Politechnika Wrocławska

Stacja Pogodowa
Projekt Zespołowy

Jakub Jakubczak 263486
Sebastian Furmaniak 264323
Kacper Ugorny 263504

Spis treści

1 Wstęp

Dzisiaj, w czasach rozwoju technologii, szczególnie popularnym konceptem są inteligentne urządzenia Internetu Rzeczy - IoT. W róznych dziedzinach niezbędne jest monitorowanie i analizowanie wszelakich parametrów. Następnie te informacje muszą zostać zapisane i przedstawione w prosty i czytelny sposób

Nasz projekt - stacje pogodowe - ma za zadanie zbierać informacje atmosferyczne w dowolnym miejscu z połączeniem telefonicznym, zapisywać je w bazie danych i przedstawiać w aplikacji mobilnej. Zaproponowany przez nas system, ma być roszerzalny o dowolną ilość stacji pogodowych.

Ten dokument opisuje szczegóły realizacji projektu, zacznając od celu i koncepcji, a następnie pokazuje oddzielnie trzy części projektu zaczynając od urządzenia - opisując dobór komponentów i oprogramowanie, idąc przez API i baze danych pokazujące sposób komunikacji naszego systemu, kończąc na aplikacji mobilnej i sposobie wizualizacji danych.

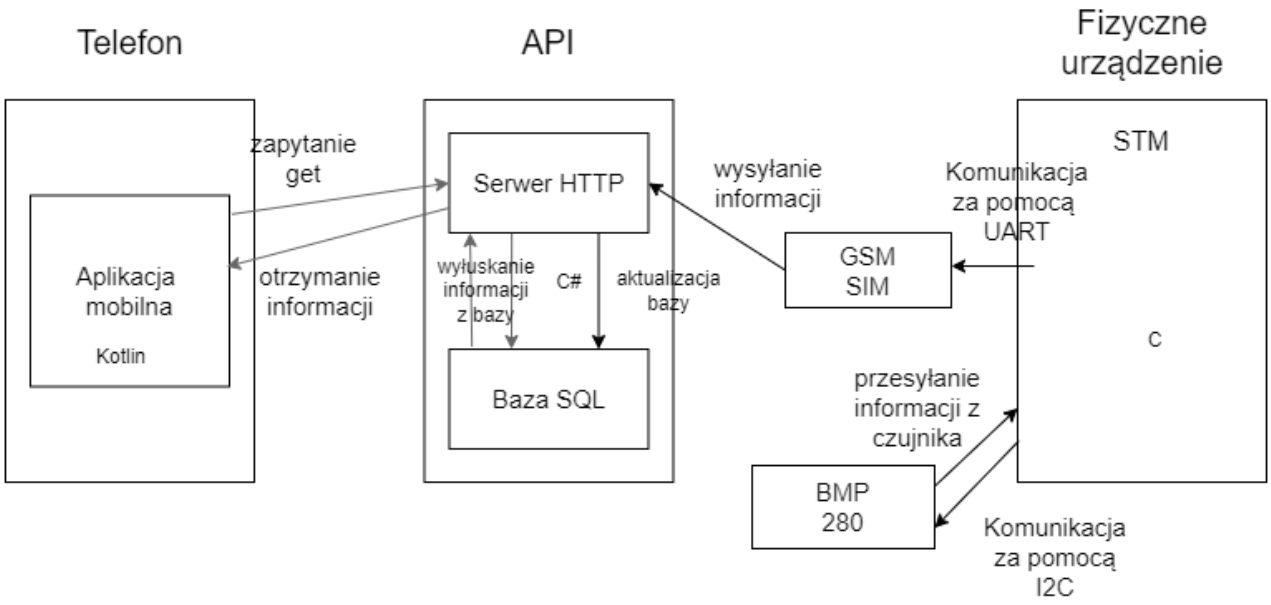
2 Cel Projektu

Celem naszego projektu jest stworzenie fizycznego urządzenia IoT, tj. stacji pogodowej z interfejsem w postaci aplikacji mobilnej. Informacje będą przechowywane w bazie danych na serwerze, a całość będzie obsługiwana przez odpowiedni back-end. Docelowo użytkownik będzie mógł dodawać wiele urządzeń do aplikacji, w celu obserwacji temperatury, ciśnienia oraz wilgotności powietrza w różnych miejscach.
W celu realizacji projektu podzieliliśmy go na trzy odrębne części:

- Stworzenie fizycznego urządzenia
- API
- Aplikacja mobilna

2.1 Schemat

Na poniższym schemacie widać koncepcje działania naszego urządzenia. Widocznie wyodrębnione są trzy częsci tego projektu - Telefon, API oraz urządzenie. Rzeczywisty system może różnić się od tego pokazanego na schemacie ilością urządzeń mikrokontrolerowych i telefonów, których można dodać dowolną ilość. Po przeanalizowaniu tego systemu wyodrębniają się dwie ścieżki, którymi podążają dane.
Obieg 1:
Pierwszy obieg informacji zaczyna się w urządzeniu mikrokontrolerowym, które odpytuje czujniki w celu pozyskania danych atmosferycznych, następnie korzystając z modułu GSM informacje wysyłane na serwer i zapisywane przez API na bazie danych. Urządzenie otrzymuje odpowiedź od API w postaci kodu odpowiedzi HTTP.
Obieg 2:
Drugi obieg informacji rozpoczyna się w aplikacji mobilnej, które wysyła zapytanie HTTP typu GET na serwer. Następnie odpowiednie informacje są wyłuskiwane z bazy danych i wysyłane jako odpowiedź tego zapytania.



Rysunek 1: Koncept projektu

3 Urządzenie

W tym rozdziale zostały opisane komponenty sprzętowe, czemu zostały wybrane oraz ich parametry. Ukazany również będzie schemat elektroniczny.

3.1 Opis

W skład realizacji tej części wchodzi napisanie odpowiedniego kodu na mikrokontroler STM32L obsługującego moduły takie jak: moduł z czujnikami AHT20 + BMP280 do pomiaru temperatury, wilgotności powietrza oraz ciśnienia, moduł GSM do komunikacji z Internetem. Urządzenie ma posiadać własne źródło zasilania, aby mogło działać w najróżniejszych miejscach. W tym celu trzeba zoptymalizować zużycie prądu, aby stacja była jak najbardziej energooszczędna.

3.2 Komponenty sprzętowe

Dobór komponentów jest bardzo ważnym aspektem przy tworzeniu urządzenia. Pod uwagę należy wziąść parametry sprzętowe.

