



Politechnika Łódzka

Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

## PRACA DYPLOMOWA INŻYNIERSKA

### **System automatyki szklarni z wykorzystaniem sieci bezprzewodowej**

*Greenhouse automation system using a wireless network*

*Karol Nowaliński*

*Numer albumu:*

**228438**

*Promotor: dr inż. Tomasz Rybicki*

Łódź 2023

<b>1. Wstęp .....</b>	<b>3</b>
<b>2. Cel i zakres pracy .....</b>	<b>5</b>
<b>3. Mikroklimat.....</b>	<b>6</b>
3.1. Wprowadzenie .....	6
3.2. Wpływ warunków mikroklimatu w szklarni na rośliny .....	7
<b>4. Systemy automatyki wykorzystywane w szklarniach.....</b>	<b>9</b>
4.1. Wprowadzenie .....	9
4.2. Przegląd urządzeń pomiarowych przeznaczonych do automatyzacji pomiarów w szklarni .....	9
4.3. Przegląd jednostek pomiarowo - sterujących .....	13
<b>5. Prace nad projektem systemu pomiarowego z możliwością integracji z urządzeniami wykonawczymi .....</b>	<b>15</b>
5.1. Założenia.....	15
5.2. Część sprzętowa.....	16
5.2.1 Mikrokontroler .....	16
5.2.2. Czujniki .....	17
5.2.3 Schemat elektryczny urządzenia pomiarowego .....	20
5.2.4 Obwód PCB.....	21
5.2.5 Zabezpieczenie przed warunkami zewnętrznymi.....	23
5.3. Oprogramowanie.....	30
5.3.1. Komunikacja na linii mikrokontroler – serwer .....	30
5.3.2. Oprogramowanie mikrokontrolera .....	30
5.3.2.1 Wprowadzenie .....	30
5.3.2.2. Schematy blokowe i omówienie działania programu .....	32
5.3.3. Oprogramowanie serwerowe.....	41
5.3.3.1 Wprowadzenie .....	41
5.3.3.2 Omówienie elementów części serwerowej .....	43
5.3.4. Główny interfejs programu .....	50
5.3.5. Interfejs rysujący wykresy .....	55
5.3.6. Symulowanie działania z kilkoma czujnikami .....	59
<b>6. Podsumowanie i wnioski .....</b>	<b>61</b>
<b>7. Literatura .....</b>	<b>63</b>
<b>8. Streszczenie.....</b>	<b>65</b>
<b>9. Abstract.....</b>	<b>66</b>

## **1. Wstęp**

W obecnych czasach produkcja żywności stanowi kluczowy czynnik dla gospodarki i życia ludzi. Wraz z szybkim rozwojem gospodarczym rośnie zapotrzebowanie na produkty spożywcze, co z kolei prowadzi do zwiększenia nakładów związanych z produkcją.

Sektor rolniczy jako podstawowe źródło dostaw żywności, również musi dostosować się do wymagań rynku. W tym celu producenci wykorzystują coraz bardziej wydajne rozwiązania, takie jak technologie uprawy pod osłonami, które pozwalają na kontrolowanie mikroklimatu i nawożenia roślin. W tym kontekście, odpowiednie warunki klimatyczne stanowią kluczowy czynnik wpływający na plon i jakość upraw. Dlatego też producenci stosują zaawansowane rozwiązania technologiczne, takie jak automatyczne wietrzenie i komputery klimatyczne, aby zapewnić roślinom odpowiednie warunki rozwoju.

Warunki klimatyczne są kluczowym czynnikiem rozwoju roślin. Czynniki mikroklimatyczne odpowiadają za ilość i jakość plonów, jakie wydają rośliny, z tego powodu w szklarniach i tunelach foliowych reguluje się dwa podstawowe parametry – temperaturę oraz wilgotność powietrza. Są one istotne ze względu na ich wpływ na proces fotosyntezy i transpiracji [2].

Fotosynteza to podstawowa naturalna przemiana biochemicalna, w trakcie której roślina przetwarza składniki odżywczce w związki organiczne niezbędne do jej życia oraz rozwoju. Transpiracja to proces parowania wody z części nadziemnej wody, przy równoczesnym pobieraniu i transportowaniu wody wraz z składnikami odżywczymi z ziemi poprzez część korzeniową rośliny [2].

Procesy fotosyntezy i transpiracji można regulować przez utrzymywanie konkretnej temperatury i wilgotności powietrza. Źle dobrane warunki mikroklimatyczne mogą spowolnić transpirację, obniżając jakość produktów fotosyntezy [2].

Optymalne warunki dla transpiracji to wysoka temperatura (powyżej 28°C) oraz wilgotność poniżej 40%. Spowoduje to, że roślina próbując chłodzić część naziemną będzie pobierać dużą ilość wody z części korzeniowej. W warunkach zbyt wysokiej wilgotności (powyżej 80%) roślina traci możliwość transpiracji, transfer wody nie następuje, w rezultacie czego nie są transportowane składniki odżywczce [2].

Wraz z rozwojem technologii zyskują również sektory takie jak rolnictwo, które ma możliwość zaimplementowania sprawdzonych technologii rozwijanych przez lata w innych

gałęziach przemysłu. Reakcja na zmiany w mikroklimacie może odbywać się manualnie lub być zautomatyzowana poprzez zastosowanie automatycznego wietrzenia, dogrzewania itp. Za automatyzację zapewniania odpowiednich warunków w obiektach najczęściej odpowiadają komputery klimatyczne, czyli jednostki, sterujące automatyką szklarni i tuneli foliowych. Bardziej zaawansowane urządzenia potrafią zaplanować działanie z wyprzedzeniem na podstawie danych z serwerów meteorologicznych. Są one bardzo drogie, przez co ich stosowanie jest najbardziej opłacalne przy dużych uprawach.

Uprawy na mniejszą skalę nie są jednak skazane na całkowite wykluczenie z procesu automatyzacji. Rozwój technologii pozwala na efektywne monitorowanie warunków mikroklimatu panującego w szklarniach i obiektach foliowych poprzez stosowanie szeregu czujników takich parametrów jak temperatura i wilgotność powietrza, wilgotność gleby, pH gleby, naświetlenie i wiele innych. Wykorzystywanie urządzeń pomiarowych pozwala z odpowiednim wyprzedzeniem zareagować na zmiany mikroklimatu, które mogą negatywnie wpływać na jakość plonów.

Istnieje szereg urządzeń pomiarowych pozwalających na monitorowanie najważniejszych parametrów mikroklimatu w hodowli z wykorzystaniem szklarni i tuneli foliowych, które pozwalają na częściowe zautomatyzowanie procesów w małych i średnich gospodarstwach.

## **2. Cel i zakres pracy**

Celem pracy jest zaprojektowanie, wykonanie prototypu oraz oprogramowanie systemu automatyki szklarni, który będzie umożliwiał komunikacje z jednostkami wykonawczymi oraz zdalne monitorowanie parametrów takich jak wilgotność i temperatura powietrza oraz wilgotność gleby przy pomocy interfejsu przeglądarkowego. Projekt ma powstać z wykorzystaniem systemu mikroprocesorowego z interfejsem bezprzewodowym oraz dedykowanych czujników na zaprojektowanym obwodzie PCB.

Zakres prac:

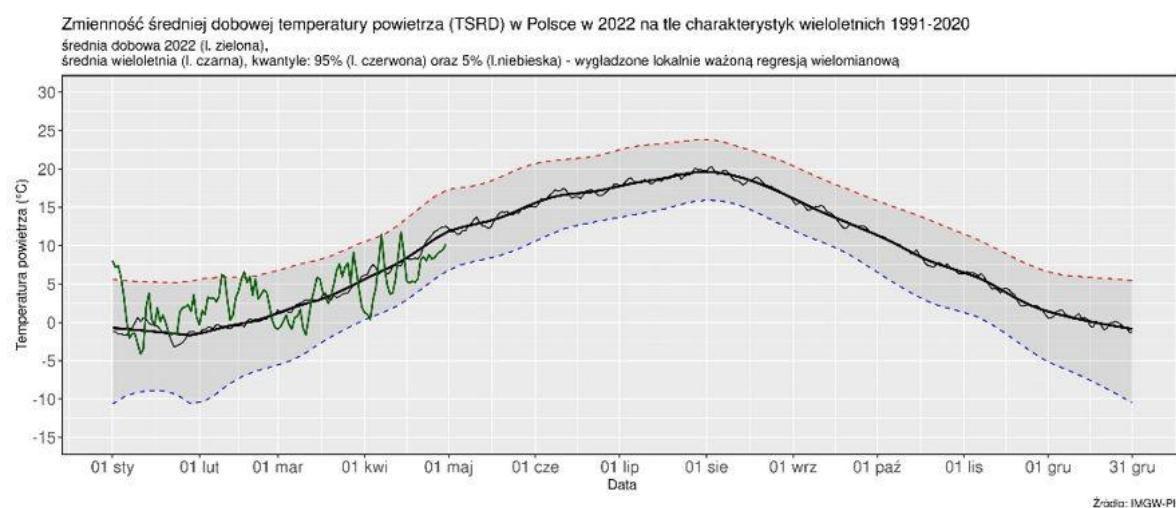
- Zbudowanie i oprogramowanie prototypu urządzenia pomiarowego bazującego na mikrokontrolerze ESP32, czujniku temperatury i wilgotności powietrza SHT30 oraz pojemnościowym czujniku wilgotności gleby.
- Dobór protokołu komunikacyjnego zapewniającego stabilną wymianę informacji pomiędzy rozproszonymi urządzeniami pomiarowymi a serwerem i urządzeniami wykonawczymi.
- Stworzenie aplikacji serwerowej odpowiadającej za zbieranie i przetwarzanie informacji z urządzeń pomiarowych oraz za komunikację z urządzeniami wykonawczymi.
- Zaimplementowanie prostego interfejsu pozwalającego na odczyt aktualnych wartości przekazanych przez urządzenia pomiarowe oraz dostęp do danych archiwalnych.

### 3. Mikroklimat

#### 3.1. Wprowadzenie

Klimatem nazywa się zespół zjawisk występujących w pogodzie, na określonym obszarze. Składa się z takich czynników jak ciśnienie atmosferyczne, prędkość i kierunek wiatru, temperatura, wilgotność powietrza [1].

Klimat określa, jakie rośliny mogą żyć i poprawnie rozwijać się na danym terenie. Polska znajduje się w strefie klimatu umiarkowanego, w którym łatwo rozróżnić pory roku, czyli okresy występowania charakterystycznych warunków atmosferycznych. Wiosnę charakteryzuje duża wilgotność powietrza, temperatury w zakresie około 10-17 °C oraz umiarkowaną ilością opadów. W lecie temperatury osiągają ponad 30 °C, wilgotność jest bardzo niska. Na Rys.3.1 została przedstawiona zmienność średniej dobowej temperatury powietrza. Pokazuje on, jakie wahania są możliwe w umiarkowanej strefie klimatycznej i jakie temperatury mogą występować w poszczególnych miesiącach. Łatwo zauważać podział roku na pory roku [1,2].



Rys. 3.1 Zmienność średniej dobowej temperatury powietrza w Polsce w 2022 roku [10].

Klimat Polski pozwala na uprawianie różnorodnej roślinności przeznaczonej do spożycia. Dodatkowo stosunkowo dobra dostępność wód gruntowych pozwala na zniwelowanie negatywnego wpływu niskiej wilgotności i wysokich temperatur w okresie letnim, kiedy uprawy są najbardziej opłacalne ze względu na brak konieczności dogrzewania

obiektów uprawnych oraz najlepsze warunki oświetleniowe w kontekście efektywnie przeprowadzanej fotosyntezy.

### **3.2. Wpływ warunków mikroklimatu w szklarni na rośliny**

Klimat to jeden z najważniejszych czynników oddziałujących na rośliny, dlatego tak ważne jest zagwarantowanie odpowiedniego mikroklimatu w szklarni lub obiekcie foliowym. Temperatura i wilgotność powietrza warunkują jak skuteczna jest fotosynteza i transpiracja.

Temperatura wpływa na wzrost i rozwój roślin poprzez warunkowanie jakości procesu fotosyntezy. Cykl życia rośliny wymaga zachowania odpowiedniej temperatury. Zbyt wysoka powoduje, że roślina zwiększa zapotrzebowanie na wodę, aby schłodzić część nadziemną rośliny. Kosztem zwiększonego zapotrzebowania na wodę jest zmniejszenie zapotrzebowania na składniki mineralne. Skutkuje to obniżeniem jakości procesu fotosyntezy, co za tym idzie zmniejszeniem ilości jej produktów i spowolnieniem rozwoju rośliny. Temperaturę można obniżyć jedynie poprzez podanie wody doglebowo. Bezpośrednie polanie rozgrzanej części nadziemnej rośliny spowoduje szok termiczny i osłabienie rośliny. Optymalna temperatura mieści się w zakresie 18-24°C. Pozwala to na odpowiedni transport wody i składników mineralnych do części nadziemnej rośliny, która odpowiada za fotosyntezę [2].

Wilgotność jest częścią składową klimatu i ważnym elementem mikroklimatu w szklarniach. Pojęciem wilgotność określa się zawartość pary wodnej w powietrzu. Zwykle jest ona podawana jako wilgotność względna, czyli wyrażony w procentach stosunek ciśnienia cząsteczkowego pary wodnej zawartej w powietrzu do prężności pary wodnej nasyconej nad płaską powierzchnią czystej wody w tej samej temperaturze. Ma ona duży wpływ na mechanizmy zachodzące w części nadziemnej rośliny [2].

Głównym procesem, którego działanie reguluje wilgotność jest transpiracja, czyli zjawisko czynnego parowania wody z liści przez aparaty szparkowe. Proces ten warunkuje przepływ wody i składników mineralnych z korzenia do części nadziemnej rośliny. Im wyższa wilgotność powietrza, tym niższa intensywność procesu transpiracji a co za tym idzie mniejszy przepływ składników odżywczych w roślinie. Optymalnym zakresem wilgotności dla rozwoju roślin jest 50%-65%. Pozwala to uzyskać najlepsze plony [2].

Jak widać warunki klimatyczne mają ogromny wpływ na cykl życia rośliny, dlatego tak ważne jest ich kontrolowanie. Drugim wnioskiem, który można wyciągnąć analizując wpływ temperatury i wilgotności powietrza na transport wody pomiędzy korzeniem a częścią nadziemną roślin jest to, że podstawowym warunkiem, aby taki transport w ogóle zaszedł, jest zapewnienie odpowiedniej wilgotności gleby [2].

## **4. Systemy automatyki wykorzystywane w szklarniach**

### **4.1. Wprowadzenie**

Parametry w szklarniach i tunelach foliowych w gospodarstwach wielkoformatowych monitorowane są poprzez skomplikowane i drogie komputery klimatyczne, które oprócz samego przeprowadzania pomiarów, sterują elementami wykonawczymi w obiektach.

Zastosowanie takiego rozwiązania w średnich i małych gospodarstwach, a tym bardziej w przydomowych szklarniach jest nieopłacalne. Z tego powodu w niektórych uprawach nie stosuje się żadnych urządzeń pomiarowych, zdane są one na ludzkie zmysły.

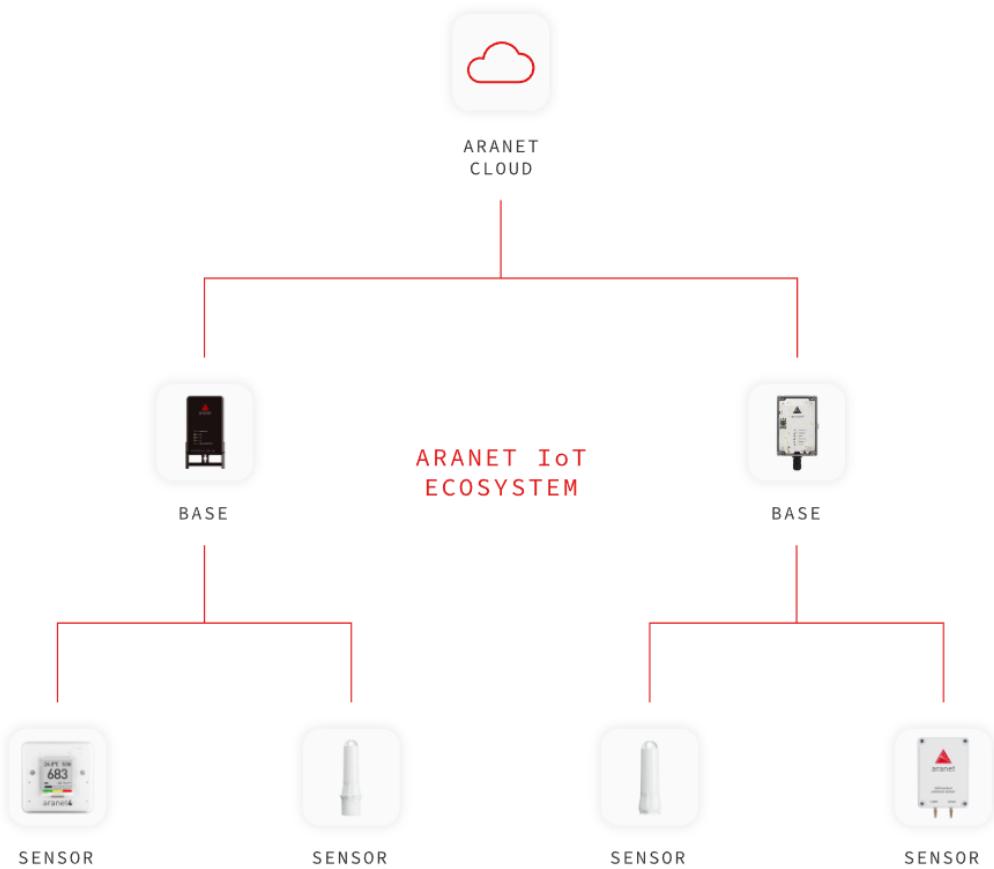
Lepszą alternatywą jest stosowanie termometrów i higrometrów, które pozwalają na zbadanie parametrów mikroklimatu – ich minusem jest to, że wymagają fizycznej obecności osoby odczytującej, aby poznać aktualne parametry mikroklimatu panującego w obiekcie – wyklucza to możliwość odpowiednio wczesnego zareagowania na zmieniające się warunki.

