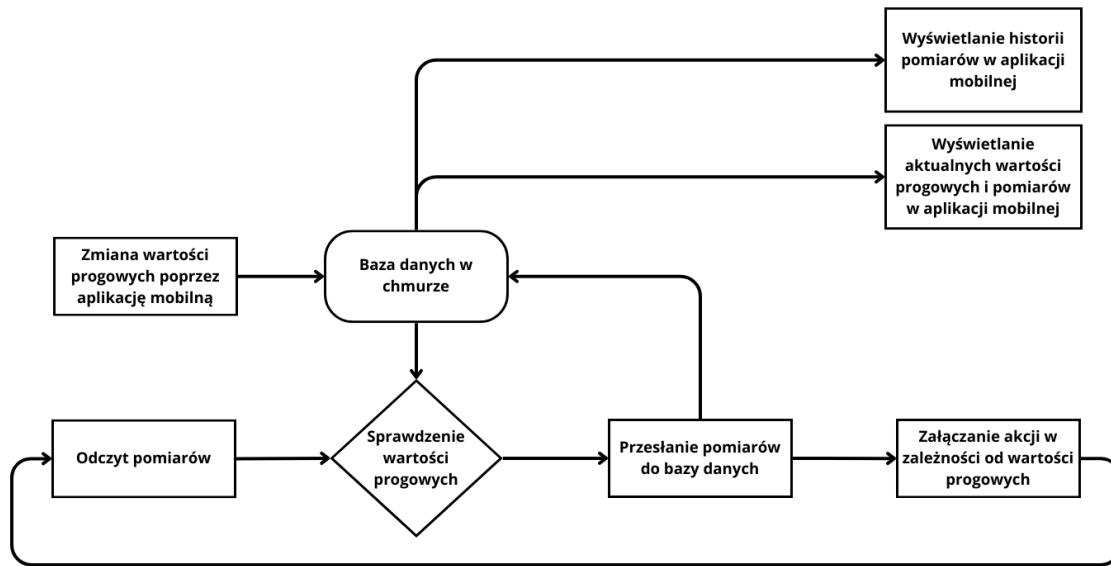
1.2. Cel projektu

Celem niniejszej pracy inżynierskiej jest implementacja zaawansowanego systemu umożliwiającego zdalne monitorowanie i sterowanie procesami za pośrednictwem aplikacji mobilnej. Kluczowym elementem projektu jest opracowanie układu opartego o komputer jednopłytkowy Raspberry Pi 4B. Układ ten będzie odpowiedzialny za zbieranie danych z czujników, ich przetwarzanie w celu aktywacji określonych działań, a także za komunikację w czasie rzeczywistym z bazą danych. Interfejs użytkownika zostanie zrealizowany poprzez aplikację mobilną na system Android, opracowaną w języku Kotlin. Aplikacja ta zapewni możliwość monitorowania parametrów w czasie rzeczywistym, wglądu w historię danych pomiarowych oraz zdalnego sterowania kluczowymi ustawieniami algorytmu systemu.

1.3. Zakres projektu

Projekt zakłada stworzenie kompletnego rozwiązania, obejmującego zarówno projekt aplikacji, jak i stację pomiarową, z perspektywą pełnego wdrożenia do rzeczywistych zastosowań. Dodatkowo, aby lepiej odzwierciedlić praktyczne możliwości systemu, w projekcie zostanie przyjęty model działania oparty na zautomatyzowanej szklarni, co pozwoli na zastosowanie opracowanego systemu w realnych warunkach, jednocześnie demonstrując jego elastyczność i adaptowalność do różnorodnych scenariuszy.

1.4. Koncepcja projektu



Rysunek 1.1: Schemat koncepcyjny projektu

Na rysunku 1.1 zaprezentowano schemat funkcjonowania modelu szklarni oraz przepływ informacji pomiędzy jego komponentami. Model demonstracyjny, pełniący rolę środowiska produkcyjnego, utrzymuje dwukierunkową komunikację z bazą danych w chmurze. Umożliwia to użytkownikowi dostęp do aktualnych oraz archiwalnych danych pomiarowych za pośrednictwem aplikacji mobilnej, a także pozwala na modyfikację klu-czowych progowych wartości procesowych.

2. Projekt systemu telemetrycznego

2.1. Założenia projektowe

2.1.1. Wymagania funkcjonalne

- Odczyt pomiarów z czujników co ustalony okres 5 sekund.
- Analiza i przetwarzanie odczytanych danych.
- Sterowanie akcjami w zależności od stosunku aktualnych pomiarów do nastawionych progów.
- Przesyłanie wartości pomiarów do bazy danych co 10 sekund.
- Zliczanie średniej wartości pomiarów co godzine i przesyłanie jej do bazy danych.
- Połączenie aplikacji mobilnej z bazą danych Firebase za pomocą internetu.
- Rejestracja, logowanie, wylogowanie i możliwość usunięcia konta w aplikacji mobilnej.
- Podgląd aktualnych wartości pomiarowych i nastawionych progów w aplikacji mobilnej.
- Możliwość zmiany progów odpowiadających za poszczególne akcje z poziomu aplikacji.
- Wyświetlanie statystyk pomiarów z okresu dnia oraz tygodnia.

2.1.2. Wymagania niefunkcjonalne

- Stała komunikacja w czasie rzeczywistym między aplikacją mobilną i Raspberry Pi a bazą danych
- Bezpieczeństwo danych przesyłanych między urządzeniami a bazą danych
- Intuicyjna i łatwa w obsłudze aplikacja mobilna
- Wysoka skalowalność systemu

2.2. Projekt modelu demonstracyjnego

2.2.1. Algorytm działania

Realizacja projektu rozpoczyna się od ustalenia algorytmu, który będzie sterował działaniem modelu demonstracyjnego. Model będzie obejmował dwa oddzielne stanowiska. Każde stanowisko wyposażone jest w czujnik natężenia światła i wilgotności. Aby uniknąć rozbudowy modelu o donice z ziemią, w celach demonstracyjnych zostaną wykorzystane czujniki wilgotności powietrza, które efektywnie symulują warunki danego stanowiska. Warunki panujące w szklarni będą odzwierciedlane poprzez zbierane dane dotyczące temperatury powietrza oraz natężenia światła słonecznego.

Zgromadzone dane będą poddawane analizie w oparciu o wyznaczone przez użytkownika wartości progowe, według poniższego schematu:

- Wentylator: Gdy zmierzona temperatura przekroczy ustaloną przez użytkownika wartość progową, system aktywuje wentylator. Wentylator zostanie wyłączony, gdy temperatura obniży się poniżej tego progu.
- Podlewanie: Jeżeli poziom wilgotności na danym stanowisku spadnie poniżej zadanej granicy, to niebieska dioda LED zostanie włączona, sygnalizując potrzebę nawodnienia roślin. Dioda zostanie wyłączona, gdy wilgotność wzrośnie ponad określony próg.
- Lampy: W sytuacji, gdy zmierzone natężenie światła będzie niższe niż minimalna wartość określona przez użytkownika, na danym stanowisku zostanie włączona żółta dioda LED. Dioda zostanie wyłączona, gdy poziom światła osiągnie lub przekroczy ustalony próg.
- Rolety: Przekroczenie przez natężenie światła słonecznego maksymalnej wartości, ustawionej przez użytkownika, spowoduje zaświecenie czerwonej diody LED, co będzie symbolizować opuszczenie rolet w szklarni. Gdy natężenie światła słonecznego spadnie poniżej wyznaczonego progu dioda LED zgaśnie.

2.2.2. Dobór urządzeń

Stworzenie środowiska symulacyjnego dla zautomatyzowanej szklarni wymaga wyboru odpowiedniego komputera jednopłytkowego zdolnego do obsługi kompletu elementów składających się na system. Niezbędne komponenty to:

- 2x czujnik wilgotności DHT11
- 1x czujnik temperatury DS18B20
- 3x czujnik natężenia światła BH1750
- 5x dioda LED
- 1x silnik DC 5V

Wybrany mikrokomputer musi również zapewniać stabilne połączenie z internetem oraz wystarczającą wydajność do efektywnego odczytu, przetwarzania i przesyłania danych pomiarowych do bazy danych.

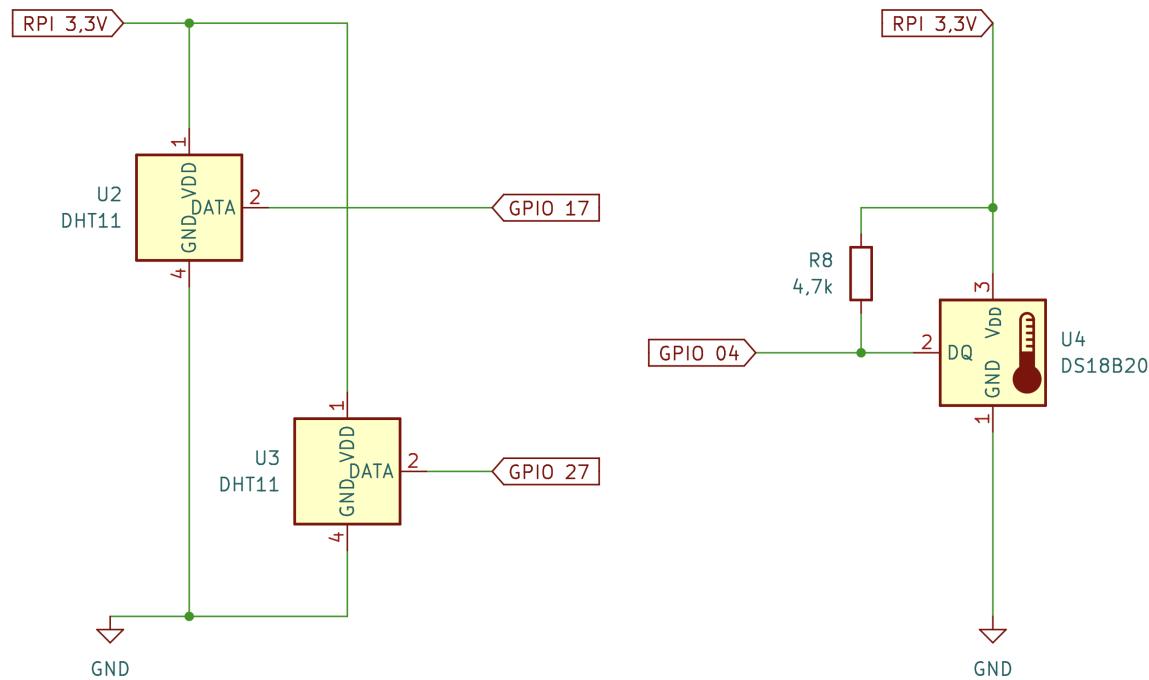
Najlepszym wyborem wydaje się być Raspberry Pi 4B, charakteryzujący się 26 pinami GPIO (General-purpose input/output), co ułatwia podłączenie czujników DHT11, DS18B20 oraz diód LED. Wyzwaniem jest sterowanie silnikiem DC 5V, gdyż piny GPIO Raspberry Pi dostarczają maksymalnie 3,3V przy prądzie do 16mA. Aby zasilić wentylator pinem GPIO, konieczne jest użycie elementu pośredniego w postaci tranzystora N-MOSFET IRLZ44N, który efektywnie pracuje przy niskim napięciu progowym bramki (1-2V).

Kolejnym wyzwaniem jest podłączenie trzech czujników natężenia światła BH1750 komunikujących się przez magistralę I2C. Raspberry Pi oferuje jedną domyślnie dostępną magistralę I2C, jednak drugą można aktywować poprzez zmianę ustawień w plikach konfiguracyjnych. Dodatkowo, czujniki BH1750 umożliwiają zmianę adresu wykorzystywanego w komunikacji I2C, co pozwala na podłączenie dwóch identycznych czujników do tej samej magistrali. Dzięki temu unika się konieczności rozbudowy systemu o dodatkowe moduły, takie jak Arduino czy ekspandery wyprowadzeń I2C, co mogłoby niepotrzebnie skomplikować projekt.

2.2.3. Czujniki DHT11 i DS18B20

Czujniki wilgotności DHT11 są zasilane z linii 3,3V Raspberry Pi. Komunikacja z mikrokomputerem odbywa się przez piny GPIO 17 oraz GPIO 27, co przedstawiono na rysunku 2.1. Wyprowadzenia GND każdego z czujników DHT11 są podłączone do uziemienia (GND) Raspberry Pi.

Czujnik temperatury DS18B20 również czerpie zasilanie z linii 3,3V Raspberry Pi i jest połączony z GND. Komunikacja odbywa się poprzez protokół 1-Wire, dlatego wyjście DQ czujnika podłączone jest do pinu GPIO 04, który domyślnie służy do obsługi tego protokołu. Protokół 1-Wire umożliwia przesyłanie danych za pomocą jednego przewodu, wykorzystując impulsy do reprezentowania stanów logicznych "1" i "0". Dla zapewnienia niezawodności transmisji, między wyjściem DQ a VDD czujnika, dołączony jest rezystor „pull-up” o wartości $4,7\text{k}\Omega$. Rezystor ten minimalizuje wpływ szumów oraz zakłóceń elektromagnetycznych na linii sygnałowej DQ.

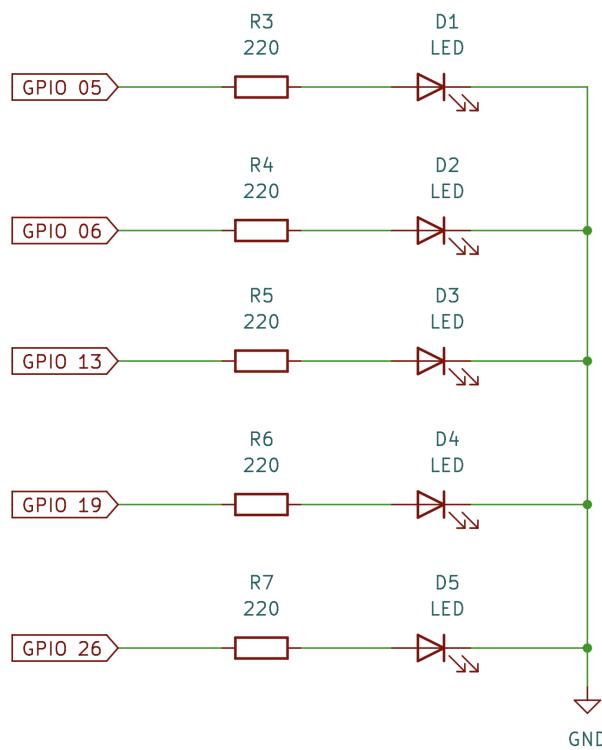


Rysunek 2.1: Schemat elektryczny podłączenia czujników DHT11 oraz DS18B20

2.2.4. Sygnalizacja diodami LED

Moduł sygnalizacyjny wykorzystuje pięć diod LED, z których każda jest kontrolowana przez dedykowany pin GPIO w mikrokomputerze Raspberry Pi:

- Dioda niebieska D1 - GPIO 05.
- Dioda niebieska D2 - GPIO 06.
- Dioda żółta D3 - GPIO 13.
- Dioda żółta D4 - GPIO 19.
- Dioda czerwona D5 - GPIO 26.