3.2.1 Czujniki

Do pomiaru potrzebnych wartości wykorzystano moduł z czujnikami AHT20, BMP280. Czujnik AHT20 pozwala na pomiar temperatury oraz wilgotności, a BMP280 na pomiar ciśnienia oraz temperatury. Moduł umożliwia komunikacje z czujnikami za pomocą protokołu I2C.
Parametry AHT20:

- Napięcie zasilania 3,3V
- Zakres pomiaru wilgotności 0%-100%RH
- Typowa dokładność pomiaru wilgotności ±2%RH
- Zakres pomiaru temperatury -40°C - 85°C

- Typowa dokładność ± 3°C

Parametry BMP280:

- Napięcie zasilania 3,3V
- Zakres pomiarowy ciśnienia: 300 ÷ 1100 hPa
- Dokładność absolutna: ±1 hPa dla zakresu 950 – 1050 hPa @0 – 40°C
- Zakres pomiarowy temperatury ogólny: od -40°C do 85°C
- Dokładność: ±1,0°C
- Zakres pomiarowy temperatury o pełnej dokładności: od 0°C do 65°C
- Dokładność: ±0,5°C

3.2.2 Moduł GSM

Do komunikacji z internetem wykorzystano moduł GSM SIM800L, który obsługuje transmisje danych pakietowych, przy pomocy GPRS. Komunikacja z tym modułem jest przeprowadzana przez UART, wysyłając odpowiednie komendy AT. Ten moduł pozwala również na uzyskanie pozycji urządzenia korzystając z triangulacji.
Parametry SIM800L:

- Napięcie zasilania: 3,7 ÷ 4,2 V
- Gniazdo microSIM
- Pobór prądu: max 2A

3.2.3 Mikrokontroler

Do urządzenia dobrano mikrokontroler STM32L476RET6. W wyborze kierowano energooszczędnością, którą gwaratnują mikrokontrolery STM32 z serii L. Prototyp urządzenia był opracowany na płytce prototypowej NUCLEO L476RG - wersji tego samego mikrokontrolera z większą ilością pamięci. Najważniejszą funkcjonalnością tego mikrokontrolera jest możliwość wejścia w Standby mode z wybudzeniem przez zegar RTC, w którym pobór prądu mikrokontrolera spada do 420nA.
Parametry STM32L476RET6:

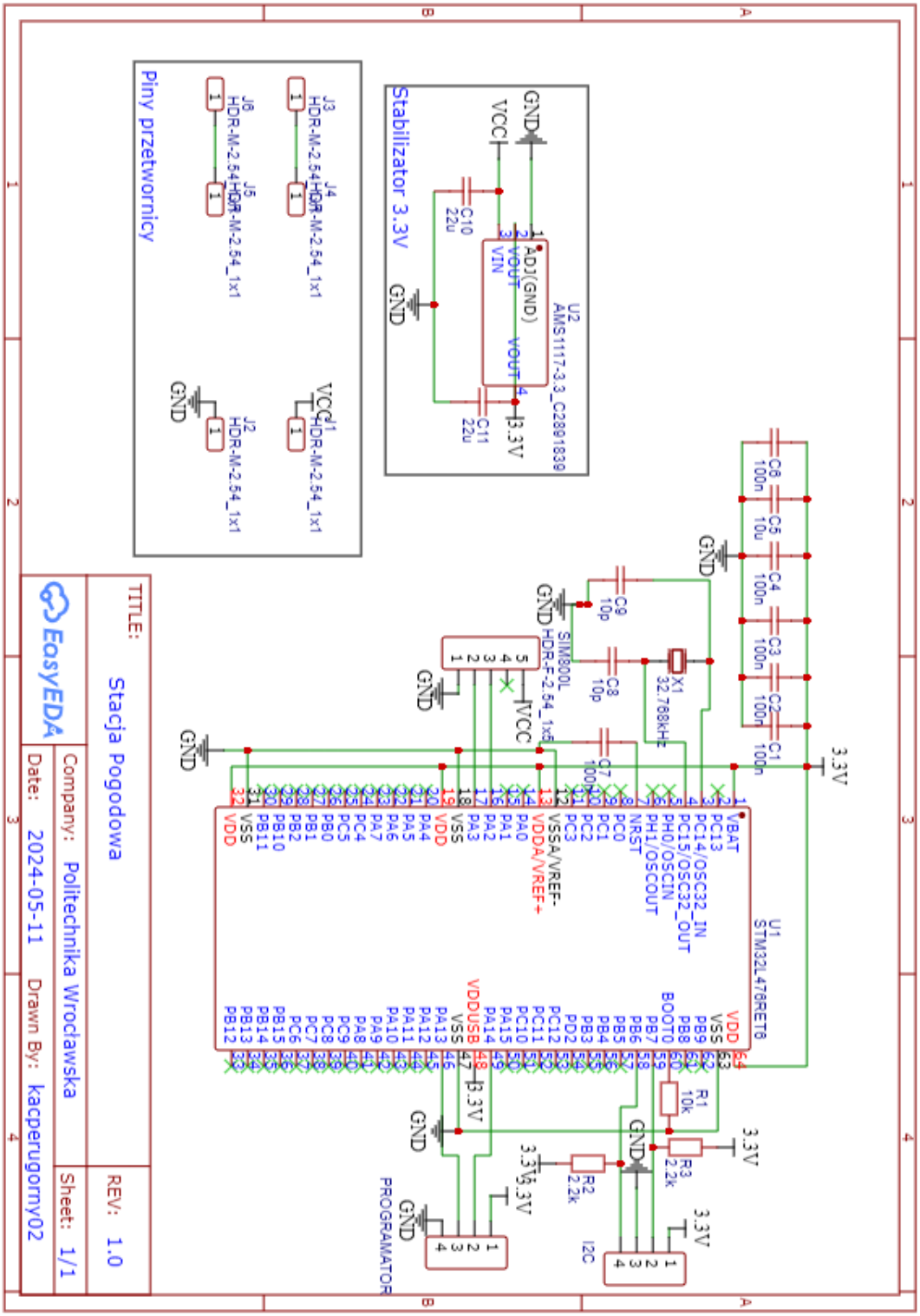
- Napięcie zasilania: 1,71 V - 3,6 V
- 512KB pamięci flash
- Taktowanie 80MHz

3.2.4 Zasilanie

Do zasilania urządzenia użyto gotowej przetwornicy stepdown obniżającej napięcie do 4V - do zasilania modułu GSM oraz stabilizatora napięcia 3.3V do zasilania mikrokontrolera i układu z czujnikami. Aby podtrzymać wymagany prąd 2A na module SIM800L zastosowano na wyjściu przetwornicy dodatkowy kondensator.

3.3 Schemat Elektroniczny

Poniżej widoczny jest schemat elektroniczny przedstawiający połączenia pomiędzy komponentami, a mikrokontrolerem.



Rysunek 2: Schemat elektroniczny

3.3.1 Mikrokontroler

Mikrokontroler, STM32L476RET6 na schemacie jest centralną jednostką sterującą stacją pogodową. Zasilany jest on z 3,3V pochodzących z stabilizatora AMS1117. Wyprowadzone zostały różne piny połączone z modułem GSM, czujnikami oraz piny programatora.
Na schemacie widoczny jest również oscylator X1 32,768kHz podłączony do PC14 i PC15 służący jako stabline taktowanie zegara RTC.

3.3.2 Zasilanie

W piny J4,J5,J1,J2 wpinana jest przetwornica step down a piny J3 i J6 służą do podłączenia zasilania przetwornicy. Wyjście - J1 i J2 przetwornicy są bezpośrednio podłączone z pinami SIM800L - wymagane 4V. Te 4V są następnie przez stabilizator AMS1117 jest zmniejszane do 3,3V, które służy do zasilania reszty układów.

3.3.3 Programator

Złącze programatora pozwala na programowanie urządzenia za pomocą intefejsu SWD - Serial Wire Debug, używane również do debugowania programu. Wyprowadzone piny do SWDIO, SWCLK oraz zasilanie.