Aby zautomatyzować działanie szklarni powstała cała gałąź urządzeń pomiarowych przeznaczonych do badania mikroklimatu z wykorzystaniem sieci bezprzewodowych. Firmy produkujące takie urządzenia stosują różne podejścia do komunikacji bezprzewodowej oraz do połączenia z urządzeniami wykonawczymi

### **4.2. Przegląd urządzeń pomiarowych przeznaczonych do automatyzacji pomiarów w szklarni**

Jedną z firm produkujących urządzenia pomiarowe dedykowane do wykorzystania w szklarniach jest Aranet. Producent oferuje całą gamę różnego rodzaju czujników mających za zadanie badanie parametrów mikroklimatycznych oraz gleby.

Firma Aranet stworzyła swój własny ekosystem pomiarowy (*Rys.4.1*), który do działania potrzebuje co najmniej dwóch dedykowanych urządzeń – sensora i stacji bazowej. Producent nie udostępnia szczegółowych informacji na temat sposobu łączności pomiędzy urządzeniami. Dostęp do danych gromadzonych przez stację bazową możliwy jest poprzez protokoły MQTT i Modbus z możliwością konwersji na inne protokoły przy pomocy urządzeń proxy [3].



Rys. 4.1. Schemat ekosystemu Aranet IOT [3].

Urządzenia pomiarowe firmy Aranet w zależności od wersji pozwalają na monitorowanie takich parametrów, jak wilgotność i temperatura powietrza, wilgotność i pH gleby, stężenie CO<sub>2</sub> w powietrzu, nasłonecznienie, ciśnienie atmosferyczne. Przez to, że producent oferuje przemyślany system oparty na wysokiej jakości podzespołach, urządzenia oferowane przez Aranet są stosunkowo drogie. Stacja bazowa wraz z jednym, podstawowym czujnikiem to koszt około 3500 zł (Rys. 4.2) [3].



Rys. 4.2. Podstawowy czujnik firmy Aranet wraz ze stacją bazową [3].

Tańszą alternatywą jest produkt firmy UbiBot – urządzenie WS1 PRO WiFi LTE oferuje możliwość wykonywania pomiarów temperatury i wilgotności powietrza oraz nasłonecznienia.

Urządzenie przedstawione na Rys. 4.3. ma postać stacji bazowej z wbudowanym ekranem. Pozwala na odczyt parametrów będąc w obiekcie oraz dzięki zaimplementowanej łączności bezprzewodowej pozwala na zdalny dostęp do danych pomiarowych. Producent przewidział możliwość rozbudowy urządzenia o zewnętrzne czujniki podłączane do portów stacji bazowej [4].

Koszt stacji bazowej firmy UbiBot to około 1000 zł.



Rys. 4.3. Stacja bazowa UbiBot WS1 PRO WiFi LTE [4].

Firma UbiBot wyszła również naprzeciw oczekiwaniom mniej wymagających klientów. Producent w cenie około 400 zł oferuje urządzenie pomiarowe typu all-in-one przedstawione na Rys. 4.4. pozwalające na pomiar wilgotności i temperatury powietrza oraz nasłonecznienia. Urządzenie UbiBot WS1 WiFi w porównaniu do WS1 PRO WiFi LTE zostało pozbawione wyświetlacza oraz łączności poprzez sieć komórkową. Została zachowana możliwość rozbudowy o zewnętrzne sondy [4].



Rys. 4.4. Urządzenie all-in-one UbiBot WS1 WiFi [4].

Omawiane wyżej urządzenia firmy UbiBot, aby mieć możliwość pobierania danych przez sieć bezprzewodową, wymagają stosowania dedykowanej aplikacji oraz serwerów. Uzależnia to użytkownika od decyzji producenta, który przez to, że urządzenia nie mają otwartej budowy i oprogramowania może wyłączyć całą linię produktu z dnia na dzień [4].

Urządzenia omawiane w tym rozdziale są w stanie wykonywać pomiary, jednak producenci nie przewidzieli możliwości integracji urządzeń pomiarowych z urządzeniami wykonawczymi. Nie podano informacji o tym, czy jest możliwe przesyłanie danych do komputera klimatycznego, który oprócz interpretacji wyników miałby możliwość reakcji na zmiany mikroklimatu szklarni.

#### **4.3. Przegląd systemów pomiarowo – wykonawczych**

Za połączenie funkcji pomiarowej i sterującej odpowiadają komputery klimatyczne, czyli jednostki sterujące klimatem obiektu za pomocą urządzeń wykonawczych takich jak automatyczne lufty, mieszalniki powietrza, zraszacze, zespoły grzewcze i urządzenia nawadniające.

Komputery klimatyczne najczęściej składają się z urządzeń pomiarowych oraz sterownika PLC lub dedykowanych urządzeń sterujących współpracujących z jednostką obliczeniową.

Jednym z dostępnych na rynku systemów jest Ridder HortiMaX-Go (Rys. 4.5). Jego głównym elementem jest jednostka obliczeniowa wyposażona w ekran dotykowy pozwalający na łatwe sterowanie pracą podłączonych urządzeń. Umożliwia zdalny dostęp przez smartphone lub komputer osobisty [13].

Funkcję wykonawczą pełni „przełącznik inteligentny”, czyli zespół zawierający przekaźniki oraz wejścia i wyjścia analogowe. Urządzenie pomiarowe umożliwia odczyt temperatury oraz wilgotności w szklarni [13].

Koszt podstawowej wersji systemu to około 6000 zł. Producent za dodatkową opłatą oferuje urządzenia pomiarowe rozbudowujące możliwości systemu o pomiar warunków panujących na zewnątrz szklarni oraz pomiar stężenia CO<sub>2</sub> [13].



Rys. 4.5. System Ridder HortiMaX-Go [13].

Komputery klimatyczne oparte na sterownikach PLC są ciężkie do scharakteryzowania, ponieważ rzadko występują w postaci gotowych systemów ze względu na dowolność w zastosowaniu urządzeń wykonawczych i pomiarowych. Dobór parametrów systemu zależny jest całkowicie od wymagań, jakie będzie miał klient oraz od tego, jakim budżetem dysponuje.

## **5. Prace nad projektem systemu pomiarowego z możliwością integracji z urządzeniami wykonawczymi**

### **5.1. Założenia**

Analiza rynku popularnych urządzeń i systemów pomiarowych pozwoliła na lepsze określenie założeń, jakie powinno spełniać przystępne cenowo urządzenie pomiarowe do wykorzystania w szklarni.

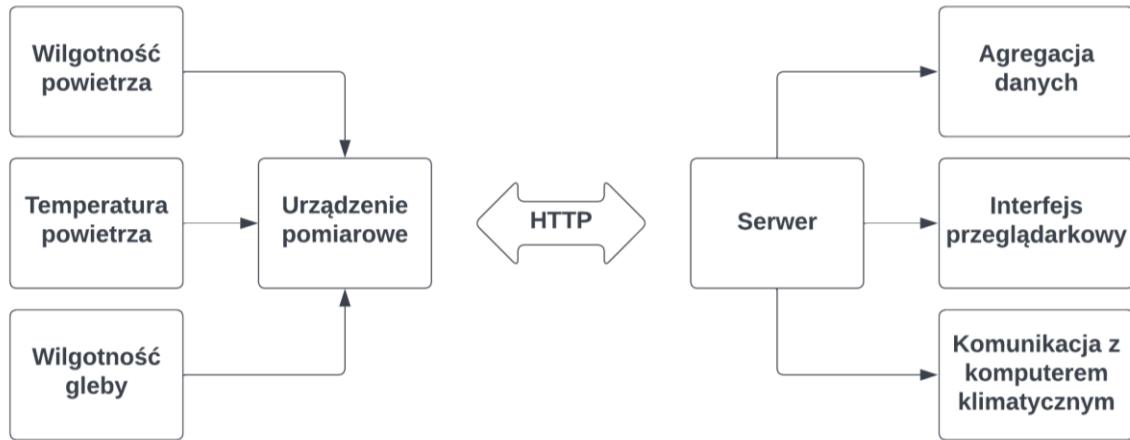
Część sprzętowa urządzenia powinna być otwarta (open source hardware) i oparta na łatwo dostępnych podzespołach. Umożliwi to prostą naprawę w przypadku awarii któregoś z podzespołów oraz możliwość łatwego rozbudowania urządzenia o kolejne funkcjonalności przez użytkowników.

Część programowa powinna być oparta na zasadach open source i nie powinna wymagać stosowania zewnętrznych serwerów. Otwarty kod oprogramowania pozwoli użytkownikom na ewentualną rozbudowę systemu o kolejne funkcjonalności. Brak konieczności stosowania zewnętrznych serwerów nie uzależnia działania systemu od decyzji zewnętrznych podmiotów. Interfejs użytkownika powinien być dostępny z poziomu przeglądarki, nie może być ograniczony jedynie do aplikacji mobilnej.

Komunikacja sieciowa powinna odbywać się z wykorzystaniem prostego w obsłudze protokołu sieciowego i nie może wymagać stosowania dodatkowych punktów komunikacyjnych. Implementacja przesyłania danych musi być na poziomie urządzenia pomiarowego.

Pomiary powinny móc być wykorzystane przez komputer klimatyczny odpowiadający za warunki w obiekcie poprzez sterowanie urządzeniami wykonawczymi takimi jak systemy zraszające, lufty wietrzące i mieszalniki powietrza.

*Rys. 5.1. w graficzny sposób przedstawia funkcjonalności, jakie zostały przewidziane i zaimplementowane w projekcie systemu.*



Rys. 5.1. Schemat blokowy projektu.

## 5.2. Część sprzętowa

Częścią sprzętową projektu jest stworzenie urządzenia pomiarowego pozwalającego na wykonywanie zdalnych pomiarów temperatury i wilgotności powietrza oraz wilgotności gleby w szklarni. W tej sekcji pracy opisane zostały zastosowane rozwiązania sprzętowe realizujące w praktyce założenia ze schematu blokowego przedstawionego na Rys. 5.1.

### 5.2.1 Mikrokontroler

W projekcie został zastosowany przedstawiony na Rys 5.2 mikrokontroler ESP32 (ESP-WROOM-32). Jest to moduł wyposażony w dwurdzeniowy mikroprocesor Tensilica LX6 240 MHz, 520KB SRAM, 4MB pamięci flash oraz obsługę sieci Wi-Fi oraz Bluetooth [5].



Rys. 5.2. Mikrokontroler ESP-WROOM-32.

Parametry jakimi cechuje się wybrany mikrokontroler są tylko częściowo wykorzystywane w projekcie. Wybór modułu został podyktowany przede wszystkim jego szeroką dostępnością, stosunkowo niską ceną, wbudowaną obsługą sieci Wi-Fi oraz wsparciem dla frameworka Arduino [5].

Zasoby sprzętowe, jakie wykorzystywane są w projekcie to łączność Wi-Fi, jeden z dwóch przetworników analogowo – cyfrowych, magistrala I<sup>2</sup>C oraz dwa rdzenie.

### 5.2.2. Czujniki

Jako czujnik temperatury został wykorzystany przedstawiony na Rys. 5.3. moduł SHT30. Jest to sensor o dokładności rzędu 0.3°C w pomiarze temperatury oraz 3 punkty procentowe w pomiarze wilgotności. Obsługuje zakres [T]-40-125°C i [H]0-100%. Jest to wystarczająca dokładność jak i rozpiętość pomiarów w tym zastosowaniu [6].

Sensor podłączany jest do mikrokontrolera za pomocą magistrali I<sup>2</sup>C. W zastosowanym układzie linia danych (SDL) podłączona jest do pinu D21, a linia zegara (SCL) do pinu D22.

Zastosowanie czujnika z komunikacją opartą na magistrali I<sup>2</sup>C pozwala na ewentualne rozszerzenie urządzenia pomiarowego o dodatkowy sensor, dla zwiększenia niezawodności pomiarów przy braku konieczności wykorzystywania dodatkowych złącz GPIO mikrokontrolera. Jest to możliwe, ponieważ moduł mimo domyślnie ustawionego adresu I<sup>2</sup>C na 0x44 pozwala go zmienić na 0x45 po fizycznej zmianie konfiguracji przez zwarcie odpowiednich pinów.



Rys. 5.3. Czujnik temperatury i wilgotności powietrza SHT30.

Czujnik wilgotności gleby (Rys. 5.4) jest prostym urządzeniem, którego działanie odbywa się na zasadzie pomiaru zmiany pojemności medium, w którym się znajduje. Odczyty realizowane są przez interpretacje wartości na wyjściu analogowym sensora.



Rys. 5.4. Pojemnościowy czujnik wilgotności gleby.

Zakres wartości sygnału został oszacowany poprzez zrobienie testu, w którym suchy czujnik został umieszczony w wodzie. Uzyskany w ten sposób zakres został przeliczony na wartości od ~0% do 100%. Aby przeprowadzić sensowną kalibrację odczytów czujnika dobrze by było przeprowadzić serie pomiarów w komorze klimatycznej o zmiennej wilgotności powietrza.

Im wyższa wilgotność badanego medium, tym niższa wartość sygnału na wyjściu czujnika. Zmiana w wartości wyjściowej zaobserwowanej podczas szacowania zakresu pomiarowego została przedstawiona na oscylogramie (Rys.5.5). Można na nim zauważyc, że w pomiarze występują duże zakłócenia.



Rys. 5.5.. Oscylogram pokazujący zmianę na wyjściu analogowym czujnika w trakcie zanurzania w wodzie.

Brak wiarygodnej dokumentacji (jedyne dostępne informacje znajdują się na stronach dystrybutorów i nie podają źródeł) i jakiegokolwiek wzorcowania czujnika powoduje, że uzyskane w ten sposób pomiary są w najlepszym wypadku poglądowe. Procentowe wartości wilgotności gleby obliczane przez program obsługujący urządzenie pomiarowe należy traktować jedynie jako sugestię i w przypadku faktycznego stosowania przeprowadzić testy, które pozwolą stwierdzić, jakie wartość wskazana przez czujnik będzie odpowiednia dla prowadzonych upraw.

Mimo swoich wad, zastosowany czujnik, jest najlepszym rozwiązańiem w swoim pułapie cenowym. Alternatywne konstrukcje opierają się o pomiar rezystancji gleby, który jest mniej dokładny ze względu na zmienną rezystancję pod wpływem stosowania różnych nawozów [12].

Oprócz dokładności największym problemem czujników rezystancyjnych jest to, że niezaizolowane elektrody pomiarowe ulegają w stosunkowo szybkim czasie korozji (Rys. 5.6) przyspieszonej zjawiskiem elektrolizy, która zachodzi, gdy tylko badana próbka gleby jest wilgotna. Rozwiązaniem problemu korozji mogłoby być zaprojektowanie czujnika zasilanego przebiegiem przemiennym o średnim napięciu równym zeru, jednak nadal nie poprawiłoby to dokładności sensora.

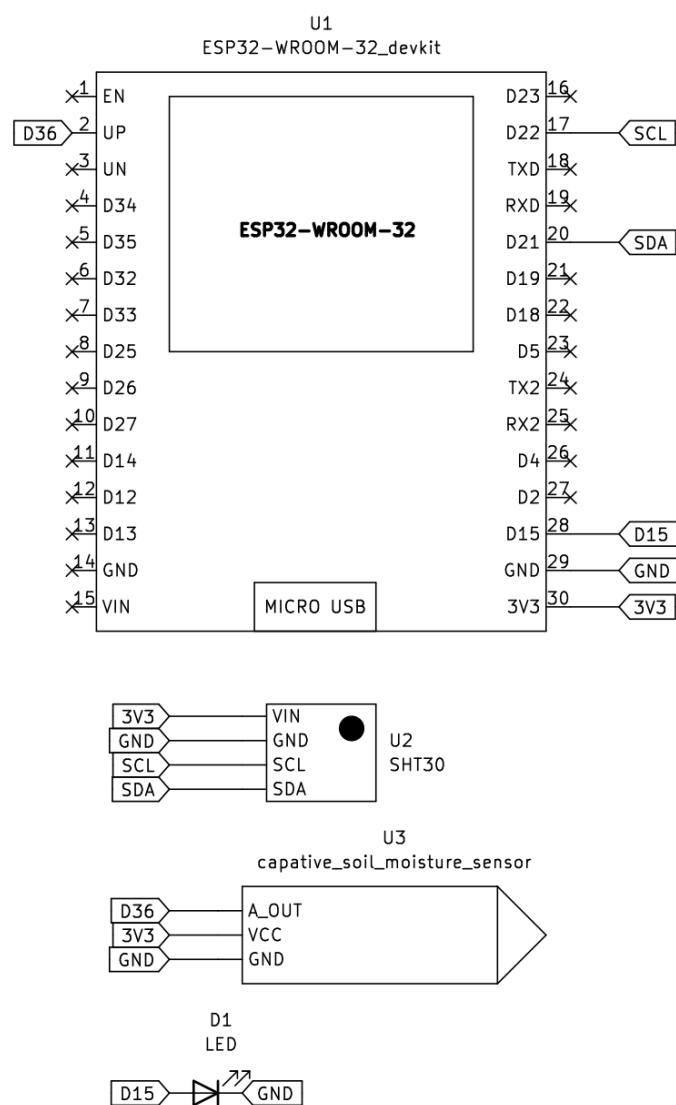


Rys. 5.6. Skorodowany rezystancyjny czujnik wilgotności gleby [12].