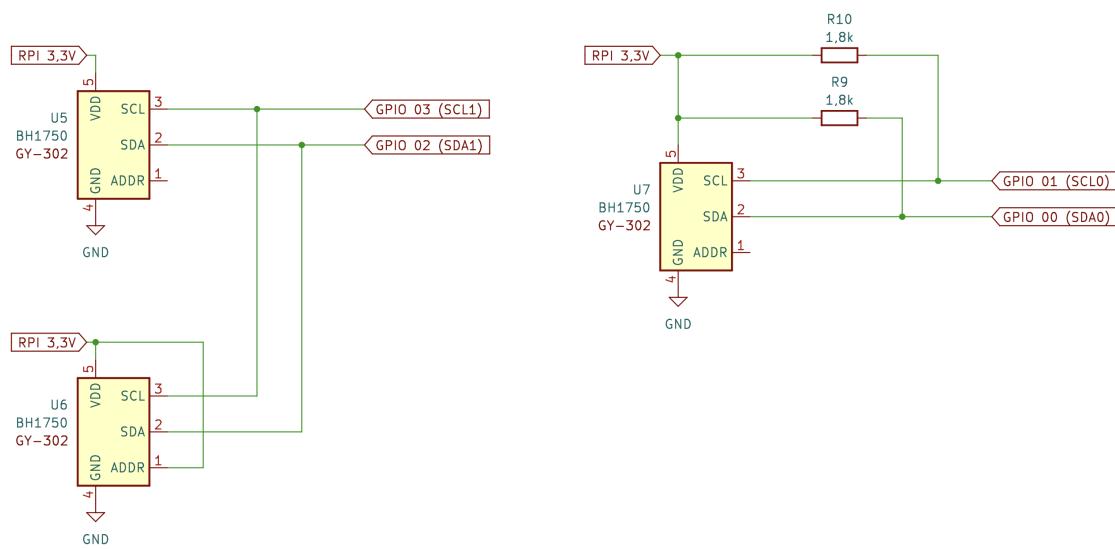


Rysunek 2.2: Schemat elektryczny modułu sygnalizującego

Zgodnie z rysunkiem 2.2 każda dioda LED połączona jest szeregowo z rezystorem ograniczającym prąd o wartości 220Ω , co zapobiega przepływowi zbyt dużego prądu skutkującego uszkodzeniem diody lub/i mikrokomputera. Wyprowadzenie katodowe każdej diody LED jest wspólne i podłączone do uziemienia (GND) Raspberry Pi.

2.2.5. BH1750 w magistrali I2C

I2C (Inter-Integrated Circuit) to dwuprzewodowa, szeregową magistralą komunikacyjną, która jest używana do połączenia urządzeń peryferyjnych z mikrokontrolerem lub komputerem w układzie wbudowanym. Ten interfejs pozwala na efektywną wymianę danych pomiędzy urządzeniami slave a centralnym urządzeniem master. Każde urządzenie podłączone do magistrali I2C musi mieć **unikalny** adres, który służy do jego identyfikacji i komunikacji. W typowej konfiguracji I2C, linia SDA (Serial Data Line) jest używana do przesyłu danych, natomiast linia SCL (Serial Clock Line) dostarcza sygnał zegara synchronizujący transmisję danych między urządzeniami.



Rysunek 2.3: Schemat elektroniczny połączenia czujników natężenia oświetlenia w magistralach dwukierunkowej

Na schemacie widać, że do Raspberry Pi podłączone są trzy czujniki BH1750. Wykorzystany czujnik natężenia światła umożliwia zmianę domyślnego adresu 0x23, na 0x5c poprzez podanie stanu wysokiego na pin ADDR czujnika. Stosując taki zabieg możliwym jest podłączenie dwóch takich samych czujników do jednej magistrali I2C. Analizując schemat z rysunku 2.3, począwszy od lewej strony, zasilanie oraz uziemienie

czujników U5 i U6 są odpowiednio podłączone do linii 3,3V oraz do uziemienia Raspberry Pi. Pin ADDR czujnika U6 jest zwarty z zasilaniem 3,3V, zapewniając zmianę jego adresu na alternatywny. Dzięki temu piny SCL oraz SDA są połączone do wspólnej magistrali I2C1 - odpowiadającej pinom GPIO 03 (SCL) oraz GPIO 02 (SDA), która jest domyślnym interfejsem I2C Raspberry Pi.

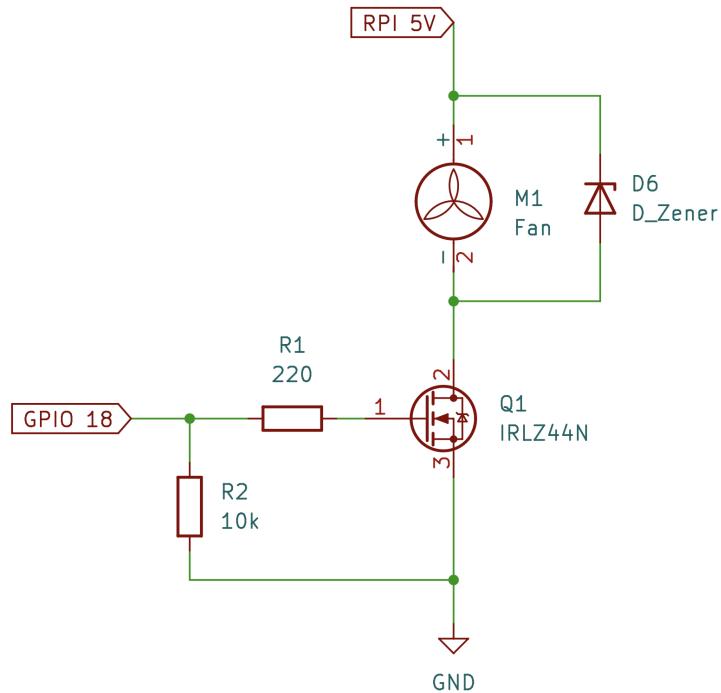
Trzeci czujnik U7 ze względu na kolidujący adres nie może być używany w tej samej magistrali. Wykorzystana zostanie druga magistrala I2C, którą należy aktywować w plikach konfiguracyjnych systemu mikrokomputera. Czujnik podpięty jest analogicznie jak U5 i U6, z uwzględnieniem wykorzystania magistrali I2C0, odpowiadającej pinom GPIO 00 (SDA) oraz GPIO 01 (SCL). Przy czujniku U7 konieczne jest dołączenie zewnętrznych rezystorów „pull-up” R9 i R10 do linii SDA i SCL, ponieważ nie są one wbudowane w układ Raspberry Pi, tak jak w przypadku domyślnej magistrali I2C1. Dołączenie tych rezystorów zapewnia stabilny poziom wysoki na liniach magistrali, gdy żadne urządzenie nie przesyła sygnału niskiego.

To rozwiązanie pozwala na efektywną i elastyczną komunikację z wieloma urządzeniami peryferyjnymi, minimalizując jednocześnie ilość potrzebnych przewodów i złącz.

2.2.6. Układ zasilania wentylatora

Na schemacie przedstawiono układ sterowania wentylatorem (Fan) przy użyciu mikrokomputera Raspberry Pi poprzez tranzystor typu N-MOSFET (Q1 IRLZ44N). Tranzystor jest sterowany przez Raspberry Pi poprzez dostarczający sygnał sterujący pin GPIO 18, do którego podłączony jest rezystor R1 o wartości 220Ω . Rezystor R2 o wartości $10k\Omega$ jest podłączony między bramkę a źródło tranzystora, pełniąc funkcję rezystora podciągającego do masy, co zapobiega niepożądanemu przełączaniu tranzystora przez szumy.

Wentylator jest zasilany z 5V DC dostarczanego przez Raspberry Pi i jest połączony szeregowo z diodą Zenera (D6), która służy jako zabezpieczenie przed przepięciami, mogącymi powstać podczas wyłączania wentylatora. Dioda Zenera pozwala na rozpraszanie nadmiaru napięcia i chroni układ przed uszkodzeniem.



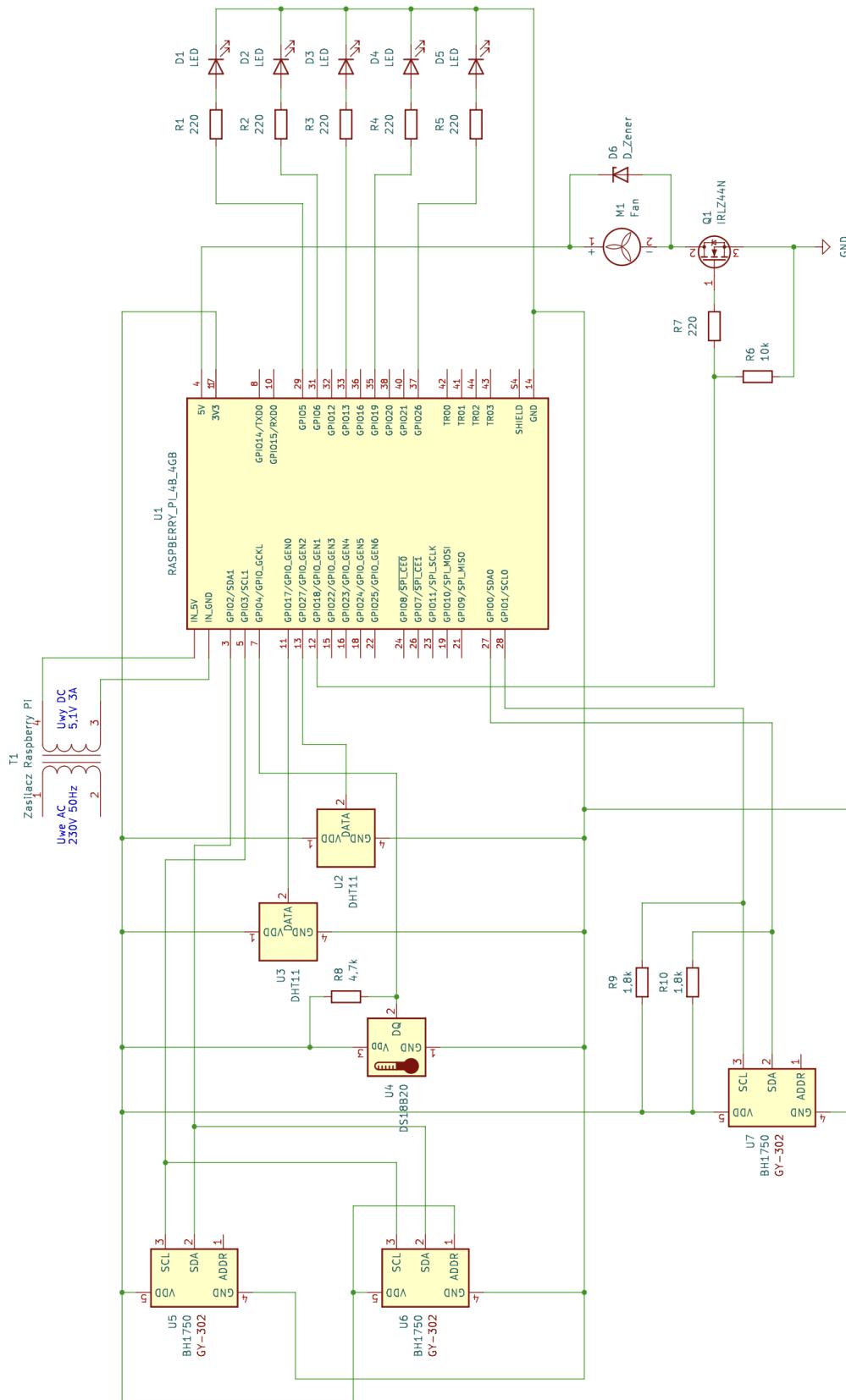
Rysunek 2.4: Schemat elektroniczny układu zasilania wentylatora

Tranzystor MOSFET w tym układzie działa jak elektroniczny przełącznik, który włącza lub wyłącza obwód wentylatora w odpowiedzi na sygnał sterujący z GPIO Raspberry Pi. Gdy na bramkę tranzystora (pin 1) podany jest sygnał wysoki, tranzystor przewodzi, a wentylator jest zasilany. Gdy sygnał jest niski, obwód jest przerwany i wentylator się wyłącza. Wykorzystanie tranzystora MOSFET jako elektronicznego przełącznika umożliwia sterowanie obwodem wentylatora 5V za pomocą 3,3V dostarczanych z pinu GPIO Raspberry Pi.

2.2.7. Projekt układu elektronicznego modelu szklarni

Na rysunku 2.5 przedstawiono kompleksowy schemat elektroniczny modelu demonstracyjnego. Schemat ten integruje wszystkie indywidualnie omówione moduły - od czujników wilgotności i natężeń światła, przez systemy sygnalizacji LED, po sterowanie wentylatorem i roletami - tworząc spójny system symulujący warunki zautomatyzowanej szklarni. Kluczowym elementem jest tutaj mikrokomputer Raspberry Pi, który stanowi centrum zarządzania danymi i koordynacji pracy poszczególnych komponentów. Układ czerpie energię z zasilacza komputera jednopłytkowego (T1), który zapewnia napięcie 5,1V przy maksymalnym prądzie 3A.