3.3.4 Czujniki

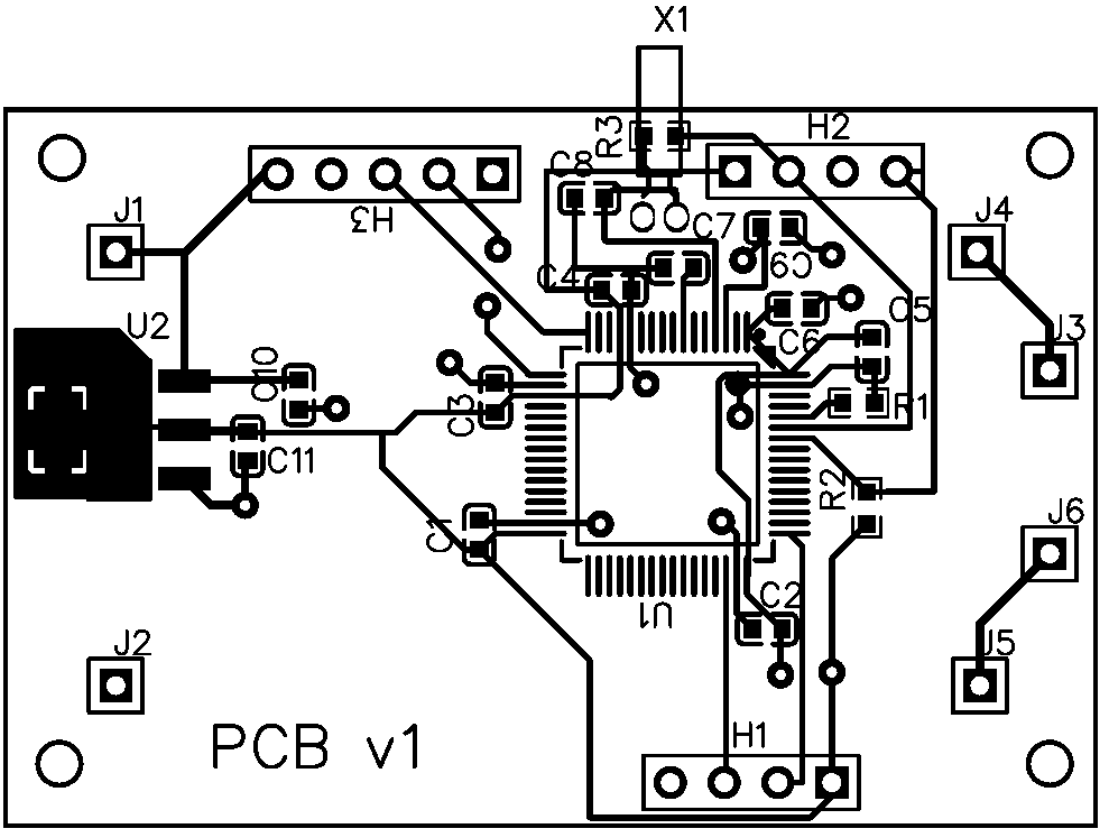
Złącze do modułu czujników jest podłączone z interfejsem I2C mikrokontrolera wraz z wymaganymi w tej komunikacji rezystorami podciągającymi.

3.3.5 GSM

Złącze dla modułu GSM jest połączone z mikrokontrolerem za pomocą UART i zasilanie prosto z przetwornicy.

3.4 Widok płytki z góry

Poniżej widoczny jest widok płytki PCB. Widoczne na niej są fizyczne połączenia.
H1 - Złącze programatora - od lewej GND, SWDIO, SWCLK, VCC
H2 - Złącze modułu z czujnikami - od lewej VCC, SDA,GND, SCL
H3 - Złącze modułu SIM800L - od lewej VCC, nie podłączone, TX, RX, GND



Rysunek 3: Widok płytki z góry

4 Oprogramowanie Urządzenia

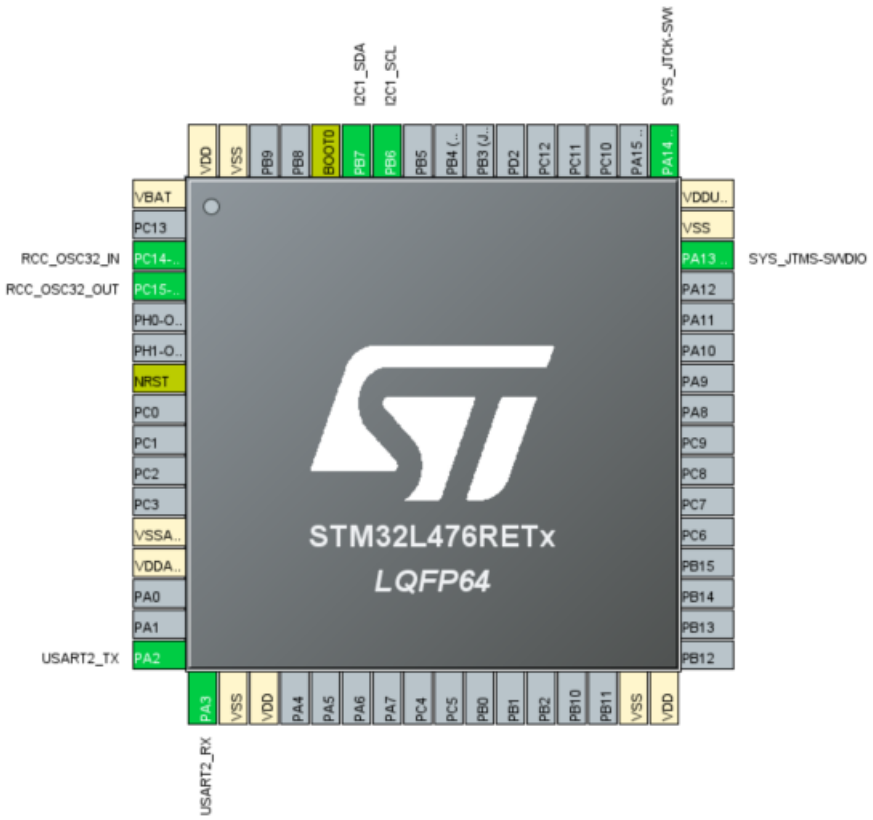
Wyżej wymieniony mikrokontroler został zaprogramowny w środowisku STM32CubeIDE w języku C, które pozwala na konfigurację oraz programowanie mikrokontrolerów firmy STMicroelectronics. Wykorzystano bibliotekę HAL (Hardware Abstraction Layer).

4.1 Biblioteka HAL

Biblioteka HAL dostarczana przez producenta mikrokontrolerów STM, umożliwia łatwiejsze programowanie mikrokontrolerów tej firmy, zapewniając szeroki zestaw funkcji wysokiego poziomu. Pozwala to na łatwą migracje kodu, między mikrokontrolerami tego producenta, bez potrzeby nauki każdego z osobna.