### 5.2.3 Schemat elektryczny urządzenia pomiarowego

Połączenia w projekcie zostały przedstawione na schemacie elektrycznym (Rys. 5.7.) wykonanym przy pomocy oprogramowania KiCad.

Oprócz podstawowych elementów wynikających ze specyfikacji projektu (mikrokontroler, czujnik temperatury i wilgotności powietrza oraz czujnik wilgotności gleby) zastosowana została dioda LED (D1), która na etapie testowania, wraz z wbudowaną diodą podłączoną do pinu D2 służyły do wizualizacji działania procedur na dwóch rdzeniach procesora.



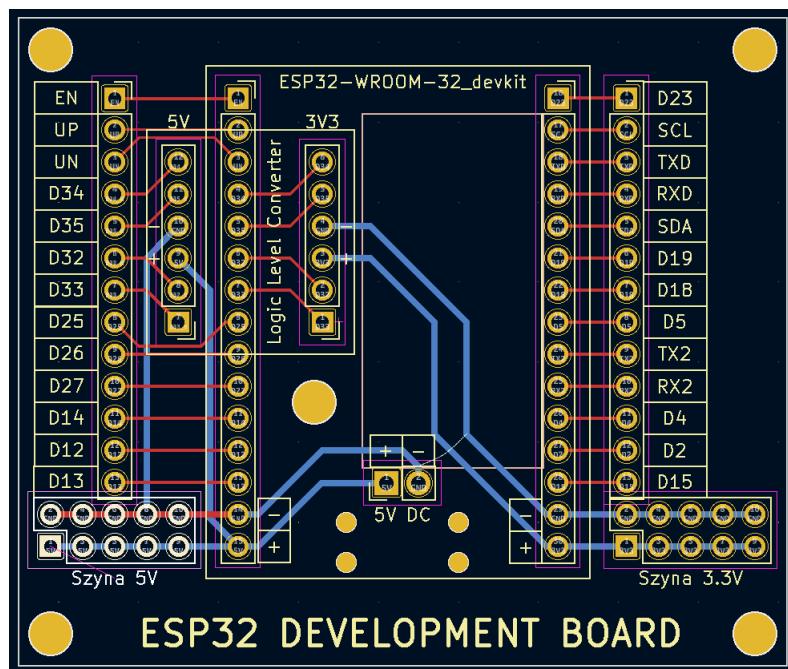
Rys. 5.7. Schemat elektryczny projektu.

## 5.2.4 Obwód PCB

W projekcie zastosowana została płytka PCB. Ze względów ekonomicznych została zaprojektowana, jako płytka deweloperska z możliwością zastosowania konwertera stanów logicznych do zamiany stosowanej logiki 3V3 na 5V. Płytkę została zaprojektowana od podstaw w programie KiCad a następnie jej wykonanie zostało zlecone zewnętrznej firmie.

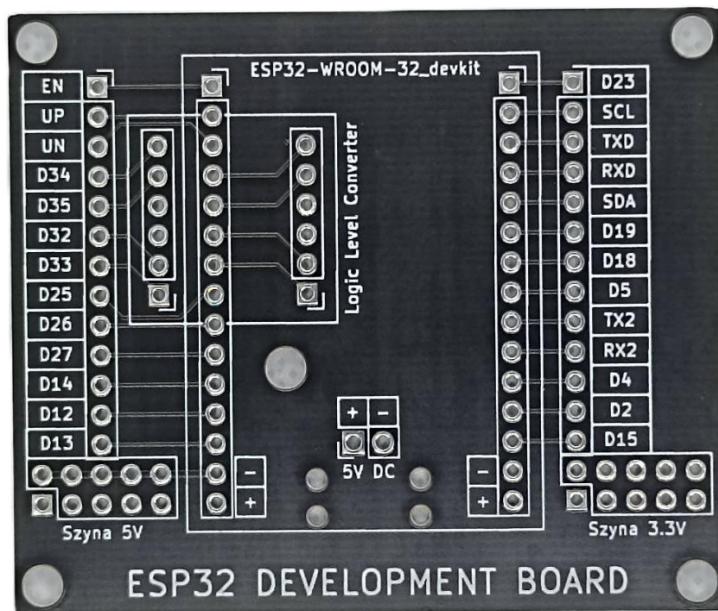
Cała część elektroniczna została umieszczona na płytce PCB. Projekt płytki został przedstawiony na Rys. 5.8. Zastosowanie takiego rozwiązania zostało podyktowane ograniczonym czasem wykonania pracy – stworzenie optymalnego obwodu drukowanego mogłoby wymagać kilku rewizji projektu, co przy okresie oczekiwania na zrealizowanie zlecenia oscylującym w granicy 6 tygodni byłoby bardzo czasochłonne. Dodatkowo przy minimalnej ilości zamówionych płytEK wynoszącej 5 sztuk wygenerowałoby dużo zbędnych odpadów. Płytkę uniwersalną pozwoli na ponowne wykorzystanie konstrukcji w projektach opartych na mikrokontrolerze ESP32.

Płytkę uniwersalna zawiera wyjścia wszystkich dostępnych pinów mikrokontrolera, możliwość podłączenia zewnętrznego zasilania 5V, które pozwala na zastosowanie gniazda DC, cechującego się większą trwałością od zintegrowanego z mikrokontrolerem portu Micro USB oraz możliwością prostej wymiany w wypadku uszkodzenia.



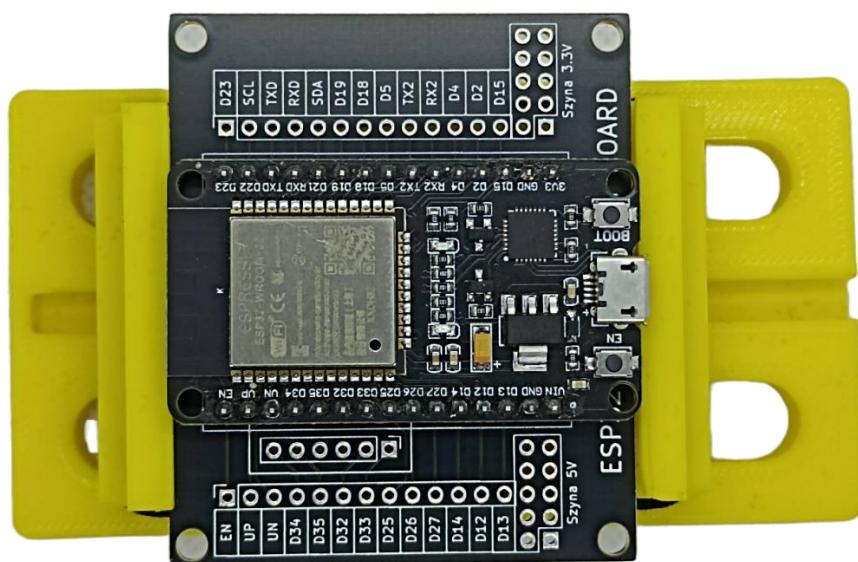
Rys. 5.8. Projekt płytki w programie KiCad.

Rys. 5.10. przedstawia gotową płytę wykonaną przez zewnętrzną firmę. Uzyskany efekt jest zadowalający. Obwód drukowany jest bardzo dobrej jakości.



Rys. 5.10. Wykonana płytkę.

Na Rys. 5.11. przedstawiona została płytka z zamontowanym mikrokontrolerem umieszczona w specjalnym imadle, którego celem było zredukowanie szansy na wystąpienie zwarć wywołanych ciałami obcymi w fazie testów urządzenia bez obudowy.



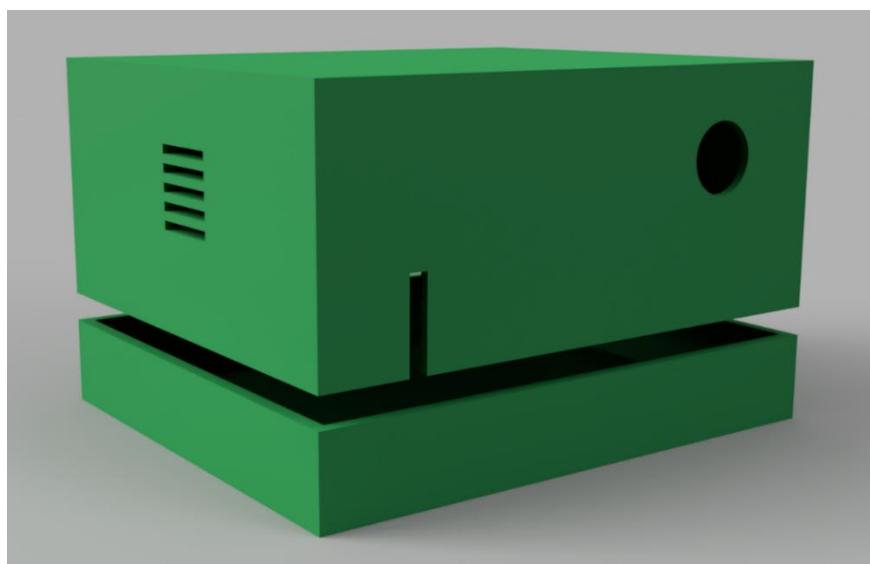
Rys. 5.11. Płytkę z zamontowanym mikrokontrolerem.

### **5.2.5 Zabezpieczenie przed warunkami zewnętrznymi**

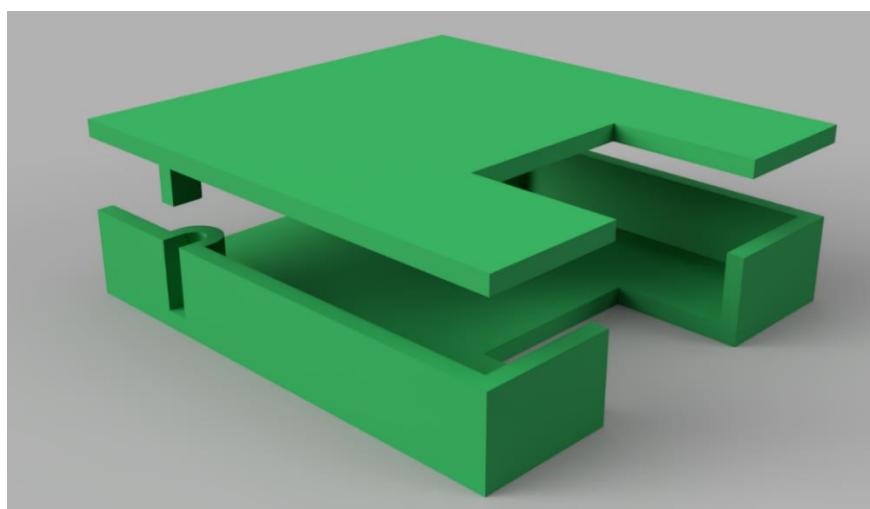
Aby zabezpieczyć podzespoły urządzenia pomiarowego przed działaniem warunków zewnętrznych i przypadkowymi zwarciami zostały wykonane obudowy (oddzielna dla zespołu mikrokontrolera z sensorem SHT30 oraz pojemnościowego czujnika wilgotności gleby).

Obudowy zostały zaprojektowane w programie Fusion360 – jest to oprogramowanie CAD, które stanowi hobbystyczną alternatywę dla skierowanego do zastosowań profesjonalnych Autodesk Inventor.

Rendery projektów obudów zostały przedstawione na Rys. 5.11. i Rys.12.



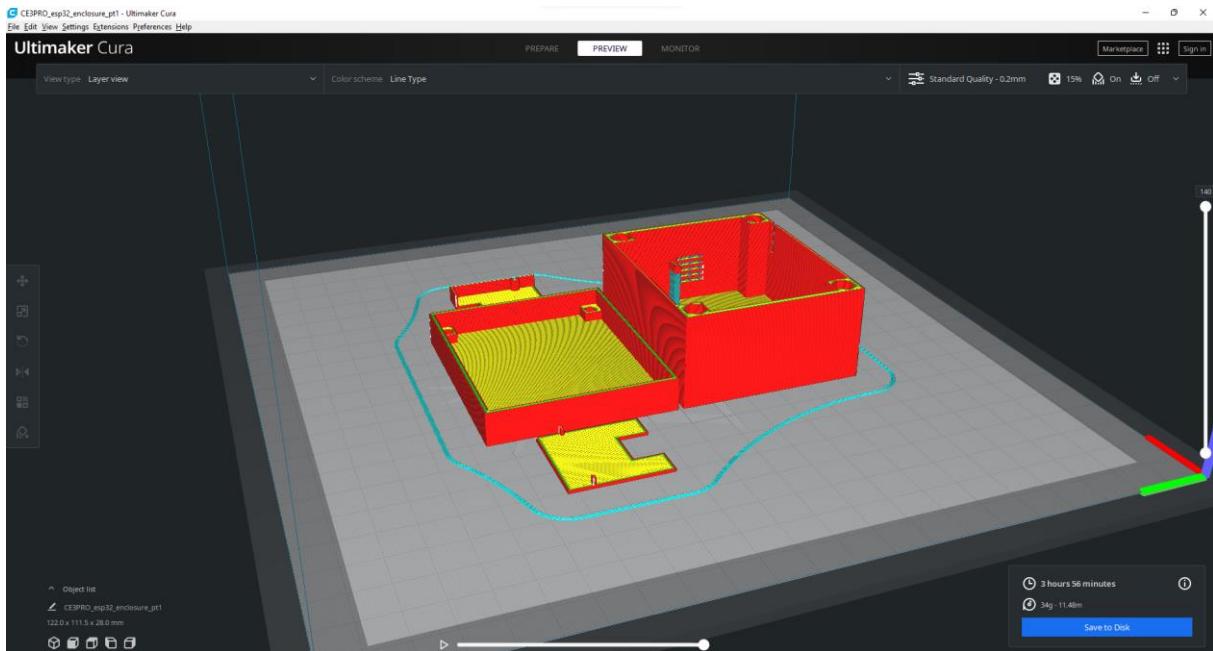
*Rys. 5.11. Render obudowy czujnika SHT30 oraz mikrokontrolera.*



*Rys. 5.12. Render obudowy pojemnościowego czujnika wilgotności gleby.*

Wyeksportowane modele stworzone w Fusion360 musiały zostać przystosowane do druku w tym celu wykorzystane zostało oprogramowanie Cura – jest to tzw. „Slicer”, czyli program dzielący model na warstwy i generujący zrozumiałe dla drukarki 3D gcode.

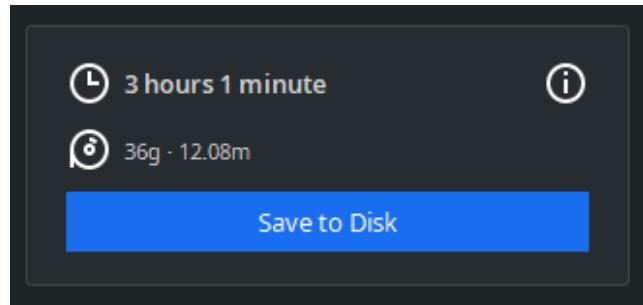
Na Rys.5.13. pokazany został pocięty na warstwy, gotowy do druku model



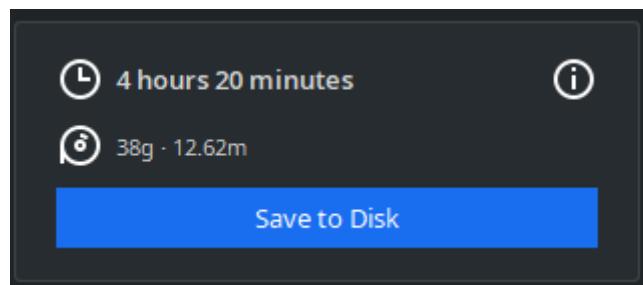
Rys. 5.13. Modele pocięte na warstwy – przygotowanie do druku.

Wydruki zostały wykonane w technologii FDM na drukarce Ender 3 Pro. Zastosowany materiał to PLA. Skrót PLA oznacza poliaktyd, czyli kwas polimlekowy. Jest to polimer otrzymywany z surowców roślinnych, a precyzyjniej rzec ujmując z sfermentowanej skrobi. Zastosowane tworzywo jest biodegradowalne, zatem nie sprawdzi się w omawianym zastosowaniu, jednak ze względu na niską cenę i łatwość druku idealnie sprawdza się w prototypach [11].

Dojście do ostatecznej wersji obudowy wymagało kilku iteracji procesu projektowania. Początkowe wydruki dla zaoszczędzenia czasu i materiału były drukowane przy niskim wypełnieniu modelu (10%), prędkości 100 mm/s i na stosunkowo dużej wysokości warstwy (0.28mm). Pozwoliło to na przyspieszenie druku o około 30% w stosunku do końcowej wersji, która została wykonana na 0.2 mm wysokości warstwy, 60% wypełnieniu i prędkości równej 70 mm/s.