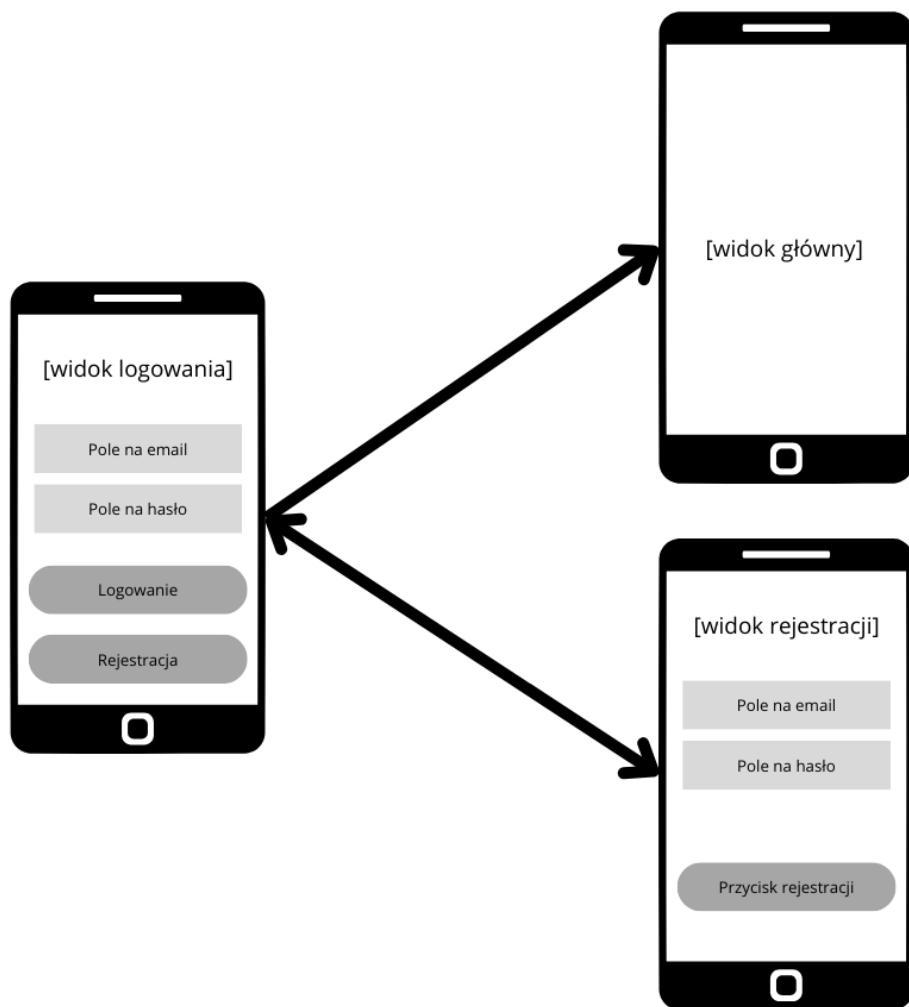
Zaawansowane techniki komunikacji, takie jak I2C, 1-wire oraz precyzyjne sterowanie poszczególnymi elementami, pozwalają na efektywną reakcję systemu na zmieniające się warunki środowiskowe. Dzięki temu, model nie tylko odzwierciedla scenariusze pracy szklarni, ale także demonstruje, jak technologia może być wykorzystana do optymalizacji zachodzących w niej procesów.



Rysunek 2.5: Schemat elektroniczny układu symulującego zautomatyzowaną szklarnię

2.3. Projekt interfejsu aplikacji

Na rysunku 2.6 zaprezentowano schemat modułu uwierzytelniania. Użytkownik początkowo napotyka ekran logowania, gdzie poprzez wprowadzenie prawidłowego loginu i hasła uzyskuje dostęp do głównych funkcji aplikacji. Wybór opcji rejestracji kieruje do interfejsu tworzenia nowego konta. Po wypełnieniu wymaganych pól i potwierdzeniu ich za pomocą przycisku, użytkownik jest przekierowywany z powrotem do ekranu logowania.

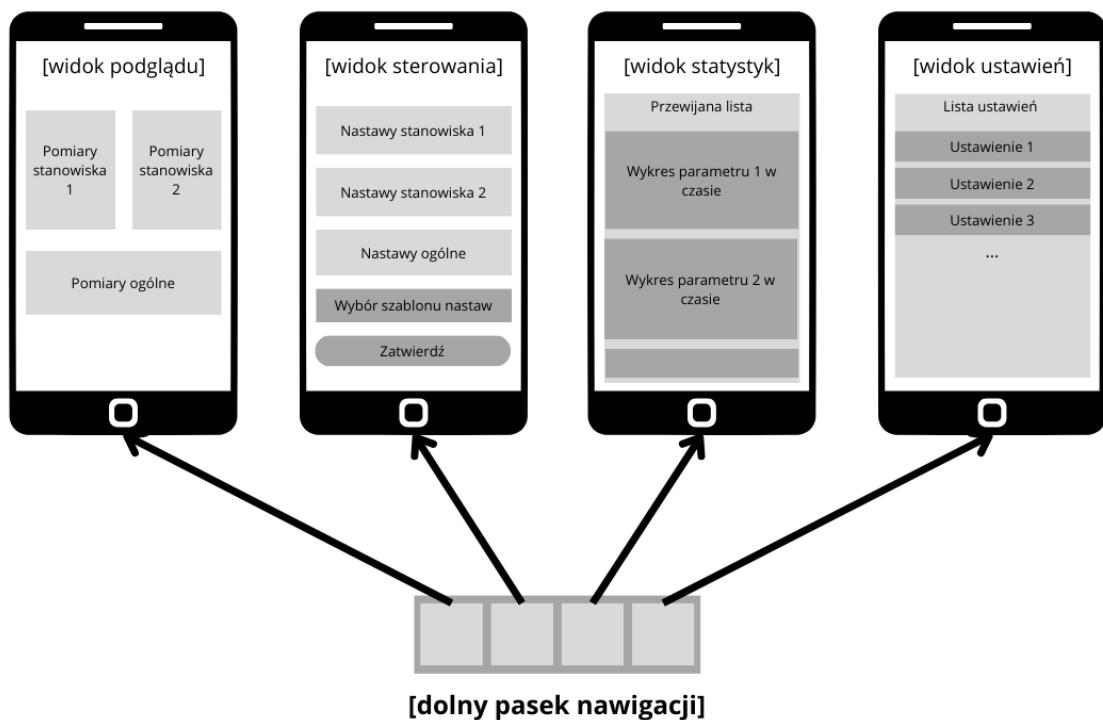


Rysunek 2.6: Schemat nawigacji części odpowiedzialnej za uwierzytelnianie

Zalogowany użytkownik zostanie przeniesiony do **widoku podglądu**, który będzie polegał na wyświetlaniu aktualnych pomiarów w stosunku do wybranych nastaw

wartości progowych. Za pomocą dolnego paska nawigacji użytkownik będzie mógł przemieszczać się pomiędzy widokami zgodnie ze schematem przedstawionym na rysunku 2.7.

Widok sterowania pozwoli użytkownikowi wybrać szablon edytowalnych nastaw wartości progowych odpowiadających za włączenie i wyłączenie lamp, wentylatora czy sygnalizację potrzeby podlania roślin na danym stanowisku. Po wybraniu jednego z czterech szablonów, możliwe (ale nie konieczne) będzie zmienianie aktualnie przypisanych do niego wartości parametrów, co po zatwierdzeniu decyzji ustawi jako obecny wybrany schemat. Zatwierdzenie szablonu ze zmienionymi nastawami automatycznie zapisze go jako zestaw trybu ręcznego.



Rysunek 2.7: Schemat nawigacji części głównej

Trzecim od lewej na pasku nawigującym będzie **widok statystyk**, składający się z przewijanej listy wyświetlającej wykresy przedstawiające zmiany wartości parametrów w czasie. Zakres czasu domyślnie ustawiony na 24 godziny wstecz, będzie możliwy do zmianienia na tydzień w odpowiadającej tej funkcji zakładce **widoku ustawień**. Innymi

głównymi funkcjami zawierającymi się w tej aktywności będzie możliwość wylogowania się i usunięcia przez użytkownika konta.

2.4. Architektura aplikacji

Następnym krokiem projektowania jest wybranie architektury, w której aplikacja będzie tworzona. Analizując określone co do aplikacji wymagania, jako optymalny wybór klaruje się postawienie na rozdzielenie logiki biznesowej od interfejsu użytkownika. Jest to popularne i promowane przez developerów androida rozwiązanie, ułatwiające systemowi zarządzanie cyklem życia aktywności, zapewniające lepszą organizację kodu oraz poprawiające jego skalowalność [4]. Skorzystanie z tego podejścia będzie kluczowe do zapewnienia dobrze skonstruowanej komunikacji z bazą danych w czasie rzeczywistym.

Wykorzystanie dolnego paska do nawigacji między ekranami oparte jest na strukturze, w której główna aktywność(gospodarz) pełni rolę kontenera dla wyświetlanych fragmentów(gości). Zachodzi między nimi relacja gospodarz-gosć, gdzie aktywność jest kontenerem zarządzającym cyklem życia, układem i interakcjami fragmentów. Rozszerzając tą strukturę o rozdzielenie logiki biznesowej od interfejsu użytkownika otrzymamy model ViewModel-Fragment. Dzięki utworzeniu klas ViewModel zapewniona zostanie płynna i efektywna komunikacja z bazą danych, zarówno poprzez jednorazowe pytania, jak i przez ciągłe nasłuchiwanie zmian w bazie danych za pomocą listenerów. Taki sposób komunikacji nie tylko zwiększy responsywność aplikacji, ale również pozwoli na automatyczne aktualizacje interfejsu użytkownika w reakcji na zmiany w bazie danych.

2.5. Baza danych

Wybór Firebase Realtime Database jako centrum danych dla modelu demonstracyjnego wiąże się z wieloma kluczowymi zaletami. Po pierwsze, Firebase oferuje wbudowane szyfrowanie HTTPS dla wszystkich danych przesyłanych między aplikacją a bazą danych, co gwarantuje bezpieczeństwo i prywatność informacji. Szyfrowanie to odgrywa kluczową rolę, zwłaszcza w kontekście rosnącego ryzyka związanego z wyciekiem poufnych informacji produkcyjnych, opisanego szerzej w pozycji [3].

Dodatkowo, Firebase Realtime Database oferuje wysoką wydajność i responsywność

dzięki swojej zdolności do synchronizacji danych w czasie rzeczywistym. Każda zmiana w bazie danych jest natychmiast odzwierciedlana w aplikacji mobilnej, co jest kluczowe dla monitorowania i reagowania na zmieniające się warunki w modelu szklarni [2].

Z punktu widzenia rozwoju aplikacji, Firebase zapewnia też znaczące ułatwienia. Moduł Firebase Authentication, na przykład, umożliwia łatwą implementację systemu rejestracji i logowania użytkowników. Zautomatyzowane zarządzanie kontami i bezpieczne przechowywanie danych użytkowników znacznie upraszcza proces wdrażania funkcji autoryzacji. Ograniczenie dostępu do danych w bazie do zalogowanych użytkowników dodatkowo zwiększa bezpieczeństwo systemu.

Kolejną zaletą wyboru Firebase jest jego skalowalność. Dzięki elastycznemu modelowi cenowemu i możliwości adaptacji do rosnących wymagań projektu, Firebase jest idealnym rozwiązaniem dla aplikacji, które mogą ewoluować wraz z rozwojem projektu.

Podsumowując, Firebase Realtime Database stanowi solidną podstawę do budowy efektywnego i bezpiecznego mostu komunikacyjnego między modelem demonstracyjnym a interfejsem użytkownika, łącząc zaawansowane funkcje techniczne z łatwością użytkowania i elastycznością.

3. Implementacja

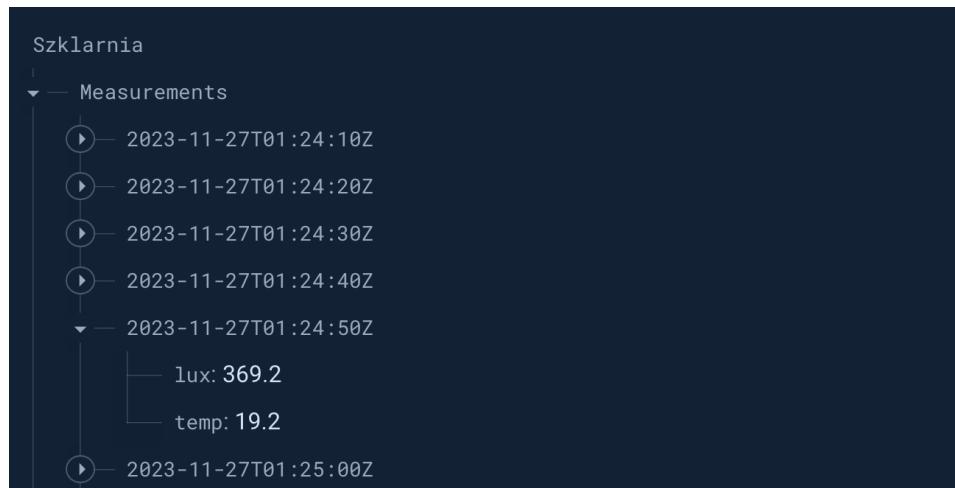
3.1. Baza danych

Baza danych Firebase Realtime Database, wykorzystująca model NoSQL, jest fundamentem modelu demonstracyjnego zautomatyzowanej szklarni. Struktura bazy danych, ukazana na rysunkach, została zaprojektowana, aby odzwierciedlać i obsługiwać hierarchiczną organizację danych w sposób intuicyjny i elastyczny. Dzięki zastosowaniu nierelacyjnego modelu danych NoSQL, baza danych jest podzielona na cztery główne segmenty, które zapewniają optymalną organizację i szybki dostęp do przechowywanych informacji:

- Measurements: Ten segment zawiera wszystkie pomiary z całej szklarni, umożliwiając precyzyjne śledzenie zmian środowiskowych w czasie rzeczywistym. Struktura ta jest kluczowa dla monitorowania stanu szklarni i reagowania na zmieniające się warunki.
- Statistics: Tutaj gromadzone są uśrednione dane, które pozwalają na analizę trendów i wzorców w dłuższym okresie. Jest to kluczowe dla podejmowania decyzji dotyczących długoterminowych strategii zarządzania szklarnią.
- Threshold sets: Segment ten przechowuje zdefiniowane przez użytkownika wartości progowe dla różnych parametrów środowiskowych. Dzięki temu system może automatycznie dostosowywać warunki w szklarni w oparciu o wybrany przez użytkownika zestaw nastaw.
- Workstations: Ostatni segment to struktura, która organizuje pomiary według indywidualnych stanowisk. Pozwala to na szczegółowe zarządzanie i kontrolę warunków dla poszczególnych obszarów szklarni.

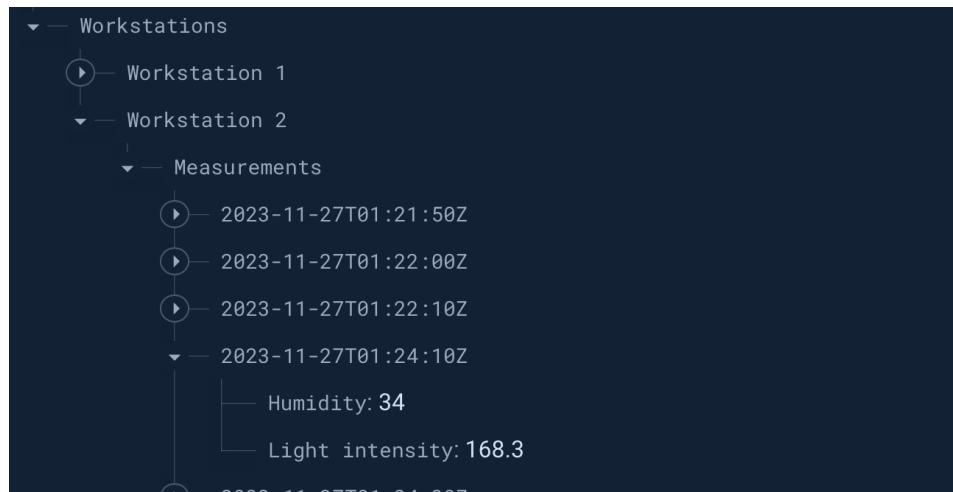
Przejrzystość i modularność struktury bazy danych Firebase Realtime Database są niezbędne dla efektywnego zarządzania danymi w złożonych systemach, takich jak wykorzystywany model szklarni. Zastosowanie nierelacyjnego modelu danych NoSQL umożliwia łatwe skalowanie i dostosowanie systemu do zmieniających się potrzeb i rosnącej ilości danych.