4.2 STM32Cube

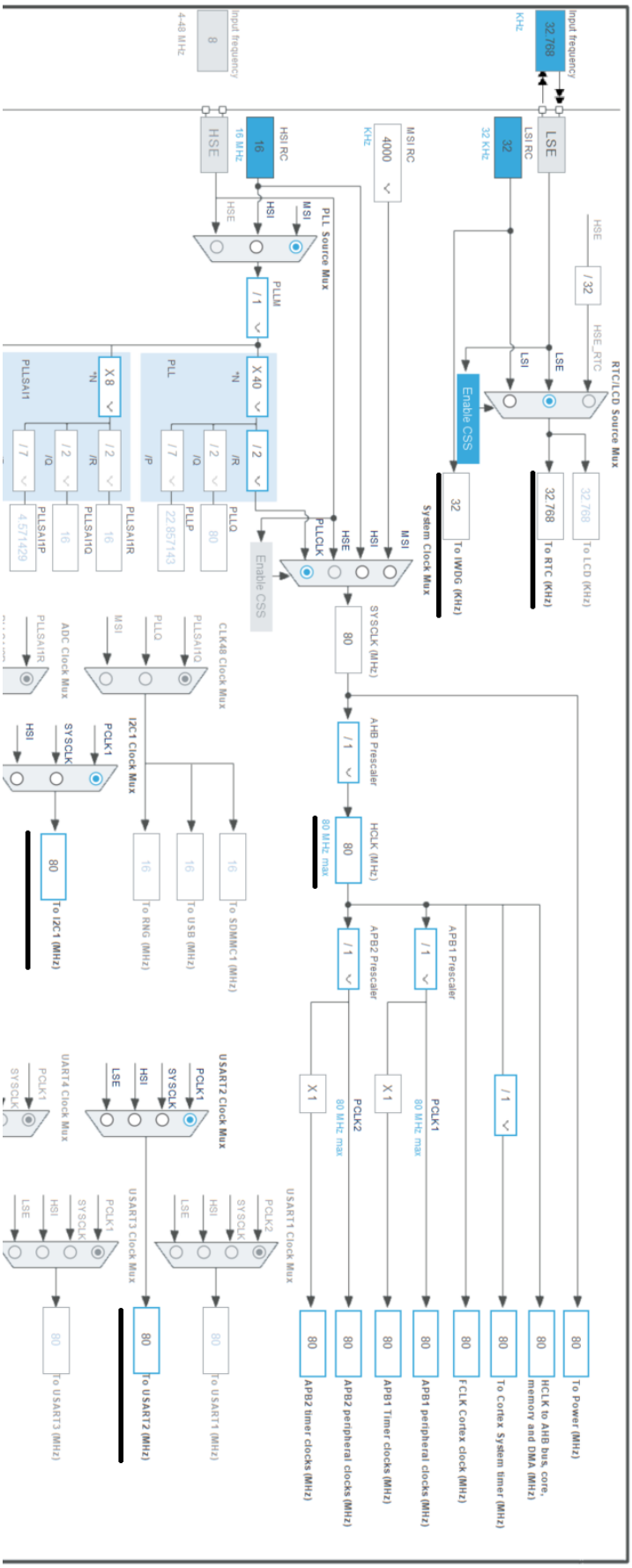
STM32Cube jest częścią środowiska programistycznego STM32CubeIDE, pozwalająca na generowanie kodu odpowiedzialnego za konfiguracje mikrokontrolera. Jest to bardzo przydatne, gdyż ten mikrokontroler posiada dużą ilość peryferiów. W tym projekcie skonfigurowane zostały: UART, I2C, WatchDog, tryb Debug, oraz taktowanie mikrokontrolera. W następnych podrozdziałach zostaną opisane ważne elementy konfiguracji.



Rysunek 4: Widok konfiguracji mikrokontrolera

4.2.1 Konfiguracja zegarów mikrokontrolera

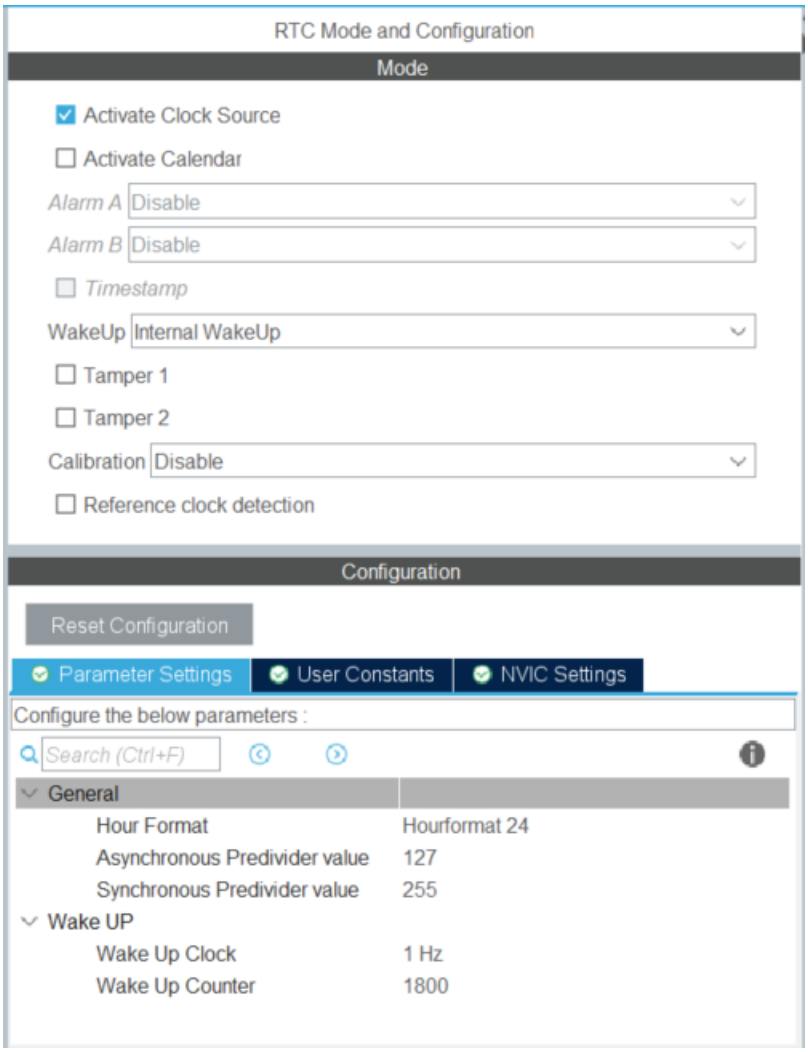
W konfiguracji jako źródło LSE - Low Speed Clock, wybrano zewnętrzny oscylator, który został wlutowany w płytkę. Następnie skonfigurowano zegar - najważniejsze elementy konfiguracji zostały podkreślone na poniższym obrazku. Od góry widać częstotliwości jakie ustawiono dla modułu RTC oraz Watchdog - IWDG. Trzecie podkreślenie od góry to częstotliwość działania mikrokontrolera - w tym przypadku ustawiona została maksymalna 80MHz. Następne dwa podkreślenia to częstotliwość używana przez UART i I2C, które również zostały ustawione na maksymalną wartość 80MHz.



Rysunek 5: Konfiguracja zegarów

4.2.2 Konfiguracja RTC

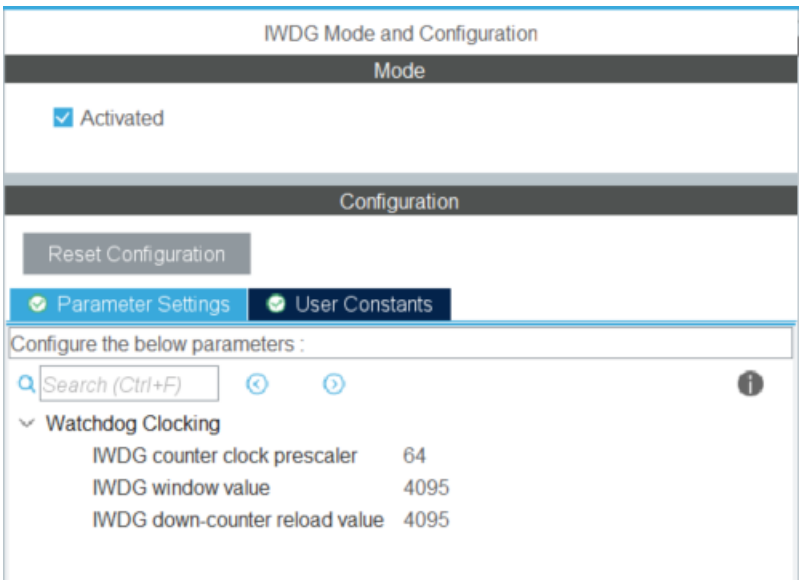
Moduł RTC skonfigurowano w celu wybudzania mikrokontrolera, z trybu Standby Mode. Do konfiguracji RTC wykorzystano wcześniej skonfigurowany LSE - Low Speed Clock. Na poniższym rysunku widać konfigurację, istotnym dla nas było ustawienie WakeUp na Internal WakeUp oraz wartości w tabelce poniżej - Wake Up Clock na 1Hz i Wake Up Counter na 1800. Te wartości gwarantują impuls wybudzającdzy mikrokontroler raz na 30 minut.