Rys. 5.14. Estymowany przez program czas wydruku dla obniżonej jakości i wypełnienia.



Rys. 5.15. Estymowany przez program czas wydruku dla wersji końcowej.

Iteracje obudów zostały przedstawione na Rys. 5.16. (czujnik wilgotności gleby) i Rys. 5.17. (mikrokontroler wraz z czujnikiem wilgotności i temperatury powietrza oraz gniazdem zasilania).

Wersja „a” obudów powstała przed otrzymaniem obwodów drukowanych i czujników. Stanowiła model poglądowy tego jak może wyglądać złożone urządzenie pomiarowe.

Obudowa urządzenia pomiarowego była gotowa w iteracji „b”. Została wyposażona we wszystkie niezbędne przepusty na kable i gniazdo zasilania. Został dołożony uchwyt czujnika SHT30. Wyprofilowano wpusty pod śruby oraz zostało dodane miejsce na lepsze ułożenie przewodów. Model został zoptymalizowany pod względem ilości użytego materiału poprzez zredukowanie zbyt dużej grubości ścianek.

W iteracji „c” obudowa czujnika wilgotności gleby została poprawiona z uwzględnieniem rzeczywistych wymiarów (parametry podane na stronie dystrybutora były niedokładne).



Rys. 5.16. Iteracje prototypów obudowy czujnika wilgotności gleby.



Rys. 5.17. Iteracje prototypów obudowy czujnika SHT30 i mikrokontrolera.

Wydrukowane obudowy dobrze zabezpieczają prototypy na etapie testowania koncepcji projektu, jednak wersja, która miałaby działać w realnych warunkach powinna być dodatkowo zabezpieczona przed działaniem wilgoci i kurzu oraz wykonana z materiału, który nie będzie ulegał biodegradacji w wyniku działania czynników atmosferycznych. Dobrym wyborem mógłby być ABS oraz ulepszenie aktualnych modeli o wpusty przeznaczone na uszczelki.

Przedstawiona na Rys. 5.18. i Rys. 5.19. obudowa mikrokontrolera oraz czujnika SHT30 posiada zintegrowaną kratkę, która umożliwia działanie sensora, przepust na przewody czujnika wilgotności gleby oraz miejsce na gniazdo zasilania.

Obudowa czujnika wilgotności gleby przedstawiona na Rys. 5.20. i Rys. 5.21. zabezpiecza jedynie przed zwarciem wywołanym przewodzącym ciałem obcym. Zabezpieczenie to jest niezwykle ważne, ponieważ nie chroni jedynie sensora, ale również mikrokontroler.



Rys. 5.18. Złożona obudowa czujnika SHT30 i mikrokontrolera.



Rys. 5.19. Złożona obudowa czujnika SHT30 i mikrokontrolera.

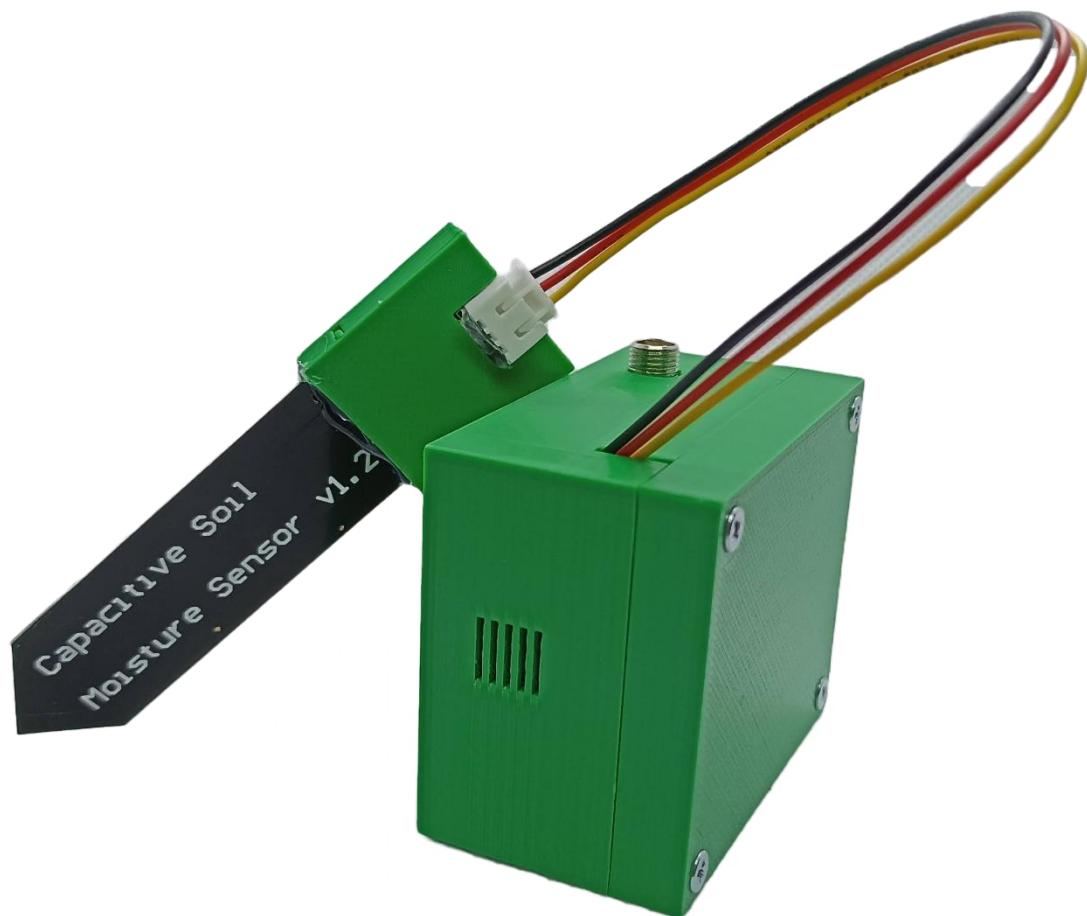


Rys. 5.20. Złożona obudowa czujnika wilgotności gleby.



Rys. 5.21. Złożona obudowa czujnika wilgotności gleby.

Obudowy (Rys. 5.22.) nie zostały ze sobą zintegrowane, aby dać możliwość przeprowadzenia pomiarów czujnikiem SHT30 na wysokości innej, niż poziom gleby oraz w przypadku obiektów wyposażonych w zraszacze, dać możliwość zabezpieczenia mikrokontrolera przed nadmierną wilgocią poprzez zamontowanie urządzenia ponad poziomem zraszania.



Rys. 5.22. Urządzenie pomiarowe zabezpieczone obudowami.

## **5.3. Oprogramowanie**

### **5.3.1. Komunikacja na linii mikrokontroler – serwer**

Komunikacja urządzenia pomiarowego z serwerem, którego funkcję pełni komputer osobisty z uruchomionym oprogramowaniem opisywanym w sekcji 5.3.3. odbywa się przez Wi-Fi z wykorzystaniem protokołu HTTP (ang. Hypertext Transfer Protocol), który służy do przesyłania dokumentów hipertekstowych za pośrednictwem sieci WWW. Protokół ten pozwala na interakcje z serwerem z wykorzystaniem ustandaryzowanych metod [8].

Metody wykorzystywane w komunikacji na linii urządzenie pomiarowe – serwer to POST oraz GET [8].

Dane przesyłane są w formacie JSON ze względu na jego czytelność, łatwą obsługę w frameworku Arduino oraz prostą konwersję do słowników języka Python w części serwerowej.

Alternatywnym podejściem do komunikacji w projekcie mogłoby być zastosowanie MQTT (Message Queue Telemetry Transport). Jest to protokół stworzony do transmisji danych między urządzeniami MQTT i działa w oparciu o wzorzec publikacji-subskrypcji, co oznacza, że urządzenie może publikować wiadomości na określonych kanałach, a odbieranie wiadomości odbywa się poprzez subskrybowanie wspomnianych kanałów.

Decyzja o wybraniu protokołu HTTP zamiast MQTT, który wydaje się lepszym wyborem w opisywanym zastosowaniu była podyktowana chęcią rozwinięcia nieszablonowego podejścia do stworzenia komunikacji na linii mikrokontroler – serwer.

### **5.3.2. Oprogramowanie mikrokontrolera**

#### **5.3.2.1 Wprowadzenie**

Oprogramowanie mikrokontrolera ma spełniać trzy podstawowe funkcje:

- Utrzymanie stabilnego połączenia z siecią Wi-Fi.
- Odczyt i przetwarzanie informacji z podłączonych czujników.
- Komunikacja z serwerem.

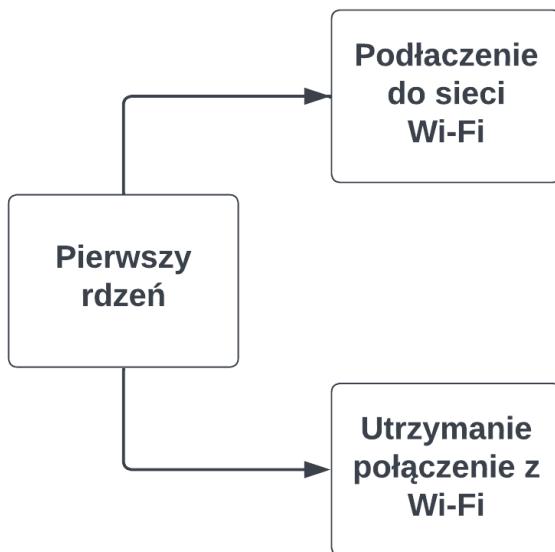
Platformą, na której zaimplementowane zostały powyższe funkcjonalności jest wspomniany w sekcji 5.2.1 mikrokontroler ESP32. Wbudowany moduł sieci bezprzewodowej

pozwolił na zrezygnowanie z zewnętrznego modułu Wi-Fi oraz zegara czasu rzeczywistego. Ponadto mikroprocesor Tensilica Xtensa LX6 pozwolił na odseparowanie priorytetowej funkcjonalności utrzymania połączenia z siecią bezprzewodową od reszty.

Obsługa połączenia z siecią bezprzewodową jest krytyczna z punktu widzenia działania systemu, dlatego została fizycznie odseparowana od pozostałych procedur poprzez zarezerwowanie dla niej oddzielnego rdzenia. Zarys tego, za jakie zadania odpowiadają poszczególne rdzenie przedstawiony został na Rys. 5.23. i Rys. 5.24.

Oprogramowanie urządzenia pomiarowego było tworzone z wykorzystaniem środowiska Arduino IDE 2.0 ze względu na wbudowane rozszerzenie do prostego pobierania potrzebnych bibliotek, wbudowany kompilator, monitor portu szeregowego oraz przede wszystkim narzędzie do wgrywania skompilowanego kodu bezpośrednio na zastosowaną płytę.

Język w jakim zostało napisane oprogramowanie to pochodna języka C/C++ wykorzystywana głównie w programowaniu płyt rozwojowych Arduino. Wykorzystanie tego framework'a było możliwe, ponieważ poza mikroprocesorami firmy Atmel (wykorzystywanymi w Arduino) wspiera on również inne chipy między innymi te produkowane przez Espressif [5].



Rys. 5.23. Schemat blokowy oprogramowania pierwszego rdzenia.



Rys. 5.24. Schemat blokowy oprogramowania drugiego rdzenia.

### 5.3.2.2. Algorytmy i omówienie działania programu

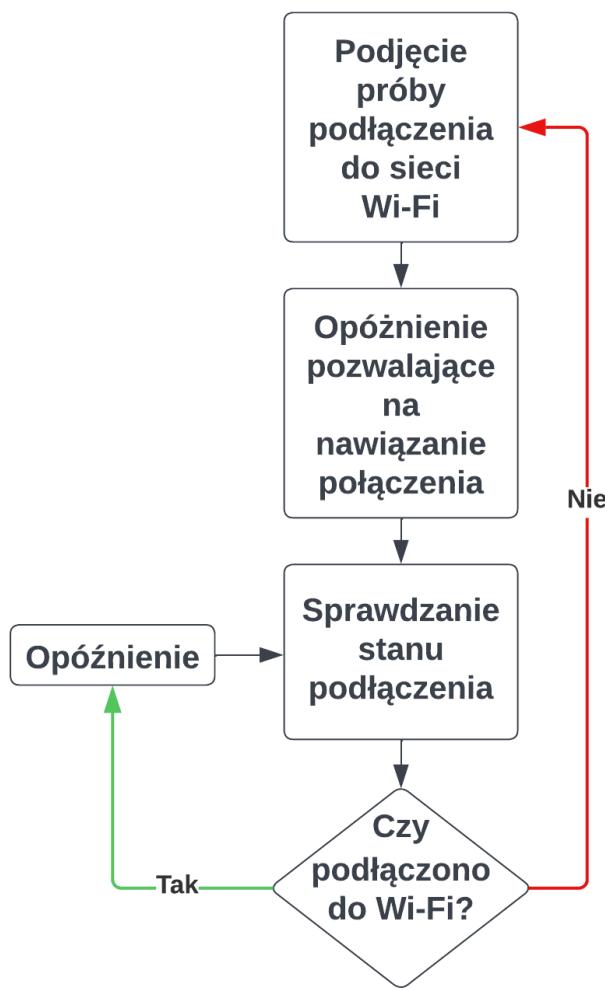
Aby zapewnić stabilną i pewną pracę programu wykorzystany został FreeRTOS – system operacyjny czasu rzeczywistego dla zastosowań embeded. Pozwala on na programowanie wielowątkowe oraz zarządzanie cechami wątków takimi jak m.in. priorytet wykonywania i przypisanie do wskazanego rdzenia procesora.

W omawianym przypadku dla każdej z wymienionych w sekcji 5.3.2.1 funkcjonalności został utworzony dedykowany wątek (task). Przypisanie procedur do poszczególnych rdzeni procesora zostało przedstawione na listingu 1. Obsługa połączenia z siecią bezprzewodową zaimplementowana jako wątek wifiTask została odseparowana od reszty programu i przypisana została do pierwszego rdzenia procesora ((0 jako ostatni argument funkcji xTaskCreatePinnedToCore)). Powoduje to, że jej działanie nie zostanie w żaden sposób zakłócone przy wykonywaniu pozostałych funkcji. Drugi rdzeń odpowiada za obsługę czujników i komunikację z serwerem (wątki httpTask, sensorTask, sensorOperatorTask).

*Listing 1. Tworzenie tasków.*

```
1 xTaskCreatePinnedToCore(wifiTaskCode,
2                         "wifi task"
3                         10000,
4                         NULL,
5                         2,
6                         &wifiTask,
7                         0);
8 delay(500);
9
10 xTaskCreatePinnedToCore(httpHandlerTask,
11                         "http_task",
12                         5000,
13                         NULL,
14                         1,
15                         &httpTask,
16                         1);
17 delay(500);
18
19 xTaskCreatePinnedToCore(sensorDataCollector,
20                         "sensorDataCollector",
21                         10000,
22                         NULL,
23                         1,
24                         &sensorTask,
25                         1);
26 delay(500);
27
28 xTaskCreatePinnedToCore(sensorOperator,
29                         "sensorOperator",
30                         5000,
31                         NULL,
32                         1,
33                         &sensorOperatorTask,
34                         1);
35 delay(500);
```

Stabilne działanie funkcji odpowiadającej za nawiązanie komunikacji bezprzewodowej jest priorytetowe, ponieważ stworzona procedura odpowiada również za ponowne połączenie w przypadku zerwania połączenia z siecią Wi-Fi. Jej struktura działania została przedstawiona na Rys. 5.25. a kod na listingu 2.



Rys. 5.25. Algorytm procedur pierwszego rdzenia.

Przy starcie urządzenia podejmowana jest próba podłączenia do Wi-Fi, jeśli się powiedzie procedura wchodzi w stan monitorowania stanu połączenia w stałych interwałach. W przypadku zerwania komunikacji z siecią bezprzewodową następuje próba ponownego połączenia. Urządzenie pozostaje w tym stanie, dopóki nie zostanie przywrócone połączenie, następnie wraca do trybu monitorowania połączenia.