W segmencie Measurements widocznym na rysunku 3.1 przechowywane są pomiary takie jak natężenie światła słonecznego i temperatura, rejestrowane w 10-sekundowych odstępach czasowych. Każdy zapis pomiaru jest oznaczony datą i czasem w formacie ISO 8601, który jest szeroko stosowany w międzynarodowych systemach. Należy jednak zaznaczyć, że w obecnym etapie projektu wszystkie daty są zapisywane jako czas lokalny bez uwzględnienia strefy czasowej, co oznacza, że używana jest lokalna godzina dla Warszawy z literą 'Z' dodaną z konwencji, a nie w celu wskazania czasu UTC. Jest to tymczasowe rozwiązanie, które zostanie dostosowane w przyszłości do pełnej zgodności z konwencją oznaczania czasu UTC, w miarę dalszego rozwoju projektu. Ta decyzja pozwala na zachowanie spójności z istniejącym zestawem danych i ułatwia dalsze prace nad projektem.



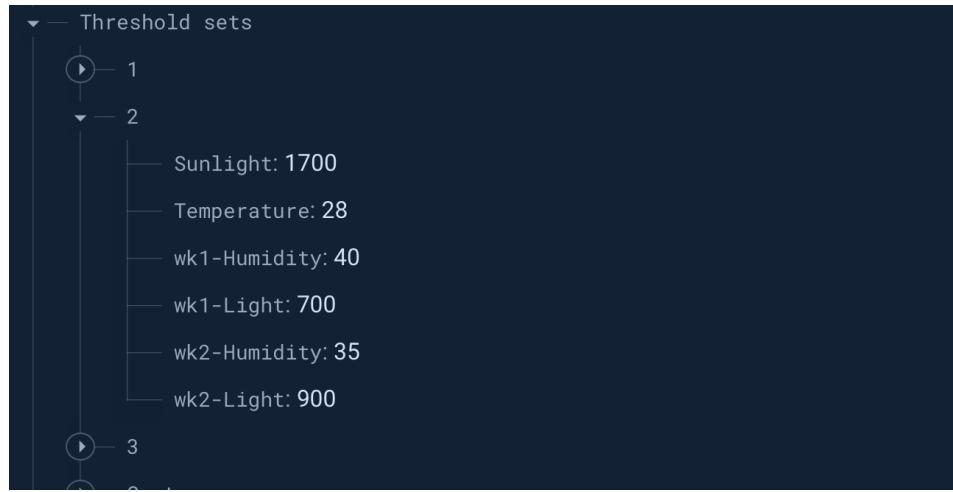
Rysunek 3.1: Struktura przechowywania pomiarów szklarni

Analogicznie wygląda zapisywanie zmierzonych wartości w segmencie Workstations (rysunek 3.2). Pomiary wilgotności powietrza oraz natężenia światła są przypisywane do poszczególnych stanowisk, w czytlenym formacie przedstawiającym odpowiadającą im datę i czas.



Rysunek 3.2: Struktura przechowywania pomiarów stanowisk

W segmencie „Threshold sets” bazy danych zapisane są z góry określone zestawy progów (thresholds), które służą do monitorowania i regulacji kluczowych parametrów środowiskowych w szklarni. Struktura ta, przedstawiona na rysunku 3.3, zawiera wartości progowe dla szablonu numer 2, które są stosowane do automatycznego sterowania warunkami w poszczególnych stanowiskach. Dla każdego stanowiska ustalone osobne wartości dla wilgotności i natężenia światła, co pozwala na zindywidualizowane zarządzanie każdą sekcją szklarni.



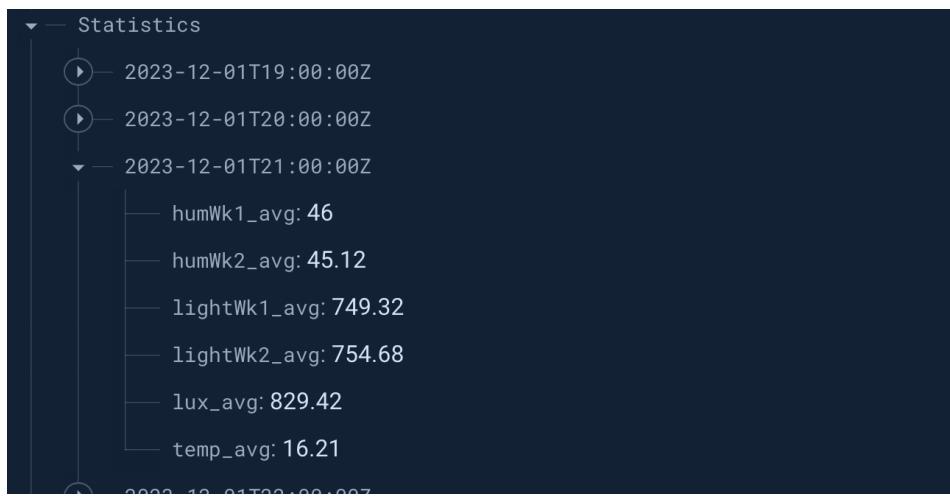
Rysunek 3.3: Przechowywanie szablonów nastaw wartości progowych

Podsegment „In Use” widoczny na rysunku 3.4 w bazie danych odpowiada za przechowywanie informacji o aktywnych szablonach nastaw. Zmienna „Set” określa, który z dostępnych szablonów jest aktualnie wykorzystywany. Użytkownik ma możliwość zmiany tego ustawienia wyłącznie za pośrednictwem aplikacji mobilnej, przy czym wybór ograniczony jest do czterech predefiniowanych opcji. Pierwsze trzy szablony mają charakter stały i nie podlegają modyfikacji – są to ustawienia wstępnie zdefiniowane, dostosowane do typowych scenariuszy działania szklarni. Czwarty szablon, oznaczony jako tryb ręczny, oferuje użytkownikowi elastyczność w konfiguracji własnych, niestandardowych wartości progowych, umożliwiając precyzyjne dopasowanie systemu do indywidualnych potrzeb.



Rysunek 3.4: Przechowywanie aktualnie używany szablon

Sekcja statystyk przedstawiona na rysunku 3.5 gromadzi dane, rejestrowane w wcześniej wspomnianym formacie czasowym. Generowanie wykresów z długiego okresu połącza za sobą konieczność pobierania znaczących ilości rekordów do aplikacji i ich lokalnego przetwarzania. Aby zoptymalizować ten proces, segment ten gromadzi średnie wartości godzinne poszczególnych pomiarów. Pobieranie mniejszej ilości danych może poprawić wydajność aplikacji mobilnej poprzez zmniejszenie czasu oczekiwania na odpowiedzi z bazy danych i redukcję zużycia danych, co jest szczególnie korzystne w środowiskach z ograniczoną przepustowością sieci.



Rysunek 3.5: Przechowywanie danych do wyświetlania statystyk

3.2. Komunikacja mikrokomputera z bazą danych

Na rysunku 3.6 przedstawiono procedurę inicjalizacji połączenia z bazą danych Firebase Realtime Database w języku Python. Proces ten rozpoczyna się od importowania niezbędnych poświadczeń za pomocą modułu credentials z pakietu firebase_admin. Poświadczenia te są wczytywane z pliku JSON, który zawiera klucz prywatny oraz inne wymagane dane uwierzytelniające, umożliwiające bezpieczne połączenie z usługą Firebase [5].

Następnie, wykorzystując metodę initialize_app z pakietu firebase_admin, aplikacja jest konfigurowana do komunikacji z bazą danych. Funkcja ta przyjmuje jako argumenty wcześniej wczytane poświadczenia oraz słownik konfiguracyjny określający URL bazy danych. Wskazany URL jest unikatowym endpointem, który odnosi się do instancji Realtime Database, lokalizowanej w regionie europejskim (europe-west1).

```

  i2c_communication.py
  main.py
  read_sensors.py
  realtimedb-7c282.firebaseio-adminsdk-sgbqc-5d77473285.json

39
40 # Ścieżka do pliku z kluczem autoryzacyjnym Firebase
41 cred = credentials.Certificate('realtimedb-7c282.firebaseio-adminsdk-sgbqc-5d77473285.json')
42
43 # Inicjalizacja aplikacji Firebase z podanymi credencjałami i bazą danych
44 firebase_admin.initialize_app(cred, {
45     'databaseURL': 'https://realtimedb-7c282-default-rtdb.europe-west1.firebaseio-adminsdk-sgbqc-5d77473285.firebaseio-app'
46 })
  
```

Rysunek 3.6: Inicjalizacja połączenia z bazą danych

Procedura ta jest standardową praktyką w przypadku aplikacji wymagających interakcji z Firebase Realtime Database, pozwalając na realizację operacji takich jak odczyt, zapis, aktualizacja czy monitorowanie danych w czasie rzeczywistym. Warto podkreślić, że plik z poświadczaniami musi być chroniony przed nieautoryzowanym dostępem, ponieważ zawiera wrażliwe dane, które mogą pozwolić na pełny dostęp do bazy danych projektu.

3.3. Integracja czujników i odczytu danych w Pythonie

Dbając o czytelność i modularność kodu podzieliłem program obsługujący model demonstracyjny na trzy pliki:

- i2c_communication.py - odpowiada za komunikacje z czujnikami BH1750
- read_sensors.py - odczytuje pozostałe czujniki DS18B20 oraz DHT11 i wywołuje i2c_communication.main()
- main.py - główny program odpowiadający za komunikacje z bazą danych oraz przetwarzanie pomiarów zwróconych przez wywołanie read_sensors.main()

3.3.1. i2c_communication.py

Program rozpoczyna się od inicjalizacji magistral I2C przy pomocy metody z zaimportowanej biblioteki smbus. Następnym krokiem jest utworzenie zmiennych pomocniczych przechowujących możliwe adresy czujnika BH1750.

```
1 import smbus
2
3 # Initialize I2C bus
4 bus = smbus.SMBus(1) # Magistrala 1
5 bus0 = smbus.SMBus(0) # Magistrala 0
6
7 # BH1750 addresses
8 DEVICE = 0x23 # Default
9 DEVICE2 = 0x5c # Alternative
10
```

Rysunek 3.7: Zawartość skryptu i2c_communication.py przedstawiająca inicjalizacje magistral

Na przedstawionym rysunku 3.8 zaprezentowano komunikację z czujnikami BH1750. Zdefiniowane na początku fragmentu kodu stałe są komendami służącymi do komunikacji z czujnikiem, zaczerpnięte z dokumentacji technicznej. Wykorzystywana w programie ONE_TIME_HIGH_RES_MODE_1 ustawia typowy czas pomiaru na 120 ms i dokładność do 1lx. Następnym elementem skryptu jest funkcja pomocnicza „convertToNumber()”, która przekształca surowe dane dwubajtowe z czujnika na wartość liczbową. Funkcje „readLight()” oraz „readLightBus0()” korzystają z tej metody konwersji, na danych pobranych za pomocą metody „read_i2c_block_data()” z pakietu smbus. Funkcje te odbierają dane z odpowiednich magistral I2C, wykorzystując przekazane im adresy czujników oraz komendy.

W funkcji głównej wywoływanie są omówione funkcje odczytywania natężenia światła dla trzech czujników. Otrzymane wartości są zwarcane w ustalonej kolejności: natężenie światła stanowiska 1, natężenie światła stanowiska 2, natężenie światła słonecznego.

```
11 # Command constants for BH1750 from datasheet
12 POWER_DOWN = 0x00
13 POWER_ON = 0x01
14 RESET = 0x07
15 CONTINUOUS_LOW_RES_MODE = 0x13
16 CONTINUOUS_HIGH_RES_MODE_1 = 0x10
17 CONTINUOUS_HIGH_RES_MODE_2 = 0x11
18 ONE_TIME_HIGH_RES_MODE_1 = 0x20
19 ONE_TIME_HIGH_RES_MODE_2 = 0x21
20 ONE_TIME_LOW_RES_MODE = 0x23
21
22 def convertToNumber(data):
23     return (data[1] + (256 * data[0])) / 1.2
24
25 def readLight(addr):
26     data = bus.read_i2c_block_data(addr, ONE_TIME_HIGH_RES_MODE_1)
27     return convertToNumber(data)
28
29 def readLightBus0(addr):
30     data = bus0.read_i2c_block_data(addr, ONE_TIME_HIGH_RES_MODE_1)
31     return convertToNumber(data)
32
33 def main():
34     lightLevel = readLight(DEVICE) # Read from first sensor in bus 1
35     lightLevel2 = readLight(DEVICE2) # Read from second sensor in bus 1
36     lightLevelG = readLightBus0(DEVICE) # Read from first sensor in bus 0
37     return lightLevel, lightLevel2, lightLevelG
```

Rysunek 3.8: Zawartość skryptu i2c_communication.py

3.3.2. read_sensors.py

Na początku programu przedstawionego na rysunku 3.9 importowane są biblioteki Adafruit_DHT do obsługi czujnika DHT11, w1thermsensor umożliwiający komunikację z czujnikiem DS18B20 za pomocą protokołu 1-wire oraz wcześniej omówiony skrypt i2c_communication odpowiadający za odczyt natężeń światła z czujników BH1750. Następnie inicializowane są zmienne odpowiadające czujnikom oraz w przypadku DHT11 piny, do których są podłączone.

```
import Adafruit_DHT
import w1thermsensor
import i2c_communication

## Init:
sensor = Adafruit_DHT.DHT11
sensor2 = Adafruit_DHT.DHT11

# DHT pins
sensor_pin = 17
sensor2_pin = 27

# DS18B20
sensorDS = w1thermsensor.W1ThermSensor()
```