Rysunek 6: Konfiguracja RTC

4.2.3 Konfiguracja Watchdog

Watchdog to spotykana w mikrokontrolerach funkcjonalość, wykrywająca błędne działanie systemu. W praktyce jest to układ zabezpieczający przed zbyt długim przebywaniem w stanie zawieszenia. W oknie z konfiguracją zegarów można było odczytać, że WatchDog ma zegar działający z częstotliwością 32kHz. W tym przypadku ustawiono prescaler na 64 i wartość licznika na 4095, co daje zresetowanie programu w przypadku zawieszenia po osmiu sekundach. $T = \frac{4095}{32000/64} \approx 8[s]$. Resetowanie tego countera odbywa się za pomocą polecenia "HAL_IWDG_Refresh(&hiwdg);".



Rysunek 7: Konfiguracja WatchDog

4.3 Komunikacja z czujnikami

Komunikacja z czujnikami jest przeprowadzana na jednej magistrali I2C. Komunikacja po tym interfejsie odbywa się przy pomocy adresacji 7 bitach, dzięki czemu w teorii można na jednym zestawie pinów obsłużyć 127 urządzeń.

4.3.1 Komunikacja z AHT20

Adres czujnika AHT20 to 0111000, korzystając z tej wiedzy i dokumentacji napisano następujący kod obsługujący komunikację z tym czujnikiem. W tablicach init i measure zapisano pokolei polecenia do inicializacji urządzenia i do wyzwolenia pomiaru przez czujnik. Następnie kod od góry sprawdza czy potrzebna jest inicializacja, jeśli tak to ją wykonuje i po 10ms wykonuje wyzwolenie pomiaru. Czujnik według dokumentacji potrzebuje 80ms na wykonanie pomiaru. Następnie dane są odczytywane i odpowiednio przeliczane.

```
1  const uint16_t addr = 0b0111000; //0x38
2  const int addr_wr = addr<<1;
3  const int addr_rc = addr_wr + 1;
4  uint32_t temp_data;
5  uint8_t init[3] = {0xbe, 0x08, 0x00};
6  uint8_t measure[3] = {0xac, 0x33, 0x00};
7  uint8_t data[6];
8  uint32_t time_stamp = HAL_GetTick();
9  HAL_I2C_Master_Receive(&hi2c1, addr_rc, (uint8_t *)data, 6, 1000);
10 if((data[0] >> 3 & 1) != 1){
11   HAL_I2C_Master_Transmit(&hi2c1, addr_wr, (uint8_t *)init, 3, 1000);
12 }
13 time_stamp = HAL_GetTick();
14 while(HAL_GetTick() - time_stamp < 10) ;
15 HAL_I2C_Master_Transmit(&hi2c1, addr_wr, (uint8_t *)measure, 3, 1000)
     ;
16 time_stamp = HAL_GetTick();
17 while(HAL_GetTick() - time_stamp < 85) ;
18 HAL_I2C_Master_Receive(&hi2c1, addr_rc, (uint8_t *)data, 6, 1000);
19 //AHT20 COMM FINISHED
```

```
20    HAL_IWDG_Refresh(&hiwdg);
21    //check control byte and calculate values
22    if(((data[0] >> 7) & 1) == 0) {
23      temp_data = ((uint32_t)data[3] << 16) + ((uint32_t)data[4] << 8) +
          (uint32_t)data[5];
24      temp_data = temp_data & (~(0xFFF00000));
25      temp = ((float)temp_data/1048576) * 200 - 50;
26      temp_data = ((uint32_t)data[1] << 16) + ((uint32_t)data[2] << 8) +
          (uint32_t)data[3];
27      temp_data = temp_data >> 4;
28      humi = ((float)temp_data/1048576) * 100;
29    }
```

4.3.2 Komunikacja z BMP280

Adres czujnika BMP280 jest zależny jednego z jego pinów, zależnie czy jest połączony z Vcc czy GND. W przypadku tego modułu ten pin jest podciągnięty do 3,3V i jego adres to 1110111. Komunikacja w tym przypadku odbywa się przy pomocy biblioteki.

```
1     bmp280_init_default_params(&bmp280.params);
2     bmp280.addr = BMP280_I2C_ADDRESS_1;
3     bmp280.i2c = &hi2c1;
4     while (!(bmp280_init(&bmp280, &bmp280.params)))
5     {
6       time_stamp = HAL_GetTick();
7       while(HAL_GetTick() - time_stamp < 2000) ;
8     }
9     HAL_IWDG_Refresh(&hiwdg);
10    bool bme280p = bmp280.id == BMP280_CHIP_ID;
11    time_stamp = HAL_GetTick();
12    while(HAL_GetTick() - time_stamp < 100) ;
13    while (!bmp280_read_float(&bmp280, &temperature, &pressure, &humidity
          )) {
14        time_stamp = HAL_GetTick();
15        while(HAL_GetTick() - time_stamp < 2000) ;
16    }
17    HAL_IWDG_Refresh(&hiwdg);
18    p = pressure/100;
```

4.3.3 Komunikacja z SIM800L

Komunikacja z modułem GSM odbywa się przy pomocy interfejsu UART, obojętnie z jaką prędkością transmisji, gdyż moduł automatycznie się dostosowuje. Aby dogadać się z tym modułem należy wykorzystać tzw. komendy AT. Poniżej pokazana jest wykonywana przez program sekwencja poleceń.

```
1       "AT\r",
2       "AT+SAPBR=3,1,Contype,"GPRS"\r",
3       "AT+SAPBR=3,1,APN,"internet"\r",
4       "AT+SAPBR=1,1\r",
5       "AT+CLBS=1,1\r",
6       "AT+CNUM\r",
7       "AT+HTTPINIT\r",
```

```
    "AT+HTTPPARA=CID,"1"\r",
    "AT+HTTPSSL=0\r",
    "AT+HTTPPARA=URL,"URL"\r",
    "AT+HTTPACTION=1\r",
    "AT+HTTPTERM\r",
    "AT+SAPBR=0,1\r"
```

Polecenia AT+SAPBR służą do połączenia się z usługą GRPS, która pozwala przesyłać pakiety internetem. Następnie AT+CLBS zwraca przybliżoną lokalizację. AT+CNUM zwraca numer karty sim. Polecenia AT+HTTP służą do konfiguracji połączenia i wykonania zapytania HTTP. AT+SAPBR = 0,1 wyłącza transmisje pakietów.
Polecenia są kolejno wywoływane dzięki zastosowaniu UART z mechanizmem przerwań - W funkcji main wywołane jest polecenie odbierające jeden bajt z uart, a cała logika znajduję się we funkcji HAL_UART_RxCpltCallback, która na samym końcu znowu wywołuje odbieranie jednego bajta. Funkcja dopisuje do po jednym bajcie do bufora Rx_data i jeśli trafi na znak końca linii to zaczyna interpretować odebrane informację. W niektórych przypadkach zapisuje informacje - lokalizacje i numer telefonu, aby potem wykorzystać je w zapytaniu HTTP.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  //Sim800l repeats the command of the user so this line skips it
  if(read == false && Rx_bit == '\n') read = true;
  else if(read == true){
    Rx_data[pos++] = Rx_bit;
    if(Rx_bit == '\n'){
      //Read latitude and longitude
      if(msg == 4 && Rx_data[0] == '+'){
    HAL_UART_Receive(&huart2, (uint8_t*)Rx_placeholder, 6, 1000);
    strcpy(Locs,Rx_data);
    pos = 0;
      }
      //read number
      else if(msg == 5 && Rx_data[0] == '+'){
    HAL_UART_Receive(&huart2, (uint8_t*)Rx_placeholder, 6, 1000);
    strcpy(Num,Rx_data);
    pos = 0;
      }
      //This line makes program wait for http response
      if(msg == 10 && Rx_data[0] == 'O' && Rx_data[1] == 'K') {
      pos = 0;
      }
      //Here it writes next functions, and sets the post url
      else{
    pos = 0; read = false; msg++;
    if(msg == 9){
      float latitude = 0.0, longitude = 0.0;
      char *numberStart = strstr(Num,"\"+");
      char *numberEnd = strstr(numberStart + 2,"\"");
      size_t len = numberEnd - numberStart;
      char num[len + 1];
      strncpy(num, numberStart, len);
      num[len] = '\0';
      strcpy(num, &num[4]);
      sscanf(Locs, "+CLBS: 0,%f,%f,", &longitude, &latitude);
```