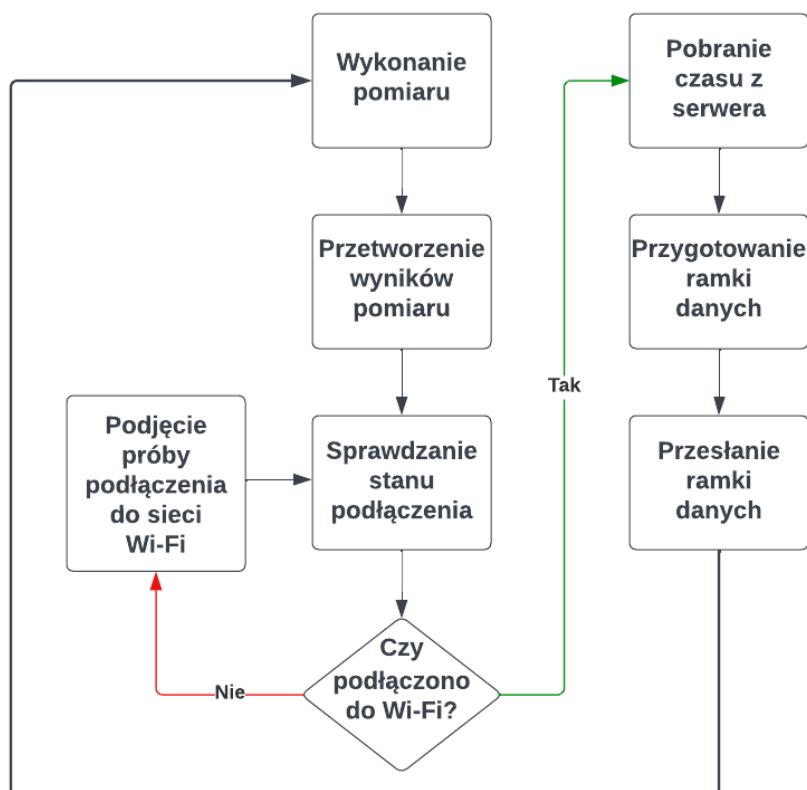
Listing 2. Kod procedury obsługującej połączenie z Wi-Fi.

```
1 void wifiTaskCode(void *parameter)
2 {
3     Serial.print("wifi task is running on core ");
4     Serial.println(xPortGetCoreID());
5     WiFi.mode(WIFI_STA);
6     WiFi.begin(ssid, password);
7     delay(10000);
8     for (;;)
9     {
10         digitalWrite(led_1, HIGH);
11         if (WiFi.status() == WL_CONNECTED)
12         {
13             Serial.println("connected");
14             delay(5000);
15         }
16         else
17         {
18             Serial.print("reconnecting\n");
19             WiFi.mode(WIFI_STA);
20             WiFi.begin(ssid, password);
21             delay(5000);
22         }
23         digitalWrite(led_1, LOW);
24         delay(5000);
25     }
26 }
```

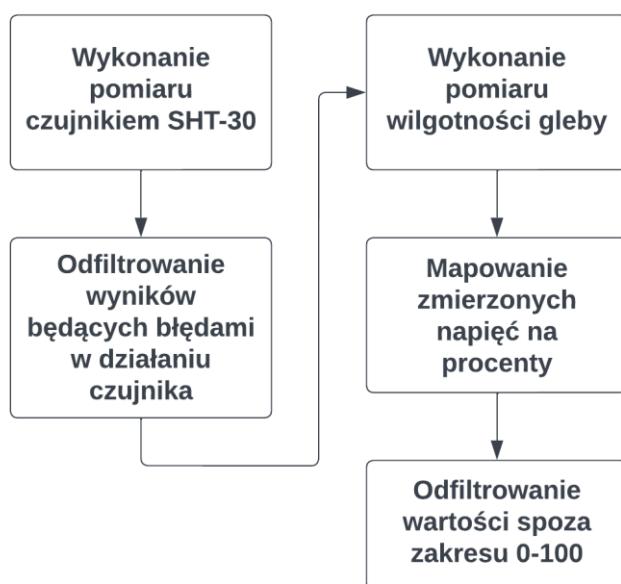
Drugi rdzeń obsługuje przedstawione na schemacie blokowym na Rys. 5.26 procedury. Wykonują się one cyklicznie i ich działanie jest zależne od funkcjonalności zaimplementowanej w pierwszym rdzeniu. W przypadku braku połączenia z siecią bezprzewodową nie zostaje podjęta próba przesłania ramki danych do serwera.

Pomiarystwo prowadzone są w przedstawionej w listingu 3 funkcji. Zbiera ona wyniki z czujnika SHT30 oraz odczytuje i konwertuje na wartości procentowe odczyty z pojemnościowego czujnika wilgotności gleby poprzez funkcję map, która mapuje oszacowany w sekcji 5.2.2 zakres pomiarowy na liczby z przedziału 0-100. Dane te są wykorzystywane w wątku odpowiadającym za przesyłanie danych do serwera.

Na poziomie funkcji sensorOperator zostały wprowadzone zabezpieczenia przed przesyłaniem danych wynikających z błędów w działaniu czujników. Dla pomiarów parametrów powietrza zastosowana została funkcja isnan, która sprawdza, czy argument jest liczbą. W przypadku wilgotności gleby rolę zabezpieczenia pełni instrukcja warunkowa, która odfiltrowuje wyniki spoza oszacowanego zakresu.



Rys. 5.26. Algorytm procedur działających na drugim rdzeniu.

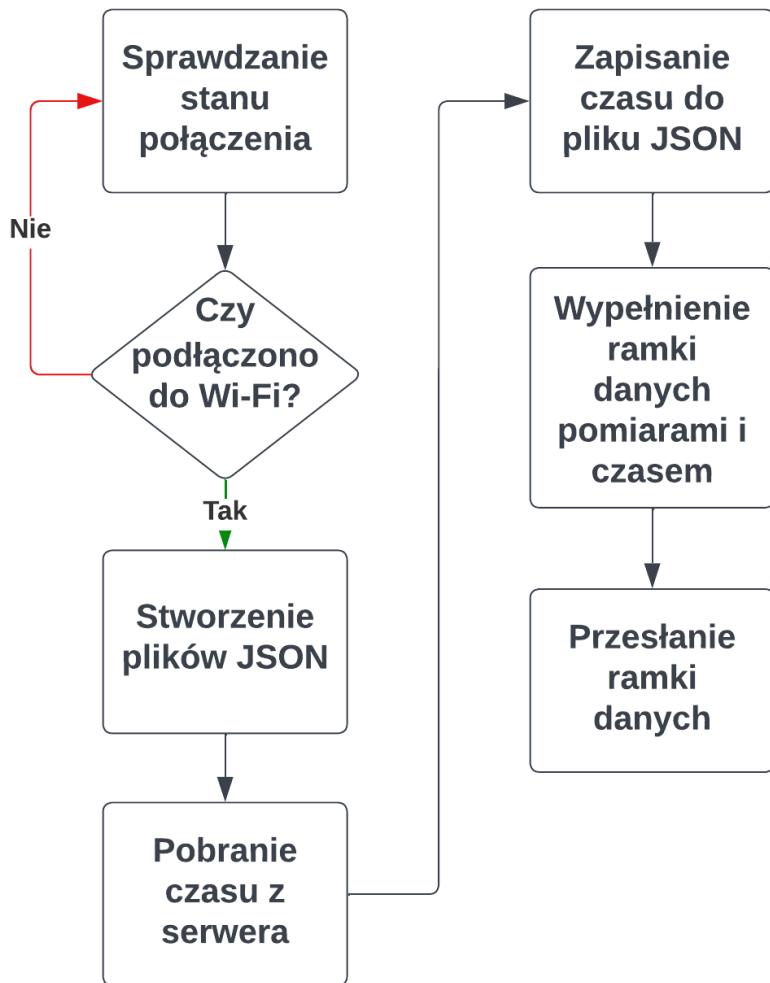


Rys. 5.27. Algorytm procedury pomiarowej.

Listing 3. Kod procedury pomiarowej.

```
1 void sensorOperator(void *parameter)
2 {
3
4     for (;;)
5     {
6         if (WiFi.status() == WL_CONNECTED)
7         {
8             double temperatureReading = sht31.readTemperature();
9             double humidityReading = sht31.readHumidity();
10
11            if (!isnan(temperatureReading))
12            {
13                temperature = temperatureReading;
14                prevTemperature = temperatureReading;
15            }
16            else
17            {
18                temperature = prevTemperature;
19            }
20
21            if (!isnan(humidityReading))
22            {
23                humidity = humidityReading;
24                prevHumidity = humidityReading;
25            }
26            else
27            {
28                humidity = prevHumidity;
29            }
30
31            double soilMoistureReading = analogRead(AOUT_PIN);
32            soilMoistureReading = map(soilMoistureReading, 100, 600, 10000, 0);
33            soilMoistureReading = soilMoistureReading / 100;
34            if (0 < soilMoistureReading && soilMoistureReading < 100)
35            {
36                soilMoisture = soilMoistureReading;
37                prevMoisture = soilMoistureReading;
38            }
39            else
40            {
41                soilMoisture = prevMoisture;
42            }
43
44            delay(1000);
45        }
46    }
47 }
```

Komunikacja z serwerem, której działanie zostało przedstawione na schemacie blokowym na Rys. 5.28., została zaimplementowana w dwóch funkcjach: sensorDataCollector oraz httpHandlerTask



Rys.5.28. Algorytm procedur komunikacyjnych.

Funkcja sensorDataCollector, której kod jest na listingu 4 odpowiada za przygotowanie ramki danych w formacie JSON. Wysyłana ramka danych zamiera datę i godzinę wykonanego pomiaru (timestamp) oraz wyniki pomiarów z sensorów. Stosowany w ramce danych czas pobierany jest z serwera przy pomocy wysłania zapytania metodą GET. Zastosowanie takiego rozwiązania pozwoliło na zrezygnowanie z fizycznego zegara czasu rzeczywistego, który wymagałby dodatkowego, zewnętrznego zasilania baterijnego oraz

ułatwiało proces synchronizacji czasu rzeczywistego z czasem urządzenia, ponieważ w takiej konfiguracji dotyczy tylko serwera.

*Listing 4 Kod procedury obsługującej tworzenie plików JSON.*

```
1 void sensorDataCollector(void *parameter)
2 {
3     StaticJsonDocument<1024> doc;
4     StaticJsonDocument<256> docTime;
5     for (;;)
6     {
7         if (WiFi.status() == WL_CONNECTED)
8             {dataFrame.clear();
9              doc.clear();
10             docTime.clear();
11             getTime.clear();
12             HTTPClient http;
13             http.begin("http://10.5.101.7:5000/get-datetime");
14             http.GET();
15             getTime = http.getString();
16             http.end();
17
18             deserializeJson(docTime, getTime);
19             Serial.println(getTime);
20             if (!docTime["date"].isNull() || !docTime["time"].isNull())
21             {
22                 doc["sensor_id"] = sensorID;
23                 doc["air_temperature"] = temperature;
24                 doc["air_humidity"] = humidity;
25                 doc["soil_moisture"] = soilMoisture;
26                 doc["date"] = docTime["date"];
27                 doc["time"] = docTime["time"];
28
29                 serializeJson(doc, dataFrame);
30                 Serial.println(dataFrame);
31             }
32         }
33
34         delay(9000);
35     }
36 }
```

W pamięci urządzenia pomiarowego ramka danych przechowywana jest jako dokument o stałym rozmiarze. Jego wielkość jest sztywno określona i bierze pod uwagę rozmiar przesyłanych danych oraz margines bezpieczeństwa. Możliwe było zaimplementowanie dynamicznego przypisywania wielkości pliku JSON, jednak

predefiniowany rozmiar eliminuje błędy, które mogłyby wystąpić przy błędach w działaniu algorytmu definiującego wielkość pliku.

Listing 5. Kod procedury odsługującej przesyłanie ramki danych.

```
1 void httpHandlerTask(void *parameter)
2 {
3     for (;;)
4     {
5         digitalWrite(led_2, HIGH);
6         if (WiFi.status() == WL_CONNECTED)
7         {
8             HTTPClient http;
9
10            http.begin("http://10.5.101.7:5000/data-collector");
11
12            http.addHeader("Content-Type", "text/plain");
13            http.POST(dataFrame);
14            http.end();
15            delay(10000);
16        }
17
18        digitalWrite(led_2, LOW);
19        delay(100);
20    }
21 }
```

```
10.9.46.215 - - [01/Apr/2023 20:46:41] "GET /get-datetime HTTP/1.1" 200 -
b'{"sensor_id":1,"air_temperature":22.23999977,"air_humidity":56.06999969,"soil_moisture":20.8,"date":"01-04-2023","time":"20:46:41"}'
10.9.46.215 - - [01/Apr/2023 20:46:45] "POST /data-collector HTTP/1.1" 200 -
10.9.46.215 - - [01/Apr/2023 20:46:51] "GET /get-datetime HTTP/1.1" 200 -
b'{"sensor_id":1,"air_temperature":22.23999977,"air_humidity":55.97999954,"soil_moisture":9.4,"date":"01-04-2023","time":"20:46:51"}'
10.9.46.215 - - [01/Apr/2023 20:46:55] "POST /data-collector HTTP/1.1" 200 -
10.9.46.215 - - [01/Apr/2023 20:47:00] "GET /get-datetime HTTP/1.1" 200 -
b'{"sensor_id":1,"air_temperature":22.25,"air_humidity":56.06999969,"soil_moisture":26.8,"date":"01-04-2023","time":"20:47:00"}'
10.9.46.215 - - [01/Apr/2023 20:47:06] "POST /data-collector HTTP/1.1" 200 -
10.9.46.215 - - [01/Apr/2023 20:47:10] "GET /get-datetime HTTP/1.1" 200 -
b'{"sensor_id":1,"air_temperature":22.28000069,"air_humidity":56.08000183,"soil_moisture":20.4,"date":"01-04-2023","time":"20:47:10"}'
10.9.46.215 - - [01/Apr/2023 20:47:17] "POST /data-collector HTTP/1.1" 200 -
10.9.46.215 - - [01/Apr/2023 20:47:19] "GET /get-datetime HTTP/1.1" 200 -
b'{"sensor_id":1,"air_temperature":22.29000092,"air_humidity":56.06999969,"soil_moisture":10.8,"date":"01-04-2023","time":"20:47:19"}'
10.9.46.215 - - [01/Apr/2023 20:47:27] "POST /data-collector HTTP/1.1" 200 -
```

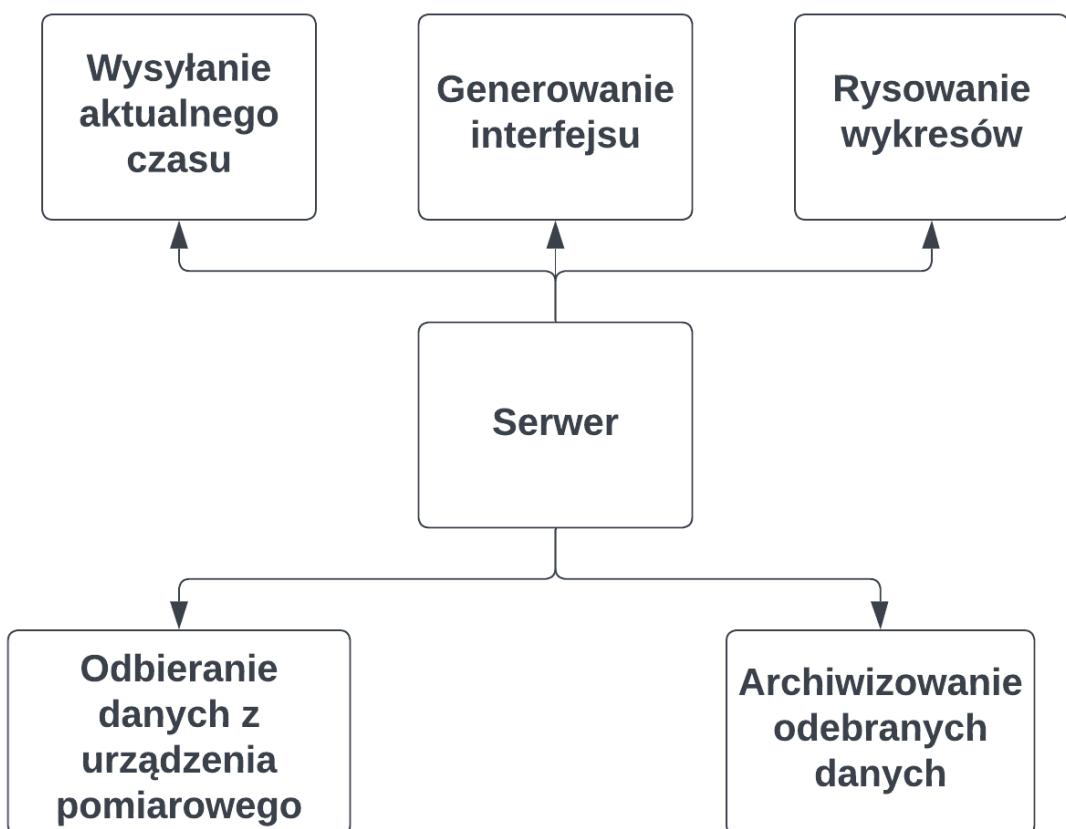
Rys. 5.29. Dane wysyłane przez urządzenie pomiarowe.

### **5.3.3. Oprogramowanie serwerowe**

#### **5.3.3.1 Wprowadzenie**

Oprogramowanie serwerowe zgodnie z założeniami projektu zawiera poniższe funkcjonalności:

- Komunikacja bezprzewodowa z dowolną ilością urządzeń pomiarowymi
- Gromadzenie odczytów z dowolnej ilości urządzeń
- Graficzne przedstawianie najnowszych pomiarów
- Graficzne przedstawienie wszystkich pomiarów z dowolnej ilości urządzeń.