Rysunek 3.9: Zawartość skryptu read_sensors.py przedstawiajaca inicializacje

Funkcja „main()” przedstawiona na rysunku 3.10 w pierwszej kolejności odczytuje aktualną temperaturę z czujnika DS18B20 za pomocą metody „get_temperature()” z pakietu w1thermsensor. Następnie wywołuje omówioną wcześniej główną funkcję skryptu i2c_communication, przypisując zwracane przez nią wartości do zmiennych lokalnych. Odczyt z czujników DHT11 odbywa się za pomocą metody „read_retry”, przyjmującej jako argumenty zainicjowany wcześniej sensor oraz odpowiadający mu pin GPIO. Funkcja próbuje do 15 razy odczytać pomiar z czujnika i, jeśli zakończy się niepowodzeniem, zostanie zwrócona wartość Null. Następnie odczytane z czujników dane zwracane są w określonej kolejności: temperatura otoczenia, natężenie światła stanowiska 1, natężenie światła stanowiska 2, natężenie światła słonecznego, wilgotność powietrza stanowiska 1, wilgotność powietrza stanowiska 2.

```
def main():
    # DS18B20
    temp = sensorDS.get_temperature()

    # i2c
    lightLevel, lightLevel2, lightLevelG = i2c_communication.main()
    print("Greenhouse: Light {0:.1f} lx, Greenhouse temperature = {1:0.1f}°C".format(lightLevelG, temp))

    # DHT
    humidity_wk1, temperature_wk1 = Adafruit_DHT.read_retry(sensor, sensor_pin)
    humidity_wk2, temperature_wk2 = Adafruit_DHT.read_retry(sensor2, sensor2_pin)
    if humidity_wk1 is not None and temperature_wk1 is not None:
        print("Workstation 1: Light {0:.1f} lx, Humidity = {1:0.1f}%".format(lightLevel, humidity_wk1))
    else:
        print("Sensor DHT11 failure. Check wiring.")
    if humidity_wk2 is not None and temperature_wk2 is not None:
        print("Workstation 2: Light {0:.1f} lx, Humidity = {1:0.1f}%".format(lightLevel2, humidity_wk2))
    else:
        print("Sensor DHT11 failure. Check wiring.")

    return temp, lightLevel, lightLevel2, lightLevelG, humidity_wk1, humidity_wk2
```

Rysunek 3.10: Zawartość skryptu read_sensors.py przedstawiająca funkcję main()

3.3.3. main.py

Rysunki 3.11 i 3.12 przedstawiają inicjalizację zmiennych, która jest przeprowadzana na wstępie głównego programu. Początkowo, określone we wcześniejszej fazie projektu piny GPIO są przypisywane do odpowiadających im zmiennych. Dalej ustalane są zmienne typu boolean, które reprezentują stan akcji, takich jak aktywacja wentylatora czy diod LED.

```
## Inicjalizacja
# Zmienne pinow GPIO
fan = 18
yellow_led_wk1 = 19
yellow_led_wk2 = 13
blue_led_wk1 = 6
blue_led_wk2 = 5
red_led = 26
# Zmienne przechowujace stan logiczny poszczegolnych akcji
FAN = False
YELLOW_LED_WK1 = False
YELLOW_LED_WK2 = False
BLUE_LED_WK1 = False
BLUE_LED_WK2 = False
RED_LED = False
```

Rysunek 3.11: Zawartość skryptu main.py przedstawiająca inicjalizację pinów GPIO

Kolejny etap to wykorzystanie zmiennych pinów w konfiguracji GPIO, ustawiając je jako wyjścia. W dalszym ciągu, inicjowane są zmienne związane z aktualnie używanym szablonem nastaw, jego wartościami, a także te używane do obliczania średnich z pomiarów.

```
#Inicializacja pinow GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(yellow_led_wk1, GPIO.OUT)
GPIO.setup(yellow_led_wk2, GPIO.OUT)
GPIO.setup(blue_led_wk1, GPIO.OUT)
GPIO.setup(blue_led_wk2, GPIO.OUT)
GPIO.setup(red_led, GPIO.OUT)
GPIO.setup(fan, GPIO.OUT)

Threshold_set = 99 # Aktualnie wybrany szablon
Thresholds = {99, 99, 99, 99, 99, 99} # Wartosci aktualnie wybranego szablonu
ref_thresh = None # Inicializacja pustej referencji

# Zmienne wyliczania wartosci srednich
temp_avg = 0
lux_avg = 0
lightWk1_avg = 0
lightWk2_avg = 0
humWk1_avg = 0
humWk2_avg = 0
# Licznik zsumowanych pomiarow
counter = 0
```

Rysunek 3.12: Zawartość skryptu main.py przedstawiająca inicializację zmiennych

Ostatnim krokiem inicializacji jest zdefiniowanie referencji zawierających scieżki do poszczególnych sektorów w bazie danych (rysunek 3.13).

```
# Referencje do bazy danych
ref_main = db.reference('Szklarnia/Measurements')
ref_work1 = db.reference('Szklarnia/Workstations/Workstation 1/Measurements')
ref_work2 = db.reference('Szklarnia/Workstations/Workstation 2/Measurements')
ref_stats = db.reference('Szklarnia/Statistics')
ref_sets = db.reference('Szklarnia/Threshold sets/In use')
```

Rysunek 3.13: Zawartość skryptu main.py przedstawiająca inicializację referencji do sektorów bazy danych

Funkcja „init_thresholds()” przedstawiona na rysunku 3.14 działa na zmiennych globalnych przechowujących informację o obecnie wybranym szablonie i jego wartościach oraz referencji ref_thresh, która w momencie pierwszego wywołania programu nie

ma przypisanej ścieżki do aktualnie wybranego szablonu. Jest ona do niego przypisywana po sprawdzeniu przez funkcję jaki szablon jest aktualnie w użyciu. Następnie za pośrednictwem uzyskanej ścieżki, aktualne wartości wybranego szablonu są pobierane i przybrane do zmiennej globalnej Thresholds.

```
def init_thresholds():
    global Threshold_set
    global Thresholds
    global ref_thresh

    result = ref_sets.get()
    if result:
        set_value = result.get('Set')
        if set_value is not None:
            Threshold_set = set_value
            if (Threshold_set == 4): ref_thresh = db.reference(f'Szklarnia/Threshold sets/Custom')
            else: ref_thresh = db.reference(f'Szklarnia/Threshold sets/{Threshold_set}')

            thresholds_values = ref_thresh.get()
            if thresholds_values:
                # Przypisanie wartości do globalnej zmiennej Thresholds w określonej kolejności
                Thresholds = [
                    thresholds_values.get('Temperature'),
                    thresholds_values.get('wk1-Light'),
                    thresholds_values.get('wk2-Light'),
                    thresholds_values.get('Sunlight'),
                    thresholds_values.get('wk1-Humidity'),
                    thresholds_values.get('wk2-Humidity'),
                ]
            else:
                print("Nie znaleziono danych dla wybranego zestawu.")
        else:
            print("Klucz 'Set' nie istnieje.")
    else:
        print("Referencja nie zwróciła danych.")
```

Rysunek 3.14: Zawartość skryptu main.py przedstawiająca funkcję init_thresholds()

Rysunek 3.15 przedstawia funkcje odpowiadające za nasłuchiwanie modyfikacji wybranych zmiennych w bazie danych. Funkcja „listen_for_threshold_set_changes()” odpowiada za obserwowanie zmian w ścieżce ref_sets. Jeżeli wystąpi tam zmiana, będa zieć to oznaczało, że zmienił się wybrany szablon. Poskutkuje to wywołaniem funkcji „init_thresholds()”, w celu zaktualizowania zmiennych globalnych odpowiadających wybranemu szablonowi oraz jego wartości.

```
# Nasłuchiwanie aktualnego setu
def listen_for_threshold_set_changes():
    def on_change(event):
        init_thresholds()

        ref_sets.listen(on_change) # Nasłuchiwanie zmiany parametru w bazie danych

# Nasłuchiwanie wartości progowych
def listen_for_threshold_changes():
    def on_change(event):
        global Threshold_set
        if event.data:
            init_thresholds()
        ref_thresh.listen(on_change) # Nasłuchiwanie zmiany parametrow w bazie danych
```

Rysunek 3.15: Zawartość skryptu main.py przedstawiająca funkcje nasłuchujące

Funkcja „is_full_hour_timestamp()” przedstawiona na poniższym rysunku 3.16 weryfikuje, czy przekazany jako argument znacznik czasowy odpowiada pełnej godzinie. Używana jest ona w instrukcji warunkowej, która determinuje moment wysłania do bazy danych średniej arytmetycznej z pomiarów z ostatniej godziny.

```
def is_full_hour_timestamp(timestamp):
    # Usunięcie 'Z' z końca timestamp, jeśli istnieje
    timestamp = timestamp.replace('Z', '')
    dt_object = datetime.fromisoformat(timestamp) # Konwersja timestamp na obiekt datetime
    return dt_object.minute == 0 and dt_object.second == 0 # Sprawdzenie, czy minuty i sekundy są równe 0
```

Rysunek 3.16: Zawartość skryptu main.py przedstawiająca funkcję porównującą czas

Funkcja „checkThresholds()” przyjmuje w parametrach aktualne wartości pomiarów, które następnie są porównywane z wartościami progowymi przechowywanymi w zmiennej globalnej Thresholds. Ze względu na potrzebę modyfikacji wartości zmiennych logicznych, które przechowują stan poszczególnych akcji, inicjowane są one za pomocą słowa kluczowego global, co pozwala na ich aktualizację na poziomie globalnym.

W pierwszej kolejności sprawdzane jest czy aktualna temperatura jest większa od ustawionej wartości progowej - jeśli tak, to załącza się wentylator. Gdy warunek nie jest spełniony, to silniczek się wyłącza. Wartym zauważenia jest rozszerzenie algorytmu o dodatkowy warunek sprawdzający czy akcja jest już włączona lub wyłączona, w efekcie tego nie wysyła się sygnałów zmieniających akcję przy każdym wywołaniu funkcji, tylko przy zmianie wartości logicznej warunku nadziednego. Robi to podwaliny pod przyszłą rozbudowę systemu o możliwość ręcznego włączenia i wyłączenia każdej z akcji. Na-

stępnie pomiar natężenia światła stanowiska 1 porównywany jest z odpowiadającą mu wartością minimalną. Jeśli jest od niej mniejszy zapalana jest żółta dioda LED przy stanowisku 1. Gdy warunek przestaje być spełniony dioda gaśnie.

```
def checkThresholds(temp, lightLevel1, lightLevel2, lightLevelG, humidity_wk1, humidity_wk2):
    global FAN
    global YELLOW_LED_WK1
    global YELLOW_LED_WK2
    global BLUE_LED_WK1
    global BLUE_LED_WK2
    global RED_LED
    if(temp > Thresholds[0]):
        if(FAN == False):
            print("Turn on fan")
            GPIO.output(fan, GPIO.HIGH)
            FAN = True
        else:
            if(FAN == True):
                print("Turn off fan")
                GPIO.output(fan, GPIO.LOW)
            FAN = False

        if(lightLevel1 < Thresholds[1]):
            if(YELLOW_LED_WK1 == False):
                print("Turn on wk1 yellow light")
                GPIO.output(yellow_led_wk1, GPIO.HIGH)
                YELLOW_LED_WK1 = True
            else:
                if(YELLOW_LED_WK1 == True):
                    print("Turn off wk1 yellow light")
                    GPIO.output(yellow_led_wk1, GPIO.LOW)
                YELLOW_LED_WK1 = False
```

Rysunek 3.17: Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzającej logiką modelu demonstracyjnego

Warunek dla natężenia światła na stanowisku 2 przedstawiony na rysunku 3.18 jest analogiczny do algorytmu stanowiska 1. Kolejnym sprawdzanym parametrem jest natężenie światła słonecznego. Jeśli przekroczy ono zadany maksymalny próg, włącza się czerwona dioda LED, symbolizująca opuszczenie rolet. Zmiana stanu logicznego tego warunku powoduje wygaszenie diody.

```
if(lightLevel2 < Thresholds[2]):  
    if(YELLOW_LED_WK2 == False):  
        print("Turn on wk2 yellow light")  
        GPIO.output(yellow_led_wk2, GPIO.HIGH)  
        YELLOW_LED_WK2 = True  
    else:  
        if(YELLOW_LED_WK2 == True):  
            print("Turn off wk2 yellow light")  
            GPIO.output(yellow_led_wk2, GPIO.LOW)  
        YELLOW_LED_WK2 = False  
  
if(lightLevelG > Thresholds[3]):  
    if(RED_LED == False):  
        print("Turn on red light")  
        GPIO.output(red_led, GPIO.HIGH)  
        RED_LED = True  
    else:  
        if(RED_LED == True):  
            print("Turn off red light")  
            GPIO.output(red_led, GPIO.LOW)  
        RED_LED = False
```

Rysunek 3.18: Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzającej logiką modelu demonstracyjnego

Dwa ostatnie warunki dotyczą wilgotności powietrza na poszczególnych stanowiskach. Jeśli wilgotność spada poniżej ustalonego progu, aktywowana jest niebieska dioda LED, która jest przypisana do danego stanowiska. Gdy wartość wilgotności przekroczy ten próg, dioda zostaje wyłączona.

```
if(humidity_wk1 < Thresholds[4]):  
    if(BLUE_LED_WK1 == False):  
        print("Turn on wk1 blue light")  
        GPIO.output(blue_led_wk1, GPIO.HIGH)  
        BLUE_LED_WK1 = True  
    else:  
        if(BLUE_LED_WK1 == True):  
            print("Turn off wk1 blue light")  
            GPIO.output(blue_led_wk1, GPIO.LOW)  
        BLUE_LED_WK1 = False  
  
if(humidity_wk2 < Thresholds[5]):  
    if(BLUE_LED_WK2 == False):  
        print("Turn on wk2 blue light")  
        GPIO.output(blue_led_wk2, GPIO.HIGH)  
        BLUE_LED_WK2 = True  
    else:  
        if(BLUE_LED_WK2 == True):  
            print("Turn off wk2 blue light")  
            GPIO.output(blue_led_wk2, GPIO.LOW)  
        BLUE_LED_WK2 = False
```

Rysunek 3.19: Zawartość skryptu main.py przedstawiająca fragment funkcji zarządzającej logiką modelu demonstracyjnego

Funkcja „sendMeasurments()” jako argumenty przyjmuje zmierzone wartości oraz znacznik czasowy, w którym zostały wykonane. Dzielone są one na ogólne oraz przypisane do stanowisk, a następnie zapisywane do odpowiednich segmentów w bazie danych, z nazwą odpowiadającą przekazanemu do funkcji znacznikowi czasowemu.

```
def sendMeasurments(temp, lightLevel1, lightLevel2, lightLevelG, humidity_wk1, humidity_wk2, timestamp):

    ref_main.child(timestamp).set({
        'temp': temp,
        'lux': lightLevelG
    })

    ref_work1.child(timestamp).set({
        'Humidity': humidity_wk1,
        'Light': lightLevel1
    })

    ref_work2.child(timestamp).set({
        'Humidity': humidity_wk2,
        'Light': lightLevel2
    })
    print("Saved")
```

Rysunek 3.20: Zawartość skryptu main.py przedstawiająca funkcję przesyłającą pomiary do bazy danych

Przedstawiona na rysunku 3.21 funkcja „sendAvgMeasurements()” jako argument przyjmuje jedynie znacznik czasowy. Zmienne z dopiskiem „avg”, do momentu wywołania funkcji, są sumą pomiarów wysyłanych do bazy danych. Przy każdym sumowaniu zmienna counter jest inkrementowana, co umożliwia wyliczenie wartości średniej. Wynik jest uśredniany i następnie zapisywany w bazie danych, uwzględniając odpowiedni znacznik czasowy. Funkcja kończy się wyzerowaniem zmiennych globalnych, by umożliwić poprawne zliczanie dla następnej godziny.

```
def sendAvgMeasurements(timestamp):
    global temp_avg, lux_avg, lightWk1_avg, lightWk2_avg, humWk1_avg, humWk2_avg, counter
    if counter != 0:
        temp_avg = round(temp_avg / counter, 2)
        lux_avg = round(lux_avg / counter, 2)
        lightWk1_avg = round(lightWk1_avg / counter, 2)
        lightWk2_avg = round(lightWk2_avg / counter, 2)
        humWk1_avg = round(humWk1_avg / counter, 2)
        humWk2_avg = round(humWk2_avg / counter, 2)

    ref_stats.child(timestamp).set({
        'temp_avg': temp_avg,
        'lux_avg': lux_avg,
        'lightWk1_avg': lightWk1_avg ,
        'lightWk2_avg': lightWk2_avg ,
        'humWk1_avg': humWk1_avg,
        'humWk2_avg': humWk2_avg
    })
    print("Avg measurements saved")
    temp_avg = 0
    lux_avg = 0
    lightWk1_avg = 0
    lightWk2_avg = 0
    humWk1_avg = 0
    humWk2_avg = 0
    counter = 0
    print("Avg var cleared")
```

Rysunek 3.21: Zawartość skryptu main.py przedstawiająca funkcję przesyłającą średnie pomiarów do bazy danych

Rysunek 3.22 przedstawia wywołanie omówionych funkcji inicjalizujących działanie głównej części programu oraz nieskończoną pętlę. Analizując jej cykl, w pierwszej kolejności pobierana jest aktualna data.