```
37        sprintf(Tx_data[msg], "AT+HTTPPARA=URL,\"http://url/NewData?num=%
              s&latitude=%.6f&longitude=%.6f&temp=%.2f&humi=%.2f&press=%.2f&
              code=%s\"\r",num,latitude,longitude,temp-2,humi,pressure,code)
              ;
38      }
39      if(msg < 13)
40        HAL_UART_Transmit_IT(&huart2, (uint8_t *)Tx_data[msg], strlen(
              Tx_data[msg]));
41      else
42        completed = true;
43        }
44      }
45    }  //keep receving data
46    HAL_UART_Receive_IT(&huart2, &Rx_bit, 1);
47    HAL_IWDG_Refresh(&hiwdg);
48  }
```

4.3.4 Usypianie urządzenia

Po wykonaniu wszystkich poleceń AT urządzenie jest usypiane - przechodzi w energooszczędny tryb Standby mode. Na poniższym kodzie jest widoczne, że aplikacja czeka uwięziona w pętli while dopóki nie skończy się transmisja danych, a gdy się skończy przechodzi w stan uśpienia. Warto wspomnieć, że we funkcji odbioru danych występują polecenia resetujące licznik Watch dog'a, więc program nie jest resetowany.

```
1   while(!completed);
2
3   HAL_PWR_EnterSTANDBYMode();
```

5 API

Jednym z kluczowych elementów projektu jest API (Interfejs Programowania Aplikacji), które umożliwia wymianę danych między różnymi systemami. W niniejszym rozdziale przedstawione zostanie API zaprojektowane w celu odbierania, przechowywania oraz udostępniania informacji o pogodzie i stacjach nadawczych.

5.1 Opis

Celem API jest odbieranie informacji o pogodzie i stacji, która zostanie wysłana przez fizyczne urządzenie, przechowywanie tych informacji oraz zapewnienie dostępu do danych dla aplikacji mobilnej przy użyciu protokołu http.
Przy tworzeniu API zostały wykorzystane technologie/narzędzia takie jak:

- Język programowania C#.
- Framework .NET.
- Baza danych MS SQL.
- Azure do stworzenia serwera i wdrożenia strony z bazą danych.
- Postman do testowania zapytań http.

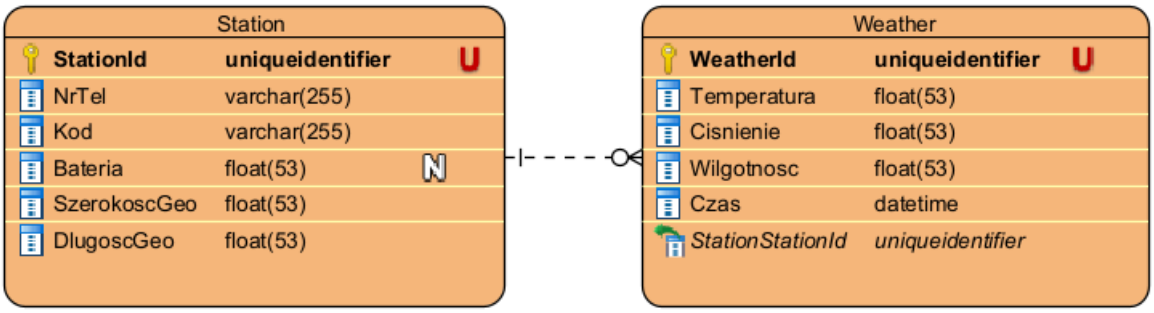
API będzie obsługiwać 1 zapytanie http POST dla stacji nadających informacje oraz 3 zapytania http GET dla aplikacji mobilnej. API składa się z serwera, back-endu napisanego w C# oraz bazy danych MS SQL. Serwer został stworzony na platformie Azure, back-end i baza danych zostały stworzone w środowisku .NET, a następnie zhostowane na platformę.

5.2 Baza danych

Baza danych składa się z 2 encji - Station i Weather, które są w relacji jeden do wielu.

5.2.1 Model fizyczny bazy danych

Model fizyczny bazy danych przedstawia strukturę tabel i relacji pomiędzy encjami Station i Weather. Pokazuje, jak dane są przechowywane i powiązane, umożliwiając efektywne zarządzanie informacjami o stacjach nadawczych i pomiarach pogodowych.



Rysunek 8: Model fizyczny

5.2.2 Stworzenie modelu w C#

Modelowanie danych w C# pozwala na odzwierciedlenie struktury bazy danych w postaci klas, co ułatwia zarządzanie danymi oraz ich przetwarzanie w aplikacji. Poniżej przedstawiono definicje klas Station i Weather, które reprezentują odpowiednie encje w bazie danych.

```
public class Station
{
    [Key]
    public Guid StationId { get; private set; }
    public string NrTel { get; set; }
    public string Kod { get; set; }
    public double? Bateria { get; set; }
    public double SzerokoscGeo { get; set; }
    public double DlugoscGeo { get; set; }
}
```

```
        public class Weather
    {
         [Key]
         public Guid WeatherId { get; private set; }
         public double Temperatura {  get; set; }
         public double Cisnienie { get; set; }
         public double Wilgotnosc { get; set; }
         public DateTime Czas {  get; set; }
```

```
9
10          [ForeignKey(nameof(Station))]
11          public Guid StationId {get; set;}
12          public Station Station { get; set; }
13      }
```

5.2.3 Mapowanie obiektowo-relacyjne(ORM)

Używając środowiska .NET stworzono bazę danych na podstawie modeli. Framework automatycznie połączył wygenerowaną bazę danych z programem. Do stworzenia encji został wygenerowany następujący kod:

```
1       CREATE TABLE [dbo].[Station] (
2       [StationId]    UNIQUEIDENTIFIER DEFAULT (newid()) NOT NULL,
3       [NrTel]        NVARCHAR (MAX)   NOT NULL,
4       [Kod]          NVARCHAR (MAX)   NOT NULL,
5       [Bateria]      FLOAT (53)       NULL,
6       [SzerokoscGeo] FLOAT (53)       NOT NULL,
7       [DlugoscGeo]   FLOAT (53)       NOT NULL,
8       PRIMARY KEY CLUSTERED ([StationId] ASC)
9       );
10
11      CREATE TABLE [dbo].[Weather] (
12      [WeatherId]   UNIQUEIDENTIFIER DEFAULT (newid()) NOT NULL,
13      [Temperatura] FLOAT (53)       NOT NULL,
14      [Cisnienie]   FLOAT (53)       NOT NULL,
15      [Wilgotnosc]  FLOAT (53)       NOT NULL,
16      [Czas]        DATETIME2 (7)    NOT NULL,
17      [StationId]   UNIQUEIDENTIFIER DEFAULT (newid()) NOT NULL,
18      PRIMARY KEY CLUSTERED ([WeatherId] ASC),
19      CONSTRAINT [FK_Weather_Station_StationId] FOREIGN KEY ([StationId])
            REFERENCES [dbo].[Station] ([StationId]) ON DELETE CASCADE
20      );
```

5.3 Back-end

Back-end został napisany w języku C#. Obsługuje on 4 zapytania http. Format danych wejściowch w metodzie Post oraz danych wyjściowych w metodach get to JSON.

5.3.1 Metoda Post

Metoda ta jest wykorzystywana przez fizyczne urządzenie, gdy wysyła informacje na serwer. Dodaje ona otrzymane informacje do bazy danych.
Dane wejściowe: StationWeatherRequest