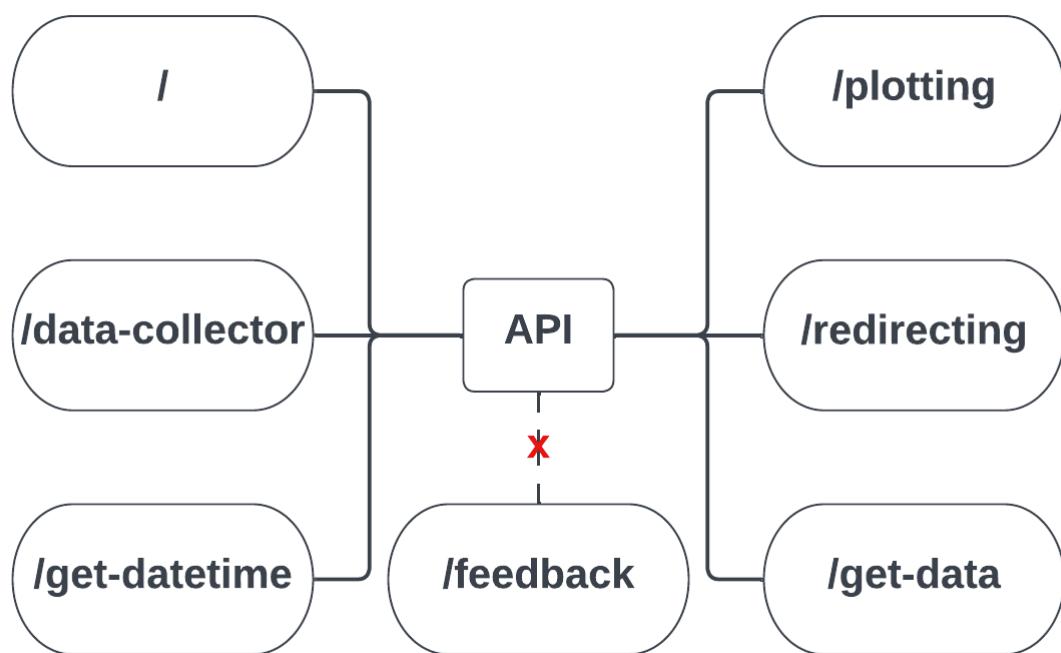


Rys. 5.30. Schemat blokowy oprogramowania serwerowego.

Oprogramowanie serwerowe pełni kluczową rolę w systemie, umożliwia przetwarzanie i analizowanie danych z dowolnej ilości urządzeń pomiarowych podłączonych do sieci bezprzewodowej. Zostało zaimplementowane w postaci API, czyli interfejsu umożliwiającego komunikację pomiędzy aplikacjami i serwisami internetowymi

Program został napisany w języku Python z wykorzystaniem frameworka Flask, który jest lekką i prostą w obsłudze biblioteką pozwalającą tworzyć API do obsługi zapytań HTTP. Główną zaletą zastosowanej technologii jest możliwość szybkiego prototypowania kolejnych funkcjonalności oraz obszerna dokumentacja, która bardzo ułatwia rozwiązywanie skomplikowanych problemów. Ze względu na charakterystykę zastosowanego frameworka, napisany program jest otwarty na możliwość rozbudowy w przypadku chęci rozszerzenia projektu o np. urządzenia wykonawcze lub bardziej skomplikowane urządzenia pomiarowe [8].

API pozwala na komunikację przy pomocy endpointów, czyli miejsc, przez które można nawiązać połączenie z aplikacją. Ich struktura została zaprojektowana na podstawie schematu ideowego części serwerowej projektu.



Rys. 5.31. Graficzne przedstawienie struktury API

### 5.3.3.2 Omówienie elementów części serwerowej

Endpoint [/] odpowiedzialny jest za wyświetlanie interfejsu użytkownika (dashboard). Jego jedyną rolą jest renderowanie wygenerowanego przez aplikację serwerową pliku HTML przy wejściu na stronę główną projektu.

Listing 6. Kod obsługujący endpoint [/].

```
1 @app.route("/", methods=['GET', 'POST'])
2 def index():
3     return render_template('index.html')
```

Endpoint [/data-collector] odpowiada za odbieranie ramki danych z urządzenia pomiarowego – z tego powodu obsługuje jedynie metodę POST. Obsługuje go funkcja *collector()*. W funkcji została wykorzystana konstrukcja *Try Except*, której zadaniem jest wyłapanie błędów związanych z dekodowaniem danych wejściowych i ewentualne przerwanie procedury.

Listing 7. Kod obsługujący endpoint [/data-collector].

```
1 @app.route("/data-collector", methods=['POST'])
2 def collector():
3     try:
4         data = DataOperator(request.data)
5         data.csv_dump()
6         output_generator.update_records(data.sensor_id,
7                                         data.values,
8                                         data.timestamp)
9         output_generator.update_states()
10        generator = BoxGenerator(output_generator.return_output(),
11                               output_generator.return_timestamps())
12        generator.html_dump()
13        print("data collected")
14    except ValueError:
15        print("Decoding failed")
16
17    return "data-collector"
```

Za przetworzenie odebranej ramki danych oraz jej zapisanie w pliku CSV odpowiada klasa DataOperator. Podczas tworzenia jej instancji ramka danych jest automatycznie konwertowana na słownik języka Python. Następnie wywoływana jest metoda *csv\_dump()*, która zapisuje odebrane i przetworzone dane do pliku CSV. Struktura pliku CSV, który jest stosowany w projekcie została przedstawiona na Rys. 5.32.

Listing 8. Klasa DataOperator.

```
1  class DataOperator:
2      def __init__(self, input_dictionary) -> None:
3
4          self.input_dict = input_dictionary
5          self._load_json()
6          self.values = []
7          self.timestamp = []
8          self.sensor_id = 0
9
10     def _load_json(self) -> None:
11         '''decode json input into dictionary'''
12         d = json.loads(self.input_dict)
13         self.input_dict = d
14
15     def _parse_data(self) -> None:
16         '''split json sections'''
17         self.values = list(self.input_dict.values())[1:4]
18         self.timestamp = list(self.input_dict.values())[4:6]
19         self.sensor_id = list(self.input_dict.values())[0]
20
21     def csv_dump(self) -> None:
22         '''store incoming json data in separated csv files for every sensor'''
23         self._parse_data()
24         d = self.input_dict
25         with open(f'backend/dataSensor{d["sensor_id"]}.csv', 'a', newline='') as csvfile:
26             writer = csv.writer(csvfile)
27             writer.writerow(list(d.values()))
```

● ● ● eng-greenhouse-automation-system - dataSensor1.csv

```
1  1,25.57999992,41.61999893,26.8,26-03-2023,16:35:08
2  1,25.57999992,41.63999939,15,26-03-2023,16:35:27
3  1,25.56999969,41.63000107,14.4,26-03-2023,16:35:37
4  1,25.60000038,41.59999847,20.6,26-03-2023,16:35:46
5  1,25.57999992,41.65999985,25.2,26-03-2023,16:35:56
6  1,25.56999969,41.63000107,20,26-03-2023,16:36:05
7  1,25.56999969,41.70000076,14.4,26-03-2023,16:36:14
8  1,25.60000038,41.74000168,27.2,26-03-2023,16:36:24
9  1,25.57999992,41.74000168,25,26-03-2023,16:36:33
10 1,25.60000038,41.79999924,28.6,26-03-2023,16:36:52
```

Rys. 5.32. Przykładowy plik CSV wygenerowany w trakcie działania programu.

Zaletą stosowania formatu CSV jest możliwość otworzenia historii odczytów w arkuszu kalkulacyjnym, bez konieczności dodatkowej konwersji. Wadą może być rosnący czas dostępu do próbek wraz z wzrostem ich ilości.

Aby sprawdzić realny wpływ wspomnianego problemu na działanie projektu został przeprowadzony prosty test mający na celu sprawdzenie jak logarytmiczny przyrost ilości próbek wpływa na czas odczytu. Procedura testowa polegała na sprawdzeniu różnicy pomiędzy próbami czasu przed i po rozpoczęciu działania funkcji przetwarzającej plik CSV.

*Listing 9. Procedura testowania czasu egzekucji kodu.*

```
1 def check_time(body):
2     def wrapper(*arg, **kw):
3         start_time = time.time()
4         print(f"Time of parsing .csv file with {body(*arg, **kw)}"
5               f"/records is: {time.time()-start_time}\n")
6     return wrapper

1 Time of parsing .csv file with 10 records is: 0.9353170394897461
2 Time of parsing .csv file with 100 records is: 0.08031821250915527
3 Time of parsing .csv file with 1_000 records is: 0.08814573287963867
4 Time of parsing .csv file with 10_000 records is: 0.17160987854003906
5 Time of parsing .csv file with 100_000 records is: 0.9520454406738281
6 Time of parsing .csv file with 1_000_000 records is: 8.688284397125244
```

*Rys. 5.33. Wynik działania procedury testującej czas egzekucji.*

Na podstawie raportu działania funkcji można zauważyć, że do 100000 próbek, czas odczytu utrzymywany jest poniżej sekundy, co skutkuje stosunkowo szybkim dostępem do interfejsu odpowiadającego za rysowanie wykresów. Problem pojawi się przy ilości próbek wynoszącej 1000000. Czas przetwarzania pliku jest bliski 9 sekund, jest to wartość poważnie wpływająca na responsywność funkcjonalności.

Przyjmując założenie, że system ma działać na tyle długo, żeby miał możliwość zebrania ilości próbek w granicy miliona, potrzebny byłby inny, szybszy w działaniu sposób

przechowywania danych, jednak w aspekcie prototypowym osiągnięte czasy przetwarzania są wystarczające.

Funkcja *collector* określająca działanie endpointa wywołuje następnie funkcję *update\_records*, która jest metodą klasy *OutputGenerator*. Odpowiada ona za wygenerowanie słownika zawierającego najnowsze dane odebrane z urządzeń pomiarowych i stanem urządzenia (domyślnie stan określający urządzenie jako aktywne) oraz słownika z znacznikami czasowymi ostatnich pomiarów.

Metoda *update\_states* aktualizuje stany urządzeń pomiarowych. Stan definiowany jest na podstawie różnicy pomiędzy aktualnym czasem, a czasem ze znacznika w ramce danych.

*Listing 10. Klasa OutputGenerator.*

```
1  class OutputGenerator(TimeOperator):
2      def __init__(self) -> None:
3          super().__init__()
4          self.latest_records = {}
5          self.latest_timestamps = {}
6          self.timestamp = ''
7
8      def decode_timestamp(self, timestamp_input: str) -> None:
9          '''dump data from timestamp into string'''
10         self.timestamp = "{} {}".format(*timestamp_input)
11
12
13     def update_records(self, id: str|int, values: list, timestamp: list) -> None:
14         '''create and update lists of latest incoming values'''
15         self.latest_records[id] = [1, [*values]]
16         self.latest_timestamps[id] = [*timestamp]
17         print(self.latest_timestamps)
18
19     def update_states(self) -> None:
20         '''calculate states of sensors (active/inactive) based on delta time'''
21         for key in self.latest_timestamps:
22             self.decode_timestamp(self.latest_timestamps[key])
23             if self.delta_time(self.timestamp) > 1:
24                 self.latest_records[key][0] = 0
25
26     def return_output(self) -> dict:
27         '''return list of latest records'''
28         print(self.latest_records)
29         return self.latest_records
30
31     def return_timestamps(self) -> str|dict:
32         '''return list of latest records timestamps'''
33         return self.latest_timestamps
```

Endpoint [/get-datetime] publikuje aktualny czas. Celem jego wprowadzenia jest wyeliminowanie konieczności stosowania fizycznego zegara czasu rzeczywistego w urządzeniu pomiarowym, zamiast tego, czas potrzebny do uzupełnienia znacznika czasu pobierany jest z serwera.

*Listing 11. Kod obsługujący endpoint [/get-datetime].*

```
1 @app.route("/get-datetime", methods=['GET'])
2 def date_time():
3     return TimeOperator().send_datetime()
```

Zastosowana metoda *send\_datetime* będąca elementem klasy *TimeOperator* generuje ramkę danych ograniczoną jedynie do czasu. Przesyłane dane są konwertowane do formatu JSON podczas pobierania przez urządzenie pomiarowe.

*Listing 12. Wybrane metody klasy TimeOperator.*

```
1 def get_datetime(self):
2     '''return timestamp based on actual time'''
3     now = datetime.now()
4     date_list = [now.day, now.month, now.year]
5     time_list = [now.hour, now.minute, now.second]
6     return date_list, time_list
7
8 def send_datetime(self):
9     '''return dictionary containing timestamp'''
10    return {"date": "{:02d}-{:02d}-{}".format(*self.get_datetime()[0]),
11            "time": "{:02d}:{:02d}:{:02d}".format(*self.get_datetime()[1])}
```

Endpoint [/plotting] odpowiada za wygenerowanie wykresów dla wskazanego czujnika. Wskazanie czujnika odbywa się poprzez kliknięcie odpowiedniego przycisku w głównym interfejsie programu. Danymi wejściowymi dla wygenerowanego wykresu jest pełna historia pomiarów przechowywana w pliku CSV.

*Listing 13. Kod obsługujący endpoint [/plotting].*

```
1 @app.route("/plotting", methods = ['GET', 'POST'])
2 def plotter():
3     d = DataPlotter(memory.a).return_all_data()
4     return render_template('plotting.html',
5                           temperatureJSON=d[0],
6                           humidityJSON = d[1],
7                           moistureJSON = d[2])
```

Endpoint [/redirecting] obsługuje przełączanie się pomiędzy elementami interfejsu. W przypadku przejścia z głównego interfejsu do rysowania wykresów aktualizuje zmienną odpowiadającą za wybór czujnika. Jej wartość przesyłana jest do serwera poprzez plik HTML.

*Listing 14. Kod obsługujący endpoint [/redirecting].*

```
1 @app.route("/redirecting", methods = ['GET', 'POST'])
2 def redirecting_hub():
3     data = request.form.get("button")
4     if data == "go_back":
5         return redirect("/", code=302)
6     else:
7         memory.update_a(data)
8         return redirect("/plotting", code=302)
```

Do utrzymywania stanów pamięci i prostej wymiany danych pomiędzy funkcjami wprowadzona została klasa *DataStore*, która aktualnie posiada jedną zmienną – *a*, oraz jedną metodę *update\_a*, jednak w przypadku rozbudowania projektu może zostać łatwo rozszerzona, tak, aby pozwalała na dowolne manipulowanie elementami pamięci.

*Listing 15. Klasa DataStore.*

```
1 class DataStore:
2     def __init__(self):
3         self.a = None
4
5     def update_a(self, variable):
6         print(variable)
7         self.a = variable
```

Endpoint [/feedback] został zaprojektowany oraz zaimplementowany we wczesnej wersji projektu, jednak został usunięty ze względu na problemy, które generował podczas testów. Jego zadaniem było zwracanie odpowiedzi dla czujnika o powodzeniu/niepowodzeniu przesyłania ramki danych. Dane wyjściowe przekazywane były w formacie JSON. Implementacja takiego rozwiązania okazała się problematyczna przy dużej ilości urządzeń pomiarowych. Proponowane rozwiązanie działało w przypadku jednego, kilka urządzeń pomiarowych wymagałoby dynamicznego tworzenia endpointów przypisanych do konkretnych ID.

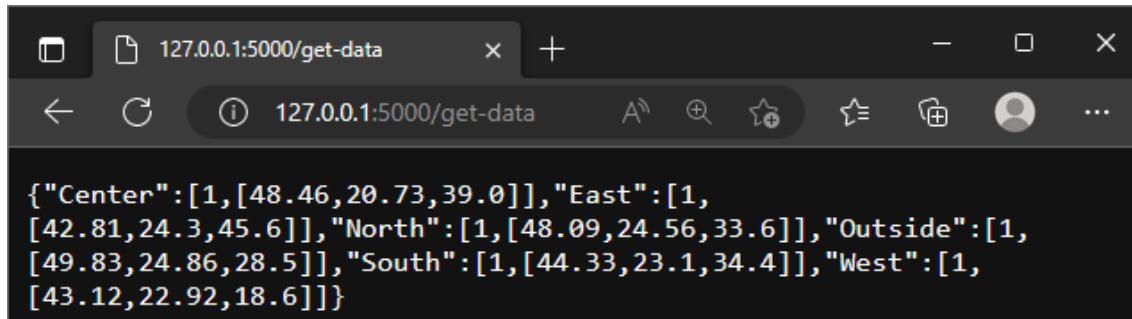
*Listing 16. Kod obsługujący endpoint [/feedback].*

```
1 @app.route("/feedback", methods = ['GET'])
2 def feedback():
3     feedback_dict = {"state":memory.b}
4     memory.update_b
5     return json.dumps(feedback_dict)
```

Endpoint [/get-data] zwraca słownik zawierający ostatnie odczyty. Pozwala on na przesłanie informacji np. do komputera klimatycznego, lub innej jednostki sprzęgniętej z urządzeniami wykonawczymi sterującymi warunkami mikroklimatu. Jego wyjście zostało przedstawione na Rys. 5.34.

Listing 17. Kod obsługujący endpoint [/get-data].

```
1 @app.route("/get-data", methods = ['GET', 'POST'])
2 def get_data():
3     return output_generator.return_output()
```



Rys. 5.34.. Dane wyjściowe z endpointa [/get-data].