Następnie program sprawdza, czy zaokrąglone sekundy w otrzymanym formacie czasu są podzielne przez liczbę 5. W efekcie warunek będzie spełniony co równy pięciosekundowy odstęp. Gdy to nastąpi, format czasu zostaje dostosowany do przyjętego, aby następnie wypisać go w konsoli. Kolejnym krokiem jest wywołanie funkcji „read_sensors.main()”, zwracającej odczytane z czujników wartości, które następnie są zaokrąglane do pierwszego miejsca po przecinku. Otrzymane wartości przekazywane są do funkcji zarządzającej logiką systemu. Po tej sekwencji wypisywane są wartości programowe przechowywane obecnie w zmiennej globalnej.

Jeśli zaokrąglone sekundy w formacie czasu są dodatkowo podzielne przez 10, pomiary wysyłane są do bazy danych. Następnie pomiary są sumowane do zmiennych służących do zliczania średniej oraz inkrementuje się licznik counter. Ostatnim krokiem

jest sprawdzenie, czy aktualny czas jest pełną godziną i, jeśli tak, zostaje wywołana funkcja „sendAvgMeasurements()”.

```
init_thresholds()
listen_for_threshold_set_changes()
listen_for_threshold_changes()

while True:
    now = datetime.now()
    rounded_second = (now + timedelta(milliseconds=500)).replace(microsecond=0)

    if rounded_second.second % 5 == 0:
        timestamp = datetime.isoformat(rounded_second) + 'Z'
        print("\n")
        print(timestamp)
        sensor_values = read_sensors.main()
        sensor_values_rounded = tuple(round(value, 1) for value in sensor_values)
        checkThresholds(*sensor_values_rounded)
        print("Thresholds: {0:d}, {1:d}, {2:d}, {3:d} ,{4:d} ,{5:d}".format(Thresholds[0], Thresholds[1],
        Thresholds[2], Thresholds[3], Thresholds[4], Thresholds[5]))

    if rounded_second.second % 10 == 0:
        sendMeasurements(*sensor_values_rounded, timestamp)
        temp_avg += sensor_values[0]
        lux_avg += sensor_values[1]
        lightWk1_avg += sensor_values[2]
        lightWk2_avg += sensor_values[3]
        humWk1_avg += sensor_values[4]
        humWk2_avg += sensor_values[5]
        counter += 1
        if is_full_hour_timestamp(timestamp):
            print(f"{timestamp} is full hour: {is_full_hour_timestamp(timestamp)}")
            sendAvgMeasurements(timestamp)
        else: print(f"{timestamp} is full hour: {is_full_hour_timestamp(timestamp)}")
    time.sleep(0.5)
```

Rysunek 3.22: Zawartość skryptu main.py przedstawiająca główną pętle programu

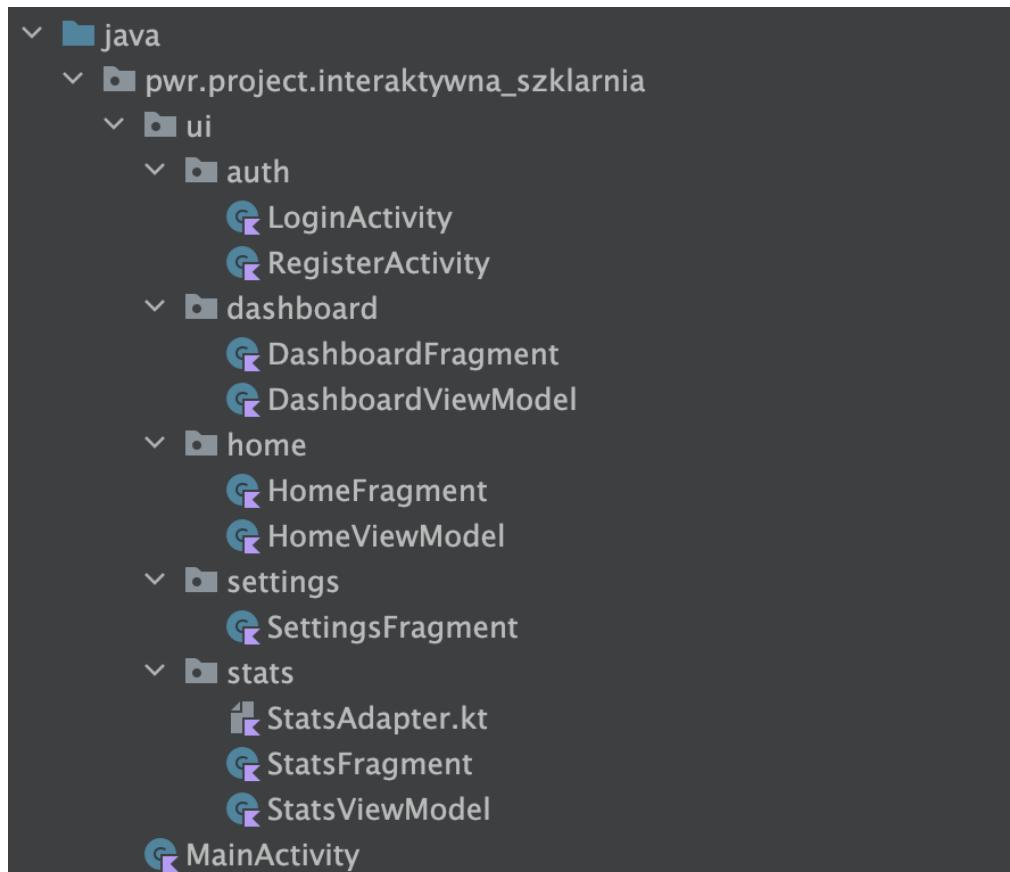
```
2023-12-09T19:34:20Z
Greenhouse: Light 1592.5 lx, Greenhouse temperature = 17.6°C
Workstation 1: Light 1911.7 lx, Humidity = 43.0%
Workstation 2: Light 1780.8 lx, Humidity = 43.0%
Thresholds: 32, 745, 650, 500 ,35 ,35
Saved
2023-12-09T19:34:20Z is full hour: False

2023-12-09T19:34:25Z
Greenhouse: Light 1592.5 lx, Greenhouse temperature = 17.6°C
Workstation 1: Light 1911.7 lx, Humidity = 43.0%
Workstation 2: Light 1780.8 lx, Humidity = 43.0%
Thresholds: 32, 745, 650, 500 ,35 ,35
```

Rysunek 3.23: Zrzut ekranu terminalu ukazujący wykonanie dwóch cykli głównej pętli programu

3.4. Struktura aplikacji mobilnej

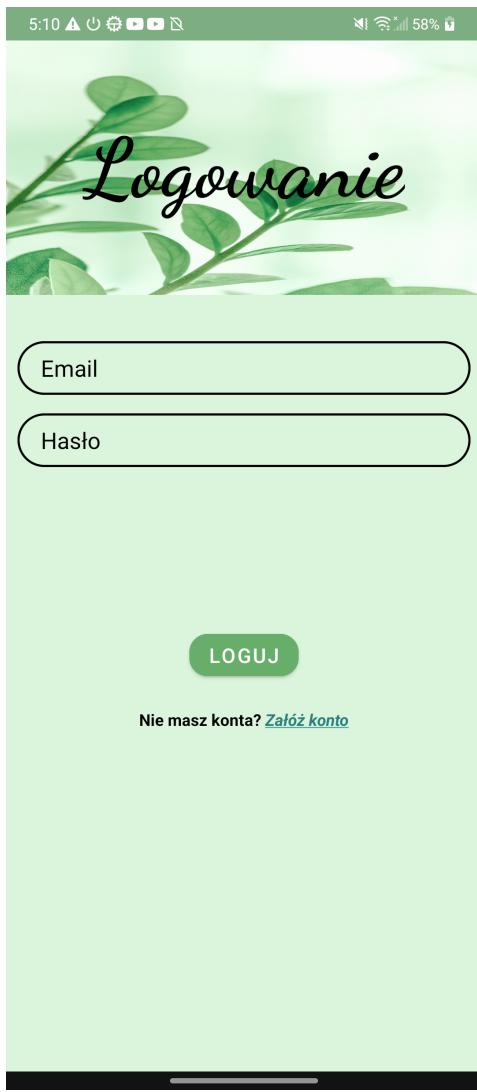
W celu implementacji aplikacji mobilnej w natywnym języku Kotlin, wykorzystane zostało środowisko Android Studio 11. Rysunek 3.24 przedstawia zaimplementowaną strukturę ViewModel-Fragment, omówioną w rozdziale 2.4.



Rysunek 3.24: Zrzut ekranu przedstawiający strukturę aplikacji mobilnej

3.5. Moduł autentykacji

Na moduł autentykacji składają się widoki logowania oraz rejestracji. Przy pierwszym uruchomieniu aplikacji użytkownik zobaczy ekran logowania, w którym znajduje się odniesienie do formularza zakładania konta. Utworzenie nowego profilu lub zalogowanie za pomocą już istniejącego poskutkuje przeniesiem do głównej części aplikacji.

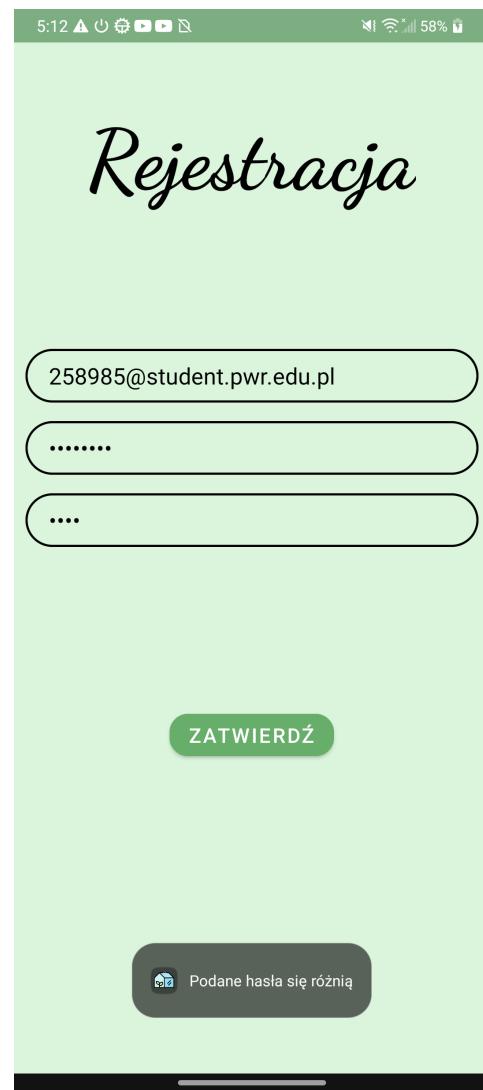


Rysunek 3.25: Ekran logowania



Rysunek 3.26: Ekran rejestracji

Ekran logowania 3.25 oraz ekran rejestracji 3.26 prezentują pola tekstowe typu EditText, przeznaczone do wprowadzania danych autentykacyjnych takich jak adres email i hasło. Naciśnięcie przycisków „loguj” oraz „zatwierdź” inicjuje proces weryfikacji poprawności wypełnienia tych pól. W przypadku wykrycia błędów, użytkownik otrzymuje stosowne komunikaty, których przykłady są przedstawione na rysunkach 3.27 i 3.28. Poprawnie wprowadzone dane autentykacyjne są następnie przekazywane do metod logowania i rejestracji zaimplementowanych w bibliotece FirebaseAuth.



Rysunek 3.27: Obsługa błędów logowania Rysunek 3.28: Obsługa błędów rejestracji

Na rysunkach 3.27 i 3.28 przedstawiono obsługę błędów, które mogą pojawić się podczas logowania i rejestracji. Te widoki demonstrują, w jaki sposób aplikacja komunikuje użytkownikowi problemy związane z niepoprawnym wypełnieniem formularzy.

Rysunek 3.29 przedstawia funkcje wywołaną poprzez naciśnięcie przycisku „loguj”. Gdy wywołanie metody „signInWithEmailAndPassword()” zakończy się powodzeniem, następuje przekazanie do funkcji updateUI instancji profilu użytkownika. W razie niepowodzenia funkcja updateUI uzyska wartość null.

```
fun login(view: View) {
    val email = binding.editTextEmail.text.toString()
    val password = binding.editTextPassword.text.toString()

    if (email.isNotEmpty() && password.isNotEmpty()) {
        auth.signInWithEmailAndPassword(email, password)
            .addOnCompleteListener(this) { task -
                if (task.isSuccessful) {
                    Log.d(TAG, msg: "signInWithEmailAndPassword:success")
                    val user = auth.currentUser
                    updateUI(user)
                } else {
                    Log.w(TAG, msg: "signInWithEmailAndPassword:failure", task.exception)
                    updateUI( currentUser: null )
                    Toast.makeText(baseContext, text: "Błędny login lub/i hasło.",
                        Toast.LENGTH_SHORT).show()
                }
            }
    } else { Toast.makeText( context: this, text: "Pole email lub/i hasło są puste",
        Toast.LENGTH_SHORT).show() }
}
```

Rysunek 3.29: Fragment skryptu LoginActivity.kt przedstawiający funkcje logowania

Funkcja updateUI przedstawiona na rysunku 3.30 sprawdza czy w parametrze został przekazany profil, a następnie zachodzi przekierowanie do głównej części programu. Wywołanie tej metody w funkcji onStart pozwala na pominięcie logowania po każdym ponownym włączeniu aplikacji oraz w przypadku założenia nowego profilu.