```
1       public class StationWeatherRequest
2   {
3       public string nr_tel { get; set; }
4       public string kod { get; set; }
5       public double? bateria { get; set; }
6       public double temp { get; set; }
7       public double press { get; set; }
8       public double humi { get; set; }
```

```
        public double lat { get; set; }
        public double longi { get; set; }
    }
```

Dane wyjściowe: Status - kod, który informuje czy operacja się powiodła.
Kod metody:

```
    [HttpPost]
public async Task<IActionResult> Post([FromBody] StationWeatherRequest
    request)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState); // Return validation errors to
            the client
    }

    string nr_tel = request.nr_tel;
    string kod = request.kod;
    double? battery = request.bateria;
    double temp = request.temp;
    double press = request.press;
    double humi = request.humi;
    double lat = request.lat;
    double longi = request.longi;

    if (string.IsNullOrEmpty(nr_tel) || string.IsNullOrEmpty(kod))
    {
        return BadRequest("Invalid data: nr_tel and kod must be
            provided.");
    }

    // Jezeli w bazie nie ma stacji
    Guid stationId = Guid.NewGuid();
    if (!(_context.Station.Any(s => s.NrTel == nr_tel && s.Kod == kod))
        )
    {
        Station station = new Station(stationId, nr_tel, kod, battery,
            lat, longi);
        _context.Add(station);
        await _context.SaveChangesAsync();
    }
    else
    {
        var updateData = _context.Station.FirstOrDefault(s => s.NrTel
            == nr_tel && s.Kod == kod);
        if (updateData == null)
        {
            // error
            return NotFound("updateData was not found");
        }
        updateData.Bateria = battery;
        updateData.SzerokoscGeo = lat;
        updateData.DlugoscGeo = longi;
```

```
42          await _context.SaveChangesAsync();
43      }
44
45
46      var record = _context.Station.FirstOrDefault(s => s.NrTel == nr_tel
            && s.Kod == kod);
47      if (record == null)
48      {
49          // error
50          return NotFound("record was not found");
51      }
52      stationId = record.StationId;
53      DateTime czas = DateTime.Now;
54      Weather weather = new Weather(Guid.NewGuid(),temp, press, humi,
           czas, stationId);
55      _context.Add(weather);
56      await _context.SaveChangesAsync();
57
58
59      return Ok();
60  }
```

5.3.2 Metoda GetLastWeather

Metoda ta jest wykorzystywana przez aplikację mobilną w celu pozyskania informacji o ostatniej pogodzie w danej stacji.
Dane wejściowe:

- numer telefonu stacji
- kod stacji

Dane wyjściowe: WeatherResponse

```
1       public class WeatherResponse
2   {
3       public double? battery {  get; set; }
4       public double temp {get; set;}
5       public double press { get; set;}
6       public double humi {  get; set;}
7       public DateTime time { get; set;}
8       public double lat {  get; set;}
9       public double longi { get; set;}
10  }
```

Kod metody:

```
1       [HttpGet("{nr_tel}/{kod}")]
2   public IActionResult GetLastWeather(string nr_tel, string kod)
3   {
4       // sprawdzenie czy taka stacja istnieje
5
6       if (!(_context.Station.Any(s => s.NrTel == nr_tel && s.Kod == kod)
           ))
7       {
```

```
8           return NotFound();
9       }
10
11      var record = _context.Station.FirstOrDefault(s => s.NrTel ==
            nr_tel && s.Kod == kod);
12      if (record == null)
13      {
14          // error
15          return NotFound("record was not found");
16      }
17      // znalezc ID tej stacji i zmapowac bateria i polozenie
18      Guid stationId = record.StationId;
19      double? bateria = record.Bateria;
20      double lat = record.SzerokoscGeo;
21      double longi = record.DlugoscGeo;
22
23
24      var lastWeather = _context.Weather
25          .Where(w => w.StationId == stationId)
26          .OrderByDescending(w => w.Czas)
27          .FirstOrDefault();
28      if (lastWeather == null)
29      {
30          // No weather record found
31          return NotFound();
32      }
33      // mapowanie danych z pogody
34      double temp = lastWeather.Temperatura;
35      double press = lastWeather.Cisnienie;
36      double humi = lastWeather.Wilgotnosc;
37      DateTime time = lastWeather.Czas;
38      WeatherResponse weatherResponse = new WeatherResponse(bateria,
            temp, press, humi, time, lat, longi);
39      // zwrocic jako json
40      return Ok(weatherResponse);
41  }
```

5.3.3 Metoda GetLast50Weather

Metoda GetLast50Weather działa identycznie jak metoda GetLastWeather, z wyjątkiem tego, że zamiast zwracania ostatniej zarejestrowanej pogody w stacji zwraca ostatnie 50 rekordów zarejestrowanych w stacji.

5.3.4 Metoda GetStation

Metoda ta jest wykorzystywana przez aplikację mobilną w celu pozyskania informacji o stacji.
Dane wejściowe:

- numer telefonu stacji
- kod stacji

Dane wyjściowe: StationResponse

```
    public class StationResponse
{
    public double? battery { get; set; }
    public double lat { get; set; }
    public double longi { get; set; }
}
```

Kod metody:

```
    [HttpGet("station/{nr_tel}/{kod}")]
public IActionResult GetStation(string nr_tel, string kod)
{
    // sprawdzicz czy taka stacja istnieje

    if (!(_context.Station.Any(s => s.NrTel == nr_tel && s.Kod == kod))
       )
    {
        return NotFound();
    }