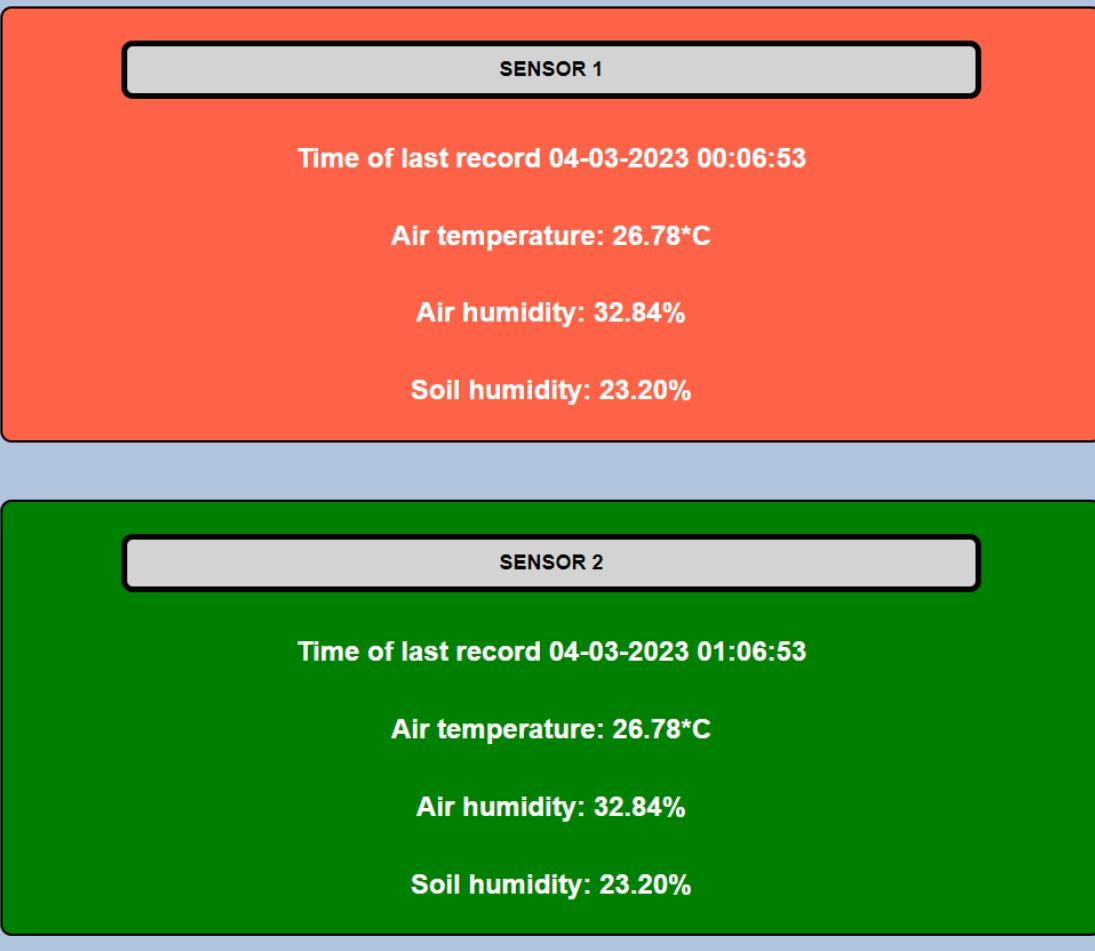
#### 5.3.4. Główny interfejs programu

Interfejs systemu dostępny jest z poziomu przeglądarki. Jest podzielony na dwie sekcje: listowanie czujników wraz z ich najnowszymi pomiarami oraz stanem (generowanym na podstawieznacznika czasu ostatnio odebranej ramki danych) oraz rysowanie wykresu temperatury, wilgotności powietrza oraz wilgotności gleby. Został zaimplementowany w języku angielskim, ponieważ jest uniwersalny, zrozumiały w większości rejonów świata, dodatkowo, zrezygnowanie ze znaków diakrytycznych występujących w języku polskim pozwoliło na uniknięcie błędów kodowania plików.

Interfejs listujący czujniki (dashboard) jest generowany dynamicznie przez kod z listingu 20. Ilość pól zależy od ilości czujników, z których system odebrał dane. Plik HTML składa się z nagłówka, który pozostaje niezmienny oraz dowolnej ilości wygenerowanych przez program pól z odpowiednio przypisana klasą informującą o tym, czy dany sensor jest aktywny.

Wygenerowany dla dwóch czujników kod przedstawiony jest na listingu 18. Efektem jego działania jest interfejs przedstawiony na Rys. 5.35. Wyświetla on wszystkie wymagane dane oraz informuje o stanie w jakim znajduje się czujnik (zielony – aktywny, czerwony – nieaktywny) i pozwala na przejście do części interfejsu, która rysuje wykresy poprzez kliknięcie przycisku z identyfikatorem sensora.

## DASHBOARD



Rys. 5.35. Główny interfejs programu.

Warunkiem przejścia sensora w stan nieaktywny jest nieprzesłanie żadnej ramki danych przez czas dłuższy niż godzina. Okres między pomiarami wyliczany jest w metodzie `delta_time` (listing 19.). Stany aktualizowane są w roucie [/data-collector] podczas wywołania metody `update_states`, czyli w momencie, gdy serwer otrzymuje jakikolwiek sygnał od urządzenia pomiarowego.

Listing 18 Wygenerowany kod HTML interfejsu.

```
1  <!DOCTYPE html>
2  <head>
3  <title>DASHBOARD</title>
4  <link rel="stylesheet"
5      href="{{url_for('static', filename='css/style.css')}}">
6  <meta http-equiv="refresh" content="60">
7  </head>
8  <body>
9  <h2>DASHBOARD <span id="dash-board"></span></h2>
10 <div class = "inactiveSensor"><br>
11     <form action = "/redirecting" method="POST">
12         <button type="submit" name="button" value="1">SENSOR 1</button>
13     </form>
14     <br><h2>Time of last record 04-03-2023 00:06:53</h2>
15     <h2>Air temperature: 26.78*C</h2>
16     <h2>Air humidity: 32.84%</h2>
17     <h2>Soil humidity: 23.20%</h2>
18     </div>
19 <div class = "activeSensor"><br>
20     <form action = "/redirecting" method="POST">
21         <button type="submit" name="button" value="1">SENSOR 2</button>
22     </form>
23     <br><h2>Time of last record 04-03-2023 01:06:53</h2>
24     <h2>Air temperature: 26.78*C</h2>
25     <h2>Air humidity: 32.84%</h2>
26     <h2>Soil humidity: 23.20%</h2>
27     </div>
28 </body>
29 </html>
```

Listing 19. Metoda delta\_time.

```
1  def delta_time(self, timestamp: str) -> float:
2      '''calculate and return delta of actual time and given timestamp'''
3      datetime_object = datetime.strptime(timestamp, '%d-%m-%Y %H:%M:%S')
4      deltatime = datetime.now() - datetime_object
5      return deltatime.total_seconds()/3600
```

Listing 20. Klasa BoxGenerator.

```
1  class BoxGenerator:
2      def __init__(self, latest_records: dict, latest_timestamps: dict | str) -> None:
3          self.input_file = latest_records
4          self.timestamps = latest_timestamps
5          self.sensor_data = []
6          self.sensor_id = 0
7          self.html = ''
8          self.state = ''
9          self.timestamp = ''
10
11     def state_class(self) -> str:
12         '''change box class based on calculated sensor state'''
13         return f'<div class = "{self.state}"><br>\n'
14
15     def sensorID_label(self) -> str:
16         '''generate button with value and label based on sensor ID'''
17         return f'<form action = "/redirecting" method="POST">\n' \
18             f'<button type="submit" name="button" value="{self.sensor_id}">' \
19             f'SENSOR {self.sensor_id}</button>\n' \
20             f'</form>\n'
21
22     def timestamp_labels(self) -> str:
23         '''generate label with timestamp of latest record'''
24         return '<br><h2>Time of last record {} {}</h2>'.format(*self.timestamp)
25
26     def data_labels(self) -> str:
27         '''generate labels with reading from lastest record'''
28         return '<h2>Air temperature: {:.2f}*C</h2>\n' \
29             '<h2>Air humidity: {:.2f}%</h2>\n' \
30             '<h2>Soil humidity: {:.2f}%'</h2>\n</div>\n'.format(*self.sensor_data)
31
32     def merge(self) -> str:
33         '''merge generated lines into one string'''
34         a: str = self.state_class()
35         b: str = self.sensorID_label()
36         c: str = self.data_labels()
37         d: str = self.timestamp_labels()
38         return a+b+d+c
39
40     def replicate(self) -> None:
41         '''create boxes for every sensor contained in list of latest records'''
42         for key in self.input_file:
43             self.sensor_id = key
44             self.state = "activeSensor" if self.input_file[key][0] else "inactiveSensor"
45             self.sensor_data = self.input_file[key][1]
46             self.timestamp = self.timestamps[key]
47             self.html+=self.merge()
48
49     def generate_html(self) -> str:
50         '''merge generated bits and add immutable parts of html into one string'''
51         opener ='<!DOCTYPE html>\n<head>\n    <title>DASHBOARD</title>\n    <link rel="stylesheet" href="{{url_for('static', filename='css/style.css')}}"\n    <meta http-equiv="refresh" content="60">\n</head>\n<body>\n    <h2>DASHBOARD <span id="dash-board"></span></h2>\n'
52         self.replicate()
53         return opener+self.html+'</body>\n</html>'
54
55     def html_dump(self) -> None:
56         '''dump generated string into html file'''
57         with open('backend/templates/index.html', 'w') as f:
58             f.write(self.generate_html())
59             f.close()
```

Listing 21. Kod CSS.

```
1 html {
2   font-size: 16px;
3   background-color: lightsteelblue;
4 }
5
6 body {
7   font-family: 'Open Sans', sans-serif;
8   padding: 0;
9   padding: 0;
10}
11
12 button {
13   background-color: lightgray;
14   margin: 0;
15   width: 80%;
16   font-size: large;
17   font-weight: bold;
18   padding: 10px;
19   border: 5px solid black;
20   border-radius: 10px;
21 }
22
23 .inactiveSensor {
24   display: flex;
25   flex-direction: column;
26   background-color: tomato;
27   color: white;
28   padding: 10px;
29   border: 2px solid black;
30   border-radius: 10px;
31   margin: 50px;
32   text-align: center;
33 }
34 .activeSensor {
35   display: flex;
36   flex-direction: column;
37   background-color: green;
38   color: white;
39   padding: 10px;
40   border: 2px solid black;
41   border-radius: 10px;
42   margin: 50px;
43   text-align: center;
44 }
```

Szczegóły wyglądu interfejsu (zaokrąglenia pól, kolory, odstępy, itp.) zapisane są w formacie CSS (listing 21.). W pliku ze stylem znajdują się klasy determinujące wygląd pól dla aktywnego i nieaktywnego czujnika. Są one niemalże identyczne, różnią się jedynie kolorem jaki przypisywany jest do pola. Przypisanie klasy do odpowiedniego pola czujnika odbywa się na etapie generowania pliku HTML przez serwer (klasa *BoxGenerator*). Zastosowanie pliku

CSS jest niezwykle ważne z punktu widzenia estetyki interfejsu. Wygląd interfejsu bez pliku ze stylem został przedstawiony na Rys. 5.36.

## DASHBOARD

SENSOR 1

**Time of last record 04-03-2023 00:06:53**

**Air temperature: 26.78\*C**

**Air humidity: 32.84%**

**Soil humidity: 23.20%**

SENSOR 2

**Time of last record 04-03-2023 01:06:53**

**Air temperature: 26.78\*C**

**Air humidity: 32.84%**

**Soil humidity: 23.20%**

*Rys. 5.36. Interfejs wygenerowany bez zastosowania pliku ze stylem.*

### 5.3.5. Interfejs rysujący wykresy

Część interfejsu rysującego wykresy została stworzona przy pomocy Plotly. Jest to narzędzie do tworzenia interaktywnych wykresów. Pozwala na bardzo dużą personalizację tworzonych interfejsów przez dobieranie różnych typów wykresów i dostosowywanie ich parametrów takich, jak zmiana kolorów, stylów linii, etykiet osi itp. [9].

Plotly oferuje interaktywność, która pozwala użytkownikom na manipulowanie wykresami, w taki sposób jak zmiana skali osi, powiększanie lub pomniejszanie wykresu, wybieranie i porównywanie danych.

Wykresy przygotowywane są w części serwerowej. Odpowiada za to klasa *DataPlotter* (listing. 22.). Jej zadaniem jest przekonwertowanie danych z odpowiedniego pliku CSV do formatu wymaganego przez Plotly.

Przekonwertowane przez DataPlotter dane są odbierane przez część skryptową pliku HTML (listing 23.: linijki 22-31). Na ich podstawie rysowane są wykresy widoczne na Rys 5.37.

Listing 22. Klasa DataPlotter.

```
1  class DataPlotter:
2      def __init__(self, sensor_id) -> None:
3          self.sensor_id = sensor_id
4
5      def return_all_data(self) -> tuple:
6          '''create jsons with data from csv files for plotting'''
7          with open(f'backend/dataSensor{self.sensor_id}.csv', 'r') as file:
8              reader = csv.reader(file)
9              temperature = []
10             humidity = []
11             moisture = []
12             time = []
13             for row in reader:
14                 temperature.append(round(float(row[1]),2))
15                 humidity.append(round(float(row[2]),2))
16                 moisture.append(round(float(row[3]),2))
17                 time.append(f'{row[4]} {row[5]}')
18             temperature_df = pd.DataFrame(dict(time = time, temperature = temperature))
19             humidity_df = pd.DataFrame(dict(time = time, humidity = humidity))
20             moisture_df = pd.DataFrame(dict(time = time, moisture = moisture))
21             temperature_fig = px.line(temperature_df, x="time",
22                                         y="temperature",
23                                         title=f"Temperature from sensor {self.sensor_id}")
24             humidity_fig = px.line(humidity_df, x="time",
25                                     y="humidity",
26                                     title=f"Humidity from sensor {self.sensor_id}")
27             moisture_fig = px.line(moisture_df,
28                                     x="time",
29                                     y="moisture",
30                                     title=f"Soil moisture from sensor {self.sensor_id}")
31
32             temperatureJSON = json.dumps(temperature_fig,
33                                         cls=plotly.utils.PlotlyJSONEncoder)
34             humidityJSON = json.dumps(humidity_fig,
35                                         cls=plotly.utils.PlotlyJSONEncoder)
36             moistureJSON = json.dumps(moisture_fig,
37                                         cls=plotly.utils.PlotlyJSONEncoder)
38
39             return temperatureJSON, humidityJSON, moistureJSON
```

Listing 23. HTML odpowiadający za tworzenie wykresów.

```
1  <!doctype html>
2  <html>
3  <head>
4      <script
5          src="https://cdn.plot.ly/plotly-latest.min.js"></script>
6      <script
7          src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
8
9  </head>
10 <body style="font-family:arial, sans-serif">
11     <h1>Data plotting</h1>
12     <form action = "/redirecting" method='POST'>
13         <button type="submit" name="button" value="go_back">GO BACK</button>
14     </form>
15     <div id="t_chart" class="chart"></div>
16     <div id="h_chart" class="chart"></div>
17     <div id="m_chart" class="chart"></div>
18
19
20 </body>
21
22 <script>
23     moisture_data = {{ moistureJSON | safe }};
24     temperature_data = {{ temperatureJSON | safe }};
25     humidity_data = {{ humidityJSON | safe }};
26
27     Plotly.newPlot('t_chart', temperature_data, {});
28     Plotly.newPlot('h_chart', humidity_data, {});
29     Plotly.newPlot('m_chart', moisture_data, {});
30
31 </script>
32 </html>
33
```

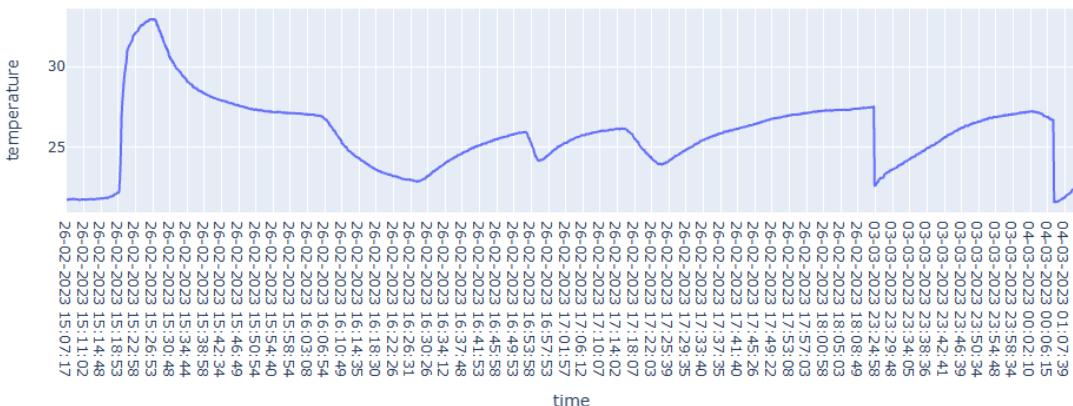
Przedstawiony na Rys 5.37. wykres został wygenerowany na podstawie danych zebranych z urządzenia pomiarowego. Można na nim zauważyc niskiej jakości działanie pojemnościowego czujnika wilgotności gleby oraz stabilny pomiar temperatury i wilgotności powietrza.

Dane wykorzystane do narysowania wykresu zostały zebrane w pomieszczeniu mieszkalnym i pokazują zmiany parametrów mikroklimatu w czasie około 4 godzin. Po 3 godzinach działania urządzenia pomiarowego, zostało ono wyłączone a następnie załączone po dłuższej przerwie, żeby zobaczyć, jak tak długi zanik zasilania wpłynie na jego funkcjonowanie. Nie zostały wykryte jakiekolwiek nieprawidłowości.