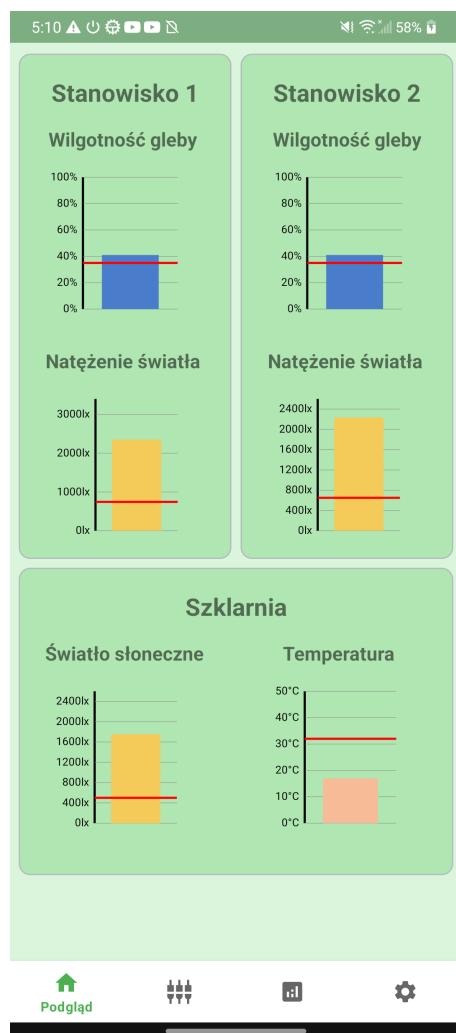
```
public override fun onStart() {
    super.onStart()
    val currentUser = auth.currentUser
    updateUI(currentUser)
} // Jeżeli użytkownik jest zalogowany, zostanie przeniesiony do głównej części programu

▲ mklimek585 *
private fun updateUI(currentUser: FirebaseUser?) {
    if (currentUser != null) {
        startActivity(Intent( packageContext: this, MainActivity::class.java))
        finish()
    }
}
```

Rysunek 3.30: Fragment skryptu LoginActivity.kt przedstawiający funkcje onStart oraz updateUI

3.6. Ekran podglądu

Ekran podglądu 3.31 stanowi pierwszy element głównej części programu i jest wyświetlany bezpośrednio po pomyślnym przejściu procesu autentykacji przez użytkownika. Prezentuje on kluczowe parametry za pomocą wykresów słupkowych. Ustalone przez użytkownika wartości progowe są wizualizowane poprzez czerwoną linię. Dla zwiększenia czytelności, zastosowano dynamiczną skalę osi Y, która dostosowuje się do wartości przedstawionych na wykresach dotyczących natężenia światła. Do prezentacji wykresów w aplikacji wykorzystuję bibliotekę MPAndroidChart, zgodnie z wymogami licencji Apache 2.0, na której jest ona oparta.



Rysunek 3.31: Ekran podglądu

3.7. Ekran sterowania

Ekran sterowania, zaprezentowany na rysunku 3.32, umożliwia użytkownikowi wybór jednego z predefiniowanych szablonów nastaw (szablony 1-3) albo ustawienie wartości progowych ręcznie. Szablony te są dostępne do wyboru za pomocą elementu typu „radio button”, umieszczonego nad przyciskiem „zatwierdź”. Wybór konkretnego szablonu powoduje wczytanie jego wartości z bazy danych. Podczas inicjalizacji ekranu sterowania, automatycznie wczytywany jest aktualnie używany zestaw nastaw. Gdy użytkownik rozpocznie edycję wartości w jednym z predefiniowanych szablonów, system automatycznie przełącza go w tryb ręczny, co zostanie zasygnalizowane poprzez zmianę stanu przycisku radiowego.

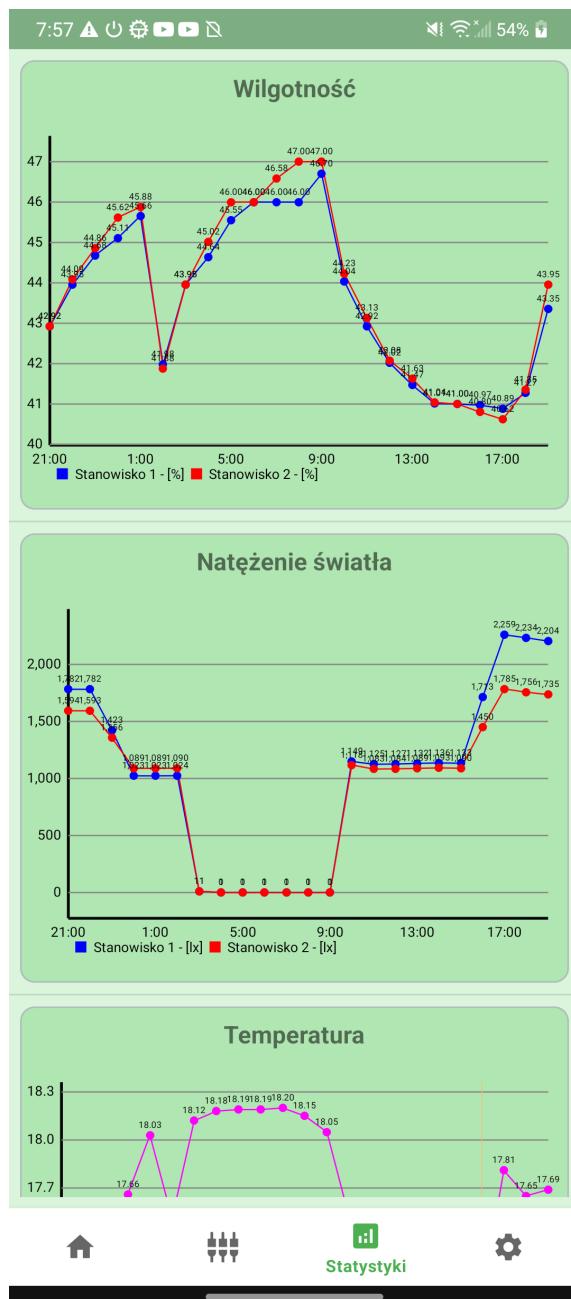


Rysunek 3.32: Ekran sterowania

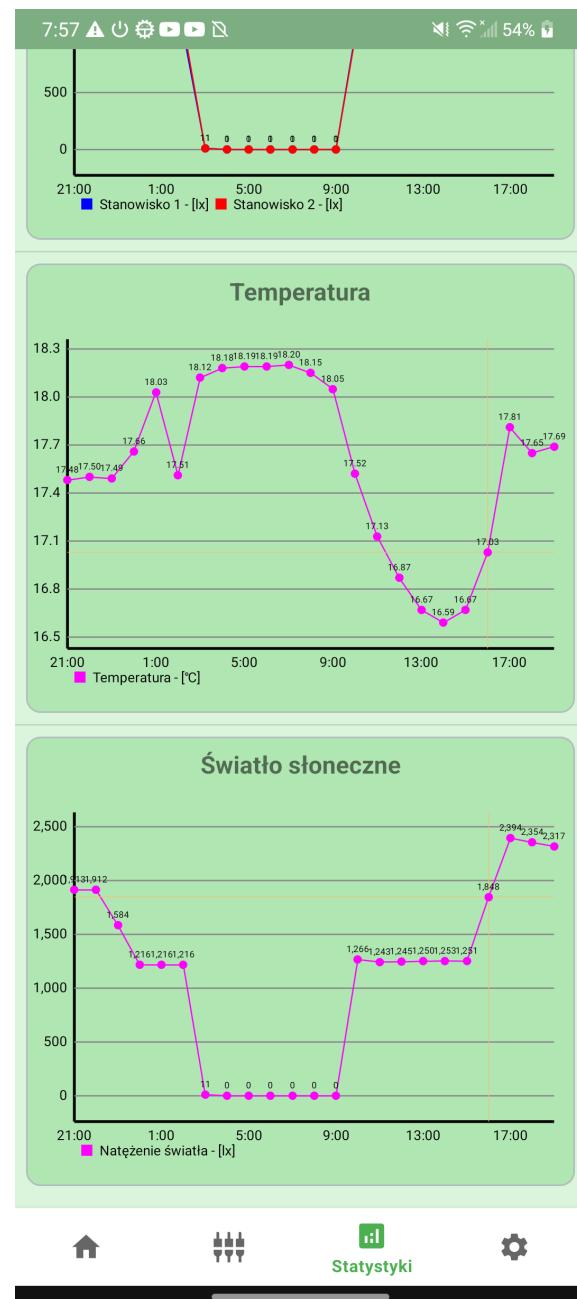
3.8. Ekran statystyk

Ekran statystyk prezentuje przewijaną listę z wykresami różnych parametrów. Domyslnie zakres danych na wykresach obejmuje ostatnie 24 godziny, jednak użytkownik ma możliwość zmiany tego zakresu na ostatni tydzień w ustawieniach aplikacji. Dane z obu stanowisk są prezentowane na jednym wykresie, co ułatwia ich porównanie (zobacz rysunek 3.33). Parametry niezależne, takie jak temperatura i natężenie światła słonecznego, są wyświetlane na osobnych wykresach (zobacz rysunek 3.34). Zaznaczone na charakterystyce punkty reprezentują średnią arytmetyczną pomiarów dla każdej godziny.

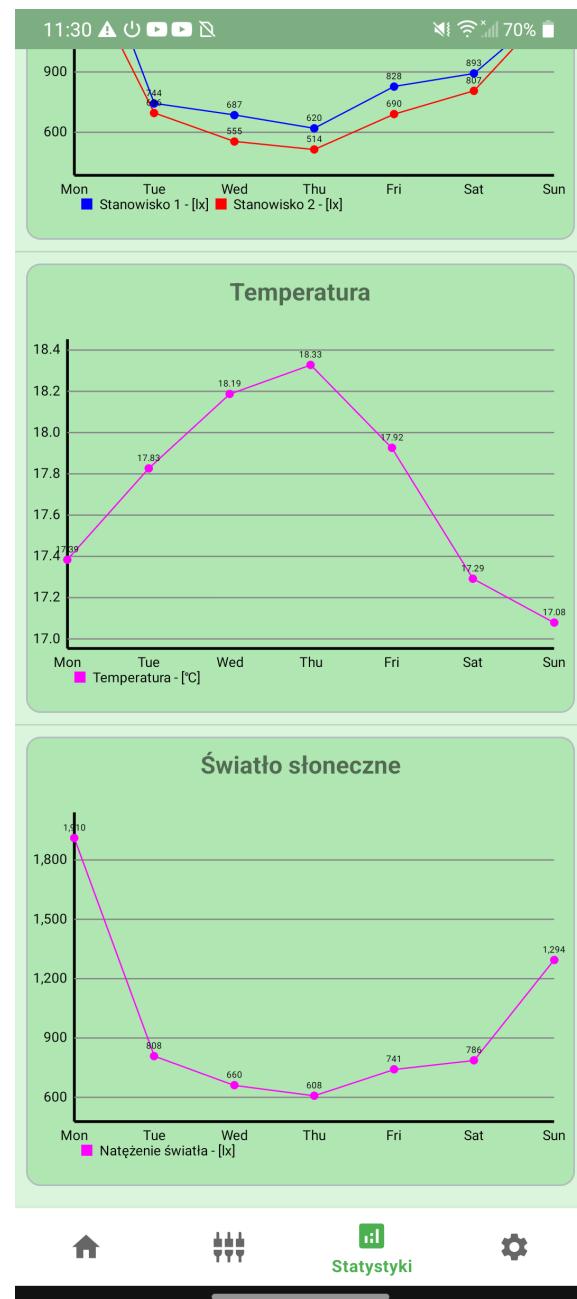
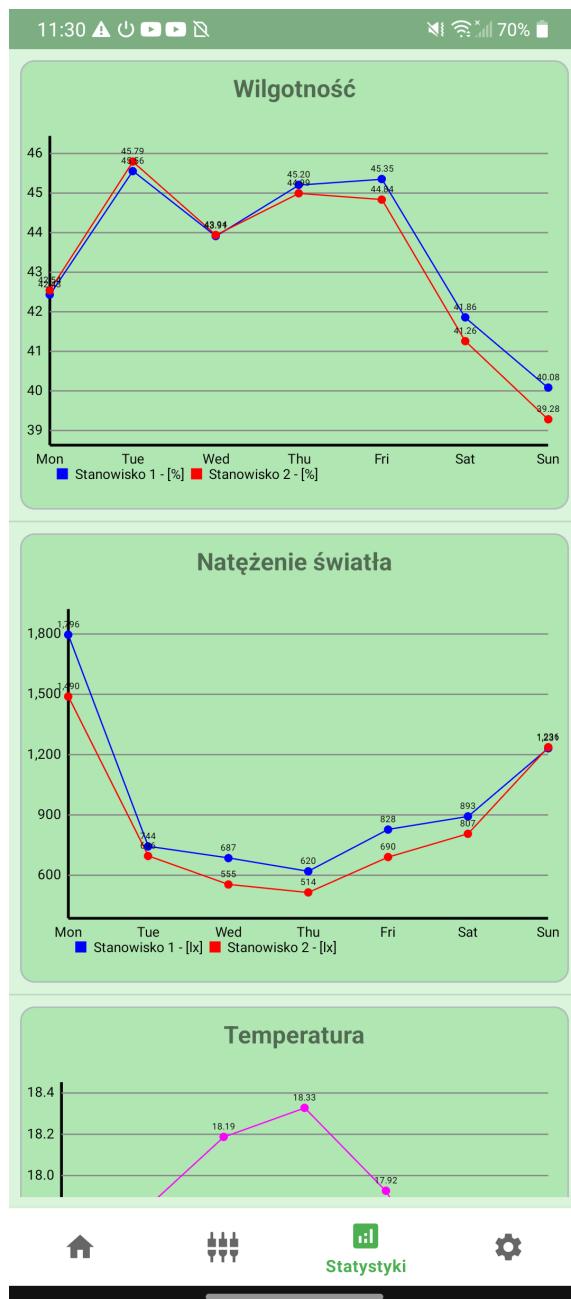
Rysunki 3.35 i 3.36 ilustrują sposób prezentacji danych, gdy zakres czasowy ustalony jest na tydzień. W tym ustawieniu, dane są filtrowane, a następnie są z nich wyliczane średnie dla każdego dnia.