    var record = _context.Station.FirstOrDefault(s => s.NrTel == nr_tel
        && s.Kod == kod);
    if (record == null)
    {
        // error
        return NotFound("record was not found");
    }
    // znalezc ID tej stacji i zmapowac bateria i polozenie
    double? bateria = record.Bateria;
    double lat = record.SzerokoscGeo;
    double longi = record.DlugoscGeo;
    StationResponse StationResponse = new StationResponse(bateria, lat,
        longi);
    // zwrocic jako json
    return Ok(StationResponse);
}
```

5.4 Hostowanie

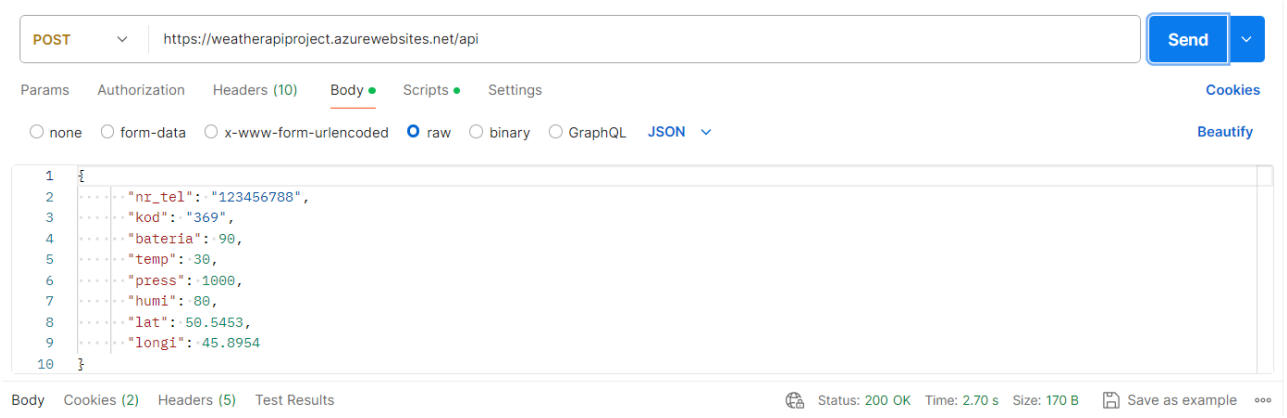
Na platformie Azure został stworzony serwer i baza danych. Strona zawierająca back-end do obsługi zapytań http została opublikowana wykorzystując Azure. Baza danych została połączona ze stroną. Nazwa domeny to weatherapiproject.azurewebsites.net. Platforma pozwala na monitorowanie jak działa aplikacja, co może być wykorzystane do monitorowania zapytań http.

5.5 Użycie i testowanie zapytań http

W tej sekcji omówiono metody używania i testowania zapytań http do interakcji z API. Przedstawiono różne metody, takie jak post do wysyłania danych oraz get do pobierania danych pogodowych z serwera.

5.5.1 Metoda Post

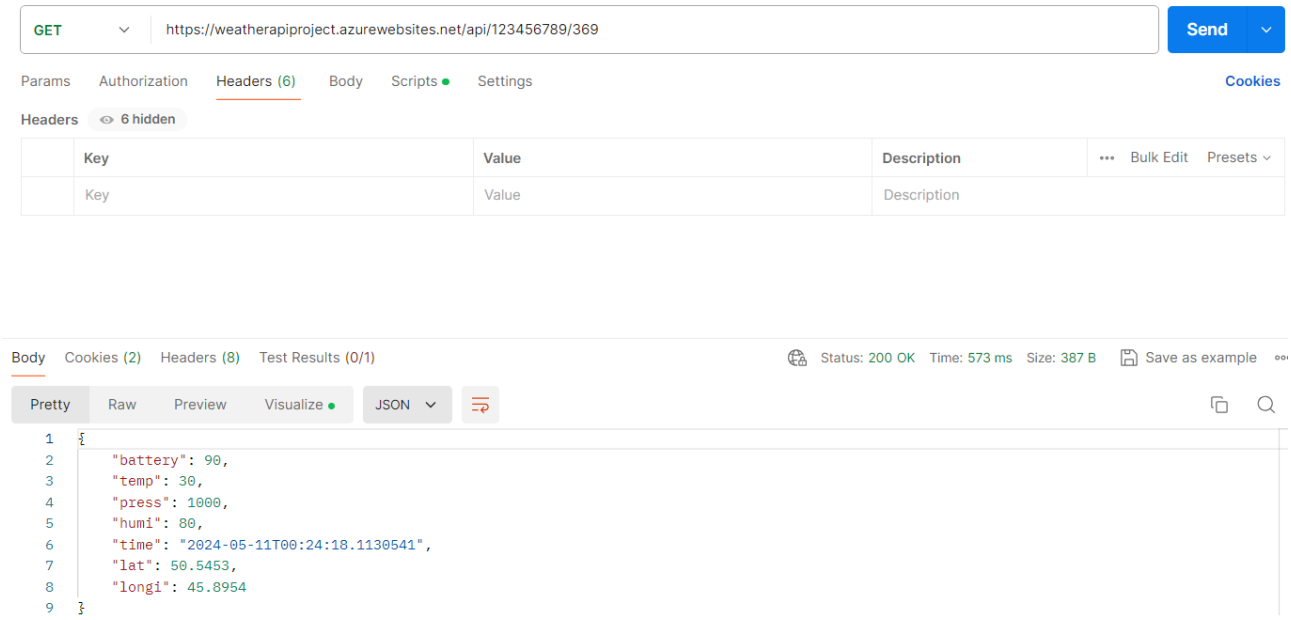
By użyć metody Post należy wysłać zapytanie http na adres: weatherapiproject.azurewebsites.net/api z odpowiednimi danymi.



Rysunek 9: Metoda działa poprawnie

5.5.2 Metoda GetLastWeather

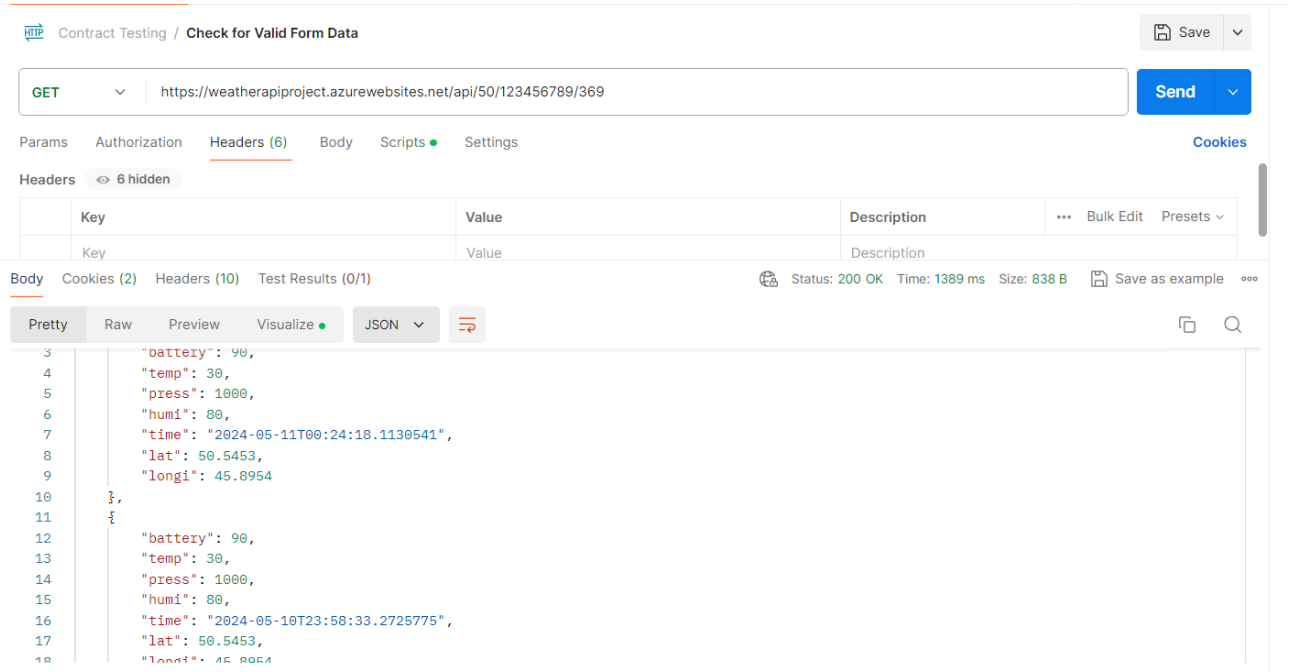
By użyć metody GetLastWeather należy wysłać zapytanie http na adres: weatherapiproject.azurewebsites.net/api/{nr tel}/{kod}



Rysunek 10: Metoda działa poprawnie

5.5.3 Metoda GetLast50Weather