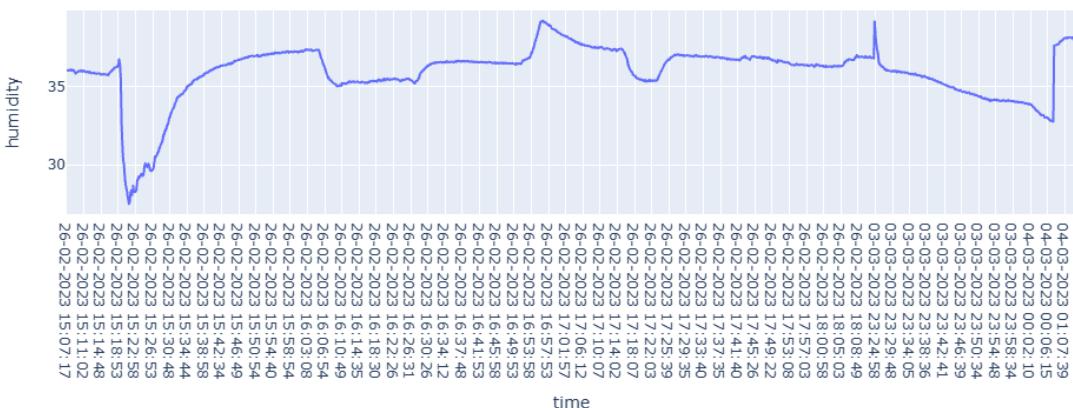
## Data plotting

[GO BACK](#)

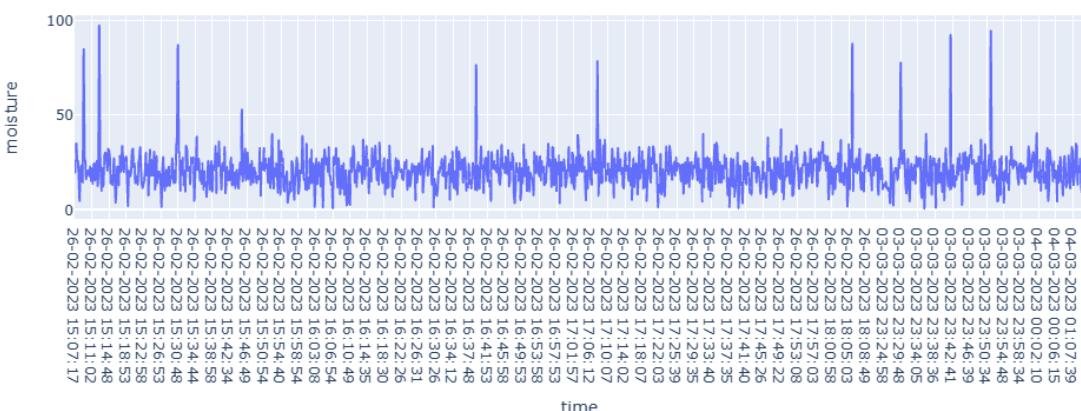
Temperature from sensor 1



Humidity from sensor 1



Soil moisture from sensor 1



Rys. 5.37. Interfejs rysujący wykresy.

### 5.3.6. Symulowanie działania z kilkoma czujnikami

Aby zasymulować działanie z kilkoma urządzeniami pomiarowymi, napisany został skrypt (listing 24.) wysyłający ramki danych dla dowolnej ilości urządzeń pomiarowych o dowolnych identyfikatorach. Pozwoliło to na zweryfikowanie czy system jest w stanie zebrać dane, wygenerować interfejs oraz wykresy dla większej ilości urządzeń pomiarowych.

Listing 24 Skrypt symulujący przesyłanie danych z urządzenia pomiarowego.

```
1 import requests
2 import json
3 import random
4 import time
5 from datetime import datetime
6
7 post_url = 'http://127.0.0.1:5000/data-collector'
8
9 def random_output(start:int, stop:int, accuracy:int) -> float:
10     return round(random.uniform(start,stop),accuracy)
11
12 def get_datetime() -> tuple:
13     now = datetime.now()
14     date_list = [now.day, now.month, now.year]
15     time_list = [now.hour, now.minute, now.second]
16     return date_list, time_list
17
18 def mock_sensor(body):
19     def wrapper(*arg, **kw):
20         requests.post(post_url, body(*arg, **kw))
21     return wrapper
22
23 @mock_sensor
24 def generate_dataframe(sensor_id = None):
25     print("dataframe generated")
26     sensor_id = random.randint(3,9) if sensor_id == None else sensor_id
27     return json.dumps({"sensor_id": sensor_id,
28                       "air_temperature": random_output(40,50,2),
29                       "air_humidity": random_output(20,25,2),
30                       "soil_moisture": random_output(0,50,1),
31                       "date": "{:02d}-{:02d}-{}".format(*get_datetime()[0]),
32                       "time": "{:02d}:{:02d}:{:02d}".format(*get_datetime()[1])})
33
34
35 if __name__ == "__main__":
36     while True:
37         generate_dataframe()
38         time.sleep(10)
```

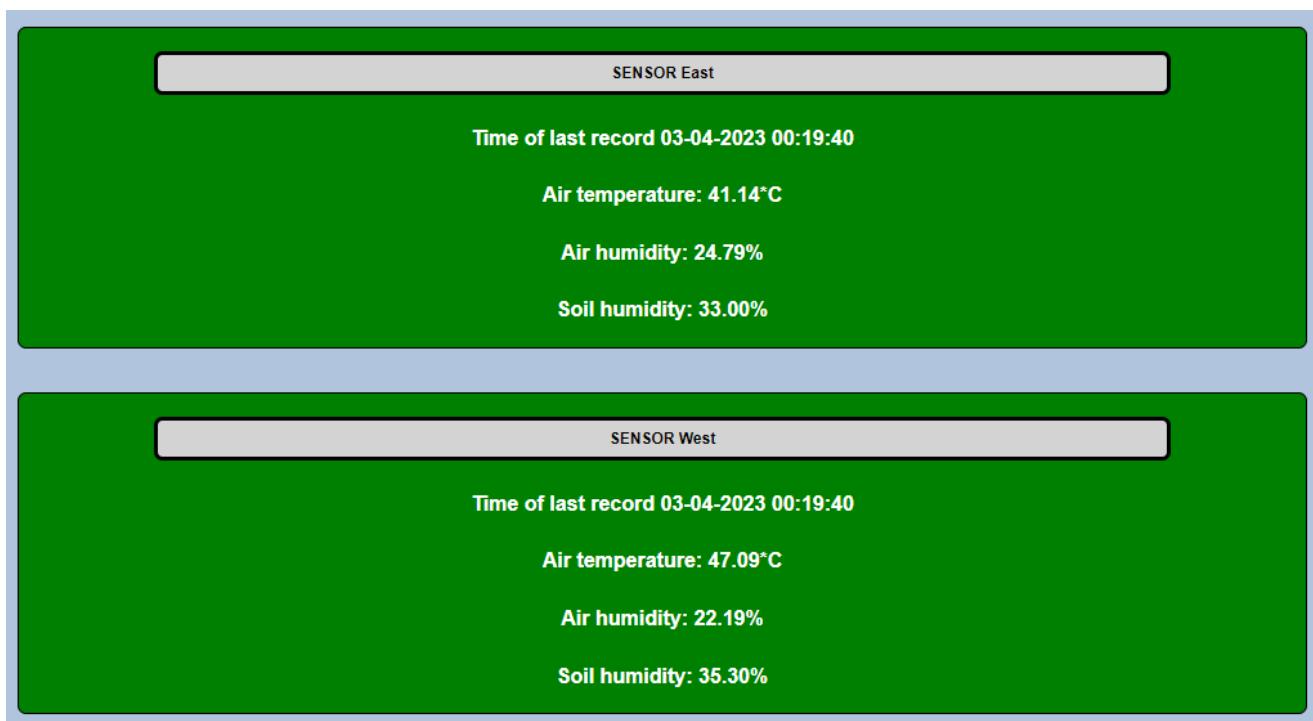
Skrypt symulujący działanie czujników działa na adresie lokalnym komputera (127.0.0.1), aby umożliwić testowanie systemu bez konieczności połączenia z siecią bezprzewodową. Ramka danych wypełniana jest losowymi wartościami dla ID sensora,

temperatury i wilgotności powietrza oraz wilgotności gleby. Znaczniki czasu zawierają rzeczywisty czas przesłania ramki danych.

Oprócz symulowania samych czujników, skrypt pozwolił na sprawdzenie, jak system radzi sobie z urządzeniami, których ID nie jest liczbą. W tym celu funkcja generate\_dataframe została wywołana w sposób przedstawiony na listingu 25. Interfejs (Rys. 5.38.) został prawidłowo wygenerowany. Oznacza to, że funkcjonalność pozwala na zastosowanie urządzeń pomiarowych o identyfikatorach, ułatwiających ich odnalezienie w przestrzeni.

*Listing 25. Wywołanie funkcji generate\_dataframe.*

```
1 generate_dataframe(sensor_id="East")
2 generate_dataframe(sensor_id="West")
```



*Rys. 5.38. Interfejs wygenerowany na podstawie symulowanych urządzeń pomiarowych.*

## **6. Podsumowanie i wnioski**

Rozwój technologii wpływa w znaczny sposób na przemysł, w tym na ten rolniczy. Niestety nowości techniczne docierają tu z opóźnieniem. Gałąź sprzętu pomiarowego przeznaczonego do szklarni jest niszowa, przez co dostępne urządzenia są albo bardzo drogie i przeznaczone dla profesjonalistów, albo tańsze i niezbyt dobrze przemyślane.

Stworzenie bezprzewodowego zespołu pomiarowego jest dużym wyzwaniem. Nie tylko przez to, jak złożony jest to temat, ale przede wszystkim przez to jak dużo możliwości daje rozwój technologii sieciowych. Prototyp urządzenia pomiarowego korzysta z sieci Wi-Fi i protokołu HTTP do przesyłania danych. Nie jest to jedyna prawidłowa opcja przesyłania informacji, tylko jedna z wielu.

W procesie projektowania należy sobie zdawać sprawę z możliwości i ograniczeń technologii sieciowych. Zastosowana w prototypie sieć Wi-Fi, z racji na swój zasięg, dobrze sprawdzi się w małych i średnich gospodarstwach. Może być też stosowana w przydomowych szklarniach, jednak w tak małych obiektach wystarczającą opcją mogłoby być stworzenie systemu ze stacją bazową i wykorzystaniem sieci Bluetooth.

Zaproponowane rozwiązanie, przez stosunkowo mały zasięg sieci Wi-Fi (do 150m) może nie sprawdzić się w dużych gospodarstwach, jeśli nie chciałoby się inwestować i urządzenia zwiększające zasięg. Alternatywą w takim przypadku mogłoby być zmodyfikowanie układu i oprogramowania w taki sposób, aby system obsługiwał sieć LoRaWAN, czyli protokół i system komunikacji bezprzewodowej małej mocy o zasięgu pomiędzy 10-15 km.

Ważnym aspektem jest dobór czujników. Zastosowany sensor SHT30 działa stabilnie, jednak pojemnościowy czujnik wilgotności gleby okazał się złym wyborem. Jego wskazania są niemiarodajne i prawdopodobnie lepiej by się sprawdził, jeśli zostałby zaprogramowany w charakterze czujnika działającego binarnie z przypisanymi wartościami sucho/mokro. Ciężko określić, czy stosując rozwiązania budżetowe istnieje alternatywa – jedynym rozwiązaniem w podobnym pułapie cenowym jest czujnik opierający się na pomiarze rezystancji, jednak ze względu na zasadę działania jego elektrody szybko ulegają korozji.

Biorąc pod uwagę, że system nie przetwarza dużych ilości danych i nie wymaga szybkiego działania podejście do tworzenia aplikacji serwerowej i wybór technologii okazał się prawidłowy. Język Python pozwolił na szybkie prototypowanie rozwiązań oraz stworzenie

czytelnego kodu opisującego poszczególne procedury. W przypadku, gdy chciałoby się przyspieszyć działanie poszczególnych procedur można by rozważyć przepisanie aplikacji na szybszy język o podobnym przeznaczeniu. W omawianym zastosowaniu sprawdziłyby się języki takie jak Java, C# lub PHP.

Aplikacja serwerowa działa stabilnie w sieci lokalnej i podczas testowania nawet z dużą ilością przychodzących ramek danych nie pojawiały się problemy, jednak, aby upewnić się,

że jej działanie pozostanie niezakłócone w czasie dłuższego działania, dobrym pomysłem na rozwój aplikacji byłoby wprowadzenie asynchroniczności w funkcjach obsługujących endpointy.

Nie da się jednoznacznie określić ile czujników należy zastosować w obiekcie oraz jak je rozmieścić. Wszystko zależy od wielu zmiennych, takich jak oczekiwana dokładność, powierzchnia obiektu, zastosowane systemy automatyzacji, dostępność sieci bezprzewodowej oraz zasilania. Logicznym jest założenie, że im więcej urządzeń pomiarowych zostanie zastosowanych, tym dokładniejsze będą odczyty i będzie możliwa stworzyć mapy cieplne obiektu.

W przypadku małych obiektów wystarczające może być jedno urządzenie pomiarowe umieszczone w centrum. Pomiar nie będzie zbyt dokładny, jednak wystarczający. Lepszym rozwiązaniem byłoby umieszczenie sensorów w rogach szklarni – łatwo byłoby wtedy zauważyc rozerwanie powłoki obiektu i zareagowanie odpowiednio szybko.

W obiektach ze zautomatyzowanym wietrzeniem optymalnym rozstawieniem urządzeń pomiarowych byłoby taki, gdzie przynajmniej jeden czujnik pokrywałby każdą wietrzoną sekcję. Dodatkowo jedno urządzenie można by umieścić na zewnątrz obiektu, aby komputer klimatyczny odpowiedzialny za sterowanie automatyką (automatyczne lufty, turbiny mieszające powietrze, itp.) mógł z wyprzedzeniem zaplanować reakcję na zmianę warunków zewnętrznych.

## **7. Literatura**

1. Degirmendžić J., Kożuchowski K., „Meteorologia i klimatologia”, 2006, PWN.
2. Jankiewicz L. S., Lech W., Borys M. W., „Fizjologia roślin sadowniczych” Wydanie: II, „1. Regulacja procesów fizjologicznych w roślinie”, 1984, PWN.
3. Informacje na temat produktów firmy Aranet:  
<https://aranet.com/> (5.02.2023)
4. Informacje na temat produktów firmy UbiBot:  
<https://ubibot.pl/> (5.02.2023)
5. Informacje na temat ESP32:  
<https://espressif.com/en/support/documents/technical-documents> (12.02.2023)
6. Informacje na temat czujnika SHT30:  
[https://mouser.com/datasheet/2/682/Sensirion\\_Humidity\\_Sensors\\_SHT3x\\_Datasheet\\_digital-971521.pdf](https://mouser.com/datasheet/2/682/Sensirion_Humidity_Sensors_SHT3x_Datasheet_digital-971521.pdf) (12.02.2023)
7. Informacje na temat HTTP:  
<https://sekurak.pl/protokol-http-podstawy/> (12.02.2023)
8. Informacje na temat framework'a Flask:  
<https://flask.palletsprojects.com/> (14.02.2023)
9. Informacje na temat Plotly:  
<https://plotly.com/> (10.02.2023)
10. Informacje o temperaturze dobowej w Polsce:  
<https://imgw.pl/wydarzenia/charakterystyka-wybranych-elementow-klimatu-w-polsce-w-2022-roku-podsumowanie> (5.02.2023)
11. Informacje na temat tworzywa PLA:  
<http://reprap.org/wiki/PLA> (6.02.2023)

12. Informacje na temat rezystancyjnych czujników wilgotności gleby:

<https://forbot.pl/blog/czujniki-wilgotnosci-gleby-dlaczego-sa-tak-problematyczne-id52948> (6.02.2023)

13. Informacje na temat systemu pomiarowego Ridder HortiMaX-Go:

<https://ridder.com/ridder-hortimax-go/> (15.04.2023)

## **8. Streszczenie**

Tematem pracy jest „System automatyki szklarni z wykorzystaniem sieci bezprzewodowej”. Celem pracy było zaprojektowanie i stworzenie prototypu urządzenia pomiarowego mierzącego parametry mikroklimatu w szklarni.

W pierwszej części pracy opisany został wpływ klimatu na cykl życia rośliny. Z naciskiem na jej najważniejsze procesy życiowe, czyli fotosyntezę i transpirację.

W drugiej części pracy przedstawiony został przegląd rynku rozwiązań urządzeń pomiarowych przeznaczony do użytku w szklarni oraz analiza wad, ograniczeń i potencjalnych zagrożeń opisywanych rozwiązań

Ostatnia część pracy opisuje proces projektowania urządzenia i tworzenia jego oprogramowania.

Część sprzętowa opisywanego urządzenia została wykonana z zastosowaniem pojemnościowego czujnika wilgotności gleby, sensora SHT30 oraz mikrokontrolera ESP32. Obudowa powstała metodą druku 3D.

Oprogramowanie mikrokontrolera powstało z wykorzystaniem frameworka Arduino w języku będącym pochodną C/C++. Aplikację serwerową napisane w języku Python stosując bibliotekę Flask.

Słowa kluczowe: szklarnia, ESP32, czujnik SHT30, pojemnościowy czujnik wilgotności gleby, WiFi

## **9. Abstract**

The subject of the work is "Greenhouse automation system using wireless network". The purpose of the work was to design and develop a prototype measuring device that measures microclimate parameters in a greenhouse.

In the first part of the work, the influence of climate on the life cycle of a plant is described. With emphasis on its most important life processes, namely photosynthesis and transpiration.

The second part of the work presents an overview of the market for measuring device solutions designed for use in the greenhouse, as well as an analysis of the disadvantages, limitations and potential risks of the solutions described

The last part of the work describes the process of designing the device and developing its software.

The hardware part of the described device was made using a capacitive soil moisture sensor, SHT30 sensor and ESP32 microcontroller. The enclosure was created using the 3D printing method.

The microcontroller software was created using the Arduino framework in a language that is a derivative of C/C++. The server application was written in Python using the Flask library.

**Keywords:** greenhouse, ESP32, SHT30 sensor, capacitive soil moisture sensor, WiFi