Rysunek 3.33: Ekran statystyk przedstawiający wykres wilgotności oraz natężenia światła w układzie godzinowym



Rysunek 3.34: Ekran statystyk przedstawiający wykres temperatury oraz natężenia światła słonecznego w układzie godzinowym.

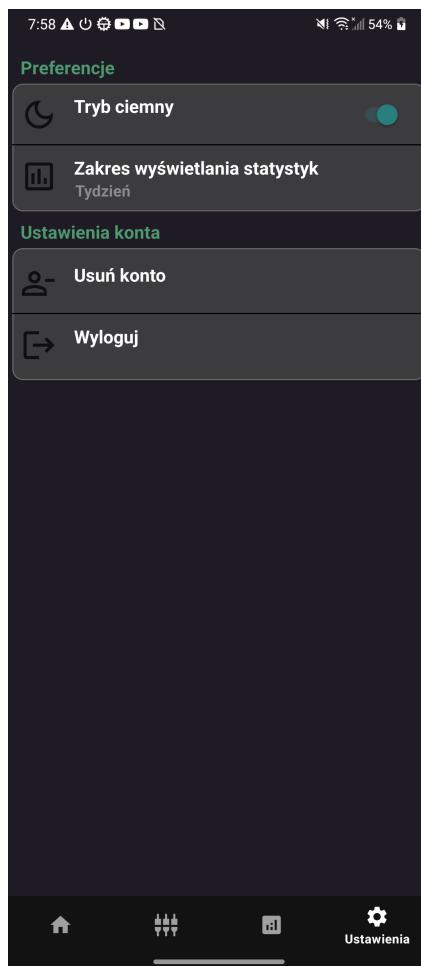


Rysunek 3.35: Ekran statystyk przedstawiający wykres wilgotności oraz natężenia światła w układzie tygodniowym

Rysunek 3.36: Ekran statystyk przedstawiający wykres temperatury oraz natężenia światła słonecznego w układzie tygodniowym

3.9. Ekran ustawień

W zakładce ustawień użytkownik ma możliwość wyboru opisanego w poprzedniej sekcji zakresu czasowego wyświetlania statystyk, za pomocą rozwijanego menu. Dostępna jest także opcja zmiany motywów aplikacji z jasnego tła na ciemny. Interfejs użytkownika został zaprojektowany tak, aby był czytelny i przejrzysty w obu wersjach kolorystycznych. Wybrane ustawienia, takie jak motyw oraz zakres czasowy wykresów, są zapisywane w SharedPreferences - mechanizmie pozwalającym zachować wartości między sesjami, co gwarantuje ich zachowanie po ponownym uruchomieniu aplikacji [1]. Pozostałymi funkcjami dostępnymi w ustawieniach są opcje umożliwiające usunięcie konta użytkownika oraz wylogowanie, po których użytkownik zostanie przekierowany do ekranu logowania.



Rysunek 3.37: Ekran ustawień z wybranym motywem ciemnym

3.10. Komunikacja aplikacji mobilnej z bazą danych

Analizując komunikację aplikacji z bazą danych można wyszczególnić następujące operacje:

Nasłuchiwanie zmian wartości

```
└─ mklimek585
    private fun setupCurrentThresholdListener() {
        val currentSetRef = databaseRef.child( pathString: "Szklarnia/Threshold sets/In use")
        val listener = object : ValueEventListener {
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                val set = dataSnapshot.child( path: "Set").getValue(Long::class.java)
                _currentSet.value = set?.toString()
                Log.i(TAG, msg: "Set currently in use: $set")
            }

            override fun onCancelled(databaseError: DatabaseError) {
                Log.w(TAG, msg: "loadPost:onCancelled", databaseError.toException())
            }
        }
        currentSetRef.addValueEventListener(listener)
        currentSetListener = listener
    }
```

Rysunek 3.38: Fragment kodu przedstawiający funkcję nasłuchującą zmiany wartości

Funkcja „setupCurrentThresholdListener()” inicjuje proces nasłuchiwanego zmian w wybranego szablonu wartości progowych. Tworzy ona instancję obiektu ValueEventListener, która reaguje na zmiany danych. Gdy nastąpi zmiana, wartość Set jest odczytywana z obiektu DataSnapshot i przypisywana do zmiennej lokalnej _currentSet.value. Dodatkowo, w logach rejestrowana jest informacja o aktualnie używanym szablonie.

W przypadku wystąpienia błędu podczas nasłuchiwanego, wywoływana jest metoda onCancelled, a błąd rejestrowany jest w logach.

Następnie utworzony obiekt listener jest przypisywany do nasłuchiwanego zmian w konkretnej ścieżce bazy danych, wskazującej na aktualnie używany szablon w sekcji „Szklarnia/Threshold sets/In use”. Na końcu, ten sam obiekt listener jest zapisywany do zmiennej currentSetListener dla późniejszego odwołania.

Zapisywanie danych

```
fun updateDatabase(setNumber: Int, updatedSet: HashMap<String, Int>, callback: (Boolean) -> Unit) {
    val setKey = if (setNumber == 4) "Custom" else setNumber.toString()
    val setRef = databaseRef.child(pathString: "Szklarnia/Threshold sets").child(setKey)

    setRef.updateChildren(updatedSet as Map<String, Any>
        .addOnSuccessListener { it: Void! // Sukces
            callback(true)
        }
        .addOnFailureListener { it: Exception // Niepowodzenie
            callback(false)
        }
    )
}
```

Rysunek 3.39: Fragment kodu przedstawiający funkcję zapisującą wartości

Funkcja „updateDatabase()”, będąca elementem klasy ViewModel, przyjmuje jako argumenty numer wybranego przez użytkownika szablonu oraz jego aktualne wartości, które są zapisane w strukturze HashMap. Następuje inicjalizacja referencji do określonego szablonu w bazie danych, po czym zostaje on nadpisany przekazaną strukturą za pomocą metody „updateChildren()”.

Pobieranie danych

```
fun loadSets(callback: DataCallback) {
    val setsRef = databaseRef.child(pathString: "Szklarnia/Threshold sets")
    val result = Array<Array<Int>?>(size: 4) { null }

    setsRef.get().addOnSuccessListener { dataSnapshot ->
        dataSnapshot.children.forEachIndexed { index, setSnapshot ->
            if (index < result.size) {
                val setValues = setSnapshot.children.mapNotNull { paramSnapshot ->
                    val value = paramSnapshot.value
                    when (value) {
                        is Long -> value.toInt() ^mapNotNull
                        is String -> value.toIntOrNull() ?: 0 ^mapNotNull
                        else -> 0 ^mapNotNull
                    }
                }.toTypedArray()

                result[index] = setValues
            }
        }
        callback.onDataLoaded(result.filterNotNull().toTypedArray())
    }.addOnFailureListener { databaseError ->
        Log.e(TAG, msg: "LoadSets error: $databaseError")
    }
}
```

Rysunek 3.40: Fragment kodu przedstawiający funkcję pobierającą wartości

Funkcja „loadSets” jest odpowiedzialna za jednorazowe pobranie podsegmentów określonej ścieżki, co osiąga poprzez wywołanie metody „get()” na referencji do danych, które zamierzamy uzyskać. W tym przypadku pobierane są wszystkie zestawy nastaw progowych, aby przypisać je do zmiennych lokalnych. Te zmienne lokalne są następnie przekazywane za pośrednictwem interfejsu DataCallback, który jest kluczowy w komunikacji zwrotnej operacji asynchronicznej. Dzięki nadpisaniu metody onDataLoaded, interfejs ten umożliwia przekazywanie wyników operacji asynchronicznej do innych części aplikacji.

3.11. Problemy implementacyjne

Podczas implementacji projektu napotkałem wiele wyzwań, których rozwiązania w większości znajdowałem w dokumentacjach technicznych. Główne trudności wynikały z błędów w wyświetlaniu wykresów przy użyciu biblioteki MPAndroidChart oraz z początkowych problemów z zarządzaniem komunikacją między aplikacją mobilną a bazą danych, zwłaszcza przy wykorzystaniu LiveData.

Większość problemów związanych z biblioteką MPAndroidChart udało się rozwiązać dzięki dokładnej analizie metod i ich ustawień, korzystając z dostępnej dokumentacji technicznej [6]. Jednakże, pozostał jeden nierozwiązyany problem: błędne wyświetlanie etykiet osi Y na wykresach słupkowych natężenia światła, gdy wartość pomiarowa była mniejsza lub równa 300lx, a zakres osi przekraczał 1000lx. To rzadki przypadek, w którym wartość progowa natężenia światła znacznie zwiększała zakres wykresu. Ostatecznie udało się osiągnąć poprawne wyświetlanie etykiet osi Y poprzez dostosowanie wykresu, tak aby nie zwiększał zakresu w przypadku bardzo niskich pomiarów natężenia światła.

3.12. Wdrożenie i testy

Dzięki wykorzystaniu wbudowanego emulatora w środowisku Android Studio 11, możliwe było efektywne testowanie nowo dodawanych funkcjonalności w trakcie rozwoju projektu. Każdy nowo wprowadzany moduł był dokładnie analizowany pod kątem działania i zgodności z pozostałymi elementami systemu.

Oprócz testowania pojedynczych modułów i funkcji, projekt przeszedł również etap

praktycznego wdrożenia i był używany przez okres 10 dni, w trakcie którego działał nieprzerwanie i zgodnie z oczekiwaniami. Model demonstracyjny wykazał się wysoką responsywnością, reagując na zmiany w założonym okresie pięciu sekund, a aplikacja mobilna była aktualizowana co 10 sekund, zapewniając użytkownikowi ciągły dostęp do aktualnych danych.

4. Podsumowanie

4.1. Podsumowanie pracy

Celem niniejszej pracy inżynierskiej było zaprojektowanie i implementacja zaawansowanego systemu do zdalnego monitorowania i sterowania procesami, zrealizowanego na platformie Raspberry Pi 4B i w aplikacji mobilnej na system Android. System wykorzystuje czujniki do zbierania danych środowiskowych i steruje warunkami w modelu demonstracyjnym zautomatyzowanej szklarni. Wykonany projekt skutecznie integruje technologie mikrokomputerów, czujników, bazy danych w chmurze oraz mobilnej aplikacji użytkownika.

Kluczowe aspekty pracy obejmowały projektowanie i konfigurację modelu szklarni z wykorzystaniem czujników temperatury, wilgotności i natężenia światła. Wdrożony system efektywnie analizuje zebrane dane, reaguje na zmiany warunków środowiskowych i umożliwia zdalne zarządzanie poprzez aplikację mobilną. Komunikacja z bazą danych Firebase Realtime Database została zrealizowana z uwzględnieniem bezpieczeństwa i szyfrowania danych. Aplikacja mobilna, opracowana w języku Kotlin, zapewnia interaktywny interfejs dla użytkownika, umożliwiając monitorowanie, sterowanie i analizę danych w czasie rzeczywistym.

Testy systemu wykazały jego niezawodność i skuteczność, potwierdzając gotowość do praktycznego zastosowania w rzeczywistych warunkach. Projekt demonstrował również możliwości adaptacji i skalowalności technologii, co otwiera drogę do jego dalszego rozwoju i zastosowania w różnorodnych scenariuszach monitorowania i sterowania środowiskiem.

4.2. Dalszy rozwój projektu

Zaprezentowany projekt systemu monitorowania i sterowania, ma elementy o które można go rozwinąć praktycznie w każdym zaimplementowanym module. Skupię się jednak na najistotniejszych, które zamierzam w przyszłości wprowadzić w pierwszej kolejności.

Implementacja sterowania wentylatorem za pomocą sygnału PWM pozwoli na regu-

lację prędkości obrotowej silnika. Użytkownik mógłby dostosować jego prędkość poprzez ustawienie serii wartości progowych, tworząc krzywą prędkości obrotowej wentylatora.

Kolejnym istotnym elementem, o który warto rozbudować system jest dodanie widoku umożliwiającego sterowanie konkretnymi akcjami (np. wyłączeniem wentylatora) przy jednoczesnym podglądzie aktualnych wartości pomiarów. Korzystanie z tego widoku, powodowałoby zatrzymanie działania algorytmu sprawdzającego wartości progowe, pozostawiając użytkownikowi dowolność w sterowaniu systemem.

Dodatkowo, projekt może zostać rozszerzony o dodanie profilu administratora. Aktualnie dostęp do aplikacji, a co za tym idzie sterowania systemem, może mieć każdy kto założy konto. W proponowanym wzbogaceniu systemu, założenie konta pozwalałoby wyłącznie na podgląd danych, a możliwość edycji poprzez danego użytkownika, przydzielana byłaby z poziomu profilu administratora.

5. Wykaz elementów

Oznaczenie na schematach	Nazwa elementu	Typ lub parametr podstawowy	Ilość
U1	Raspberry Pi	4B	1
T1	Zasilacz	5,1V; 3A	1
U2, U3	Czujnik wilgotności	DHT11	2
U4	Czujnik temperatury	DS18B20	1
U5, U6, U7	Czujnik natężenia światła	BH1750	3
Q1	Tranzystor	IRLZ44N	1
M1	Wentylator	5V	1
D1, D2	Dioda LED	Niebieska	2
D3, D4	Dioda LED	Żółta	2
D5	Dioda LED	Czerwona	1
D6	Dioda Zenera	6,2V	1
R1, R2, R3, R4, R5, R7	Rezystor	220Ω	6
R6	Rezystor	10kΩ	1
R8	Rezystor	4,7kΩ	1
R9, R10	Rezystor	1,8kΩ	2

Tabela 1: Wykaz elementów

Literatura

- [1] Antonio Leiva, *Kotlin for Android Developers*, (2017). <https://leanpub.com/kotlin-for-android-developers>, Ostatnio dostępna: 11 Grudnia 2023.
- [2] Sabina Jeschke, Christian Brecher, Houbing Song, Danda B. Rawat, *Industrial Internet of Things*, (2017). <https://link.springer.com/book/10.1007/978-3-319-42559-7>, Ostatnio dostępna: 11 Grudnia 2023.
- [3] Cristina Alcaraz, *Security and Privacy Trends in the Industrial Internet of Things*, (2019). <https://link.springer.com/book/10.1007/978-3-030-12330-7>, Ostatnio dostępna: 11 Grudnia 2023.
- [4] JetBrains, Kotlin documentation, (2023). <https://kotlinlang.org/docs/home.html>, Ostatnio dostępna: 11 Grudnia 2023.
- [5] Google, Firebase documentation, (2023). <https://firebase.google.com/docs>, Ostatnio dostępna: 11 Grudnia 2023.
- [6] Philipp Jahoda, MPAndroidChart documentation, (2021).
<https://weeklycoding.com/mpandroidchart-documentation>, Ostatnio dostępna: 11 Grudnia 2023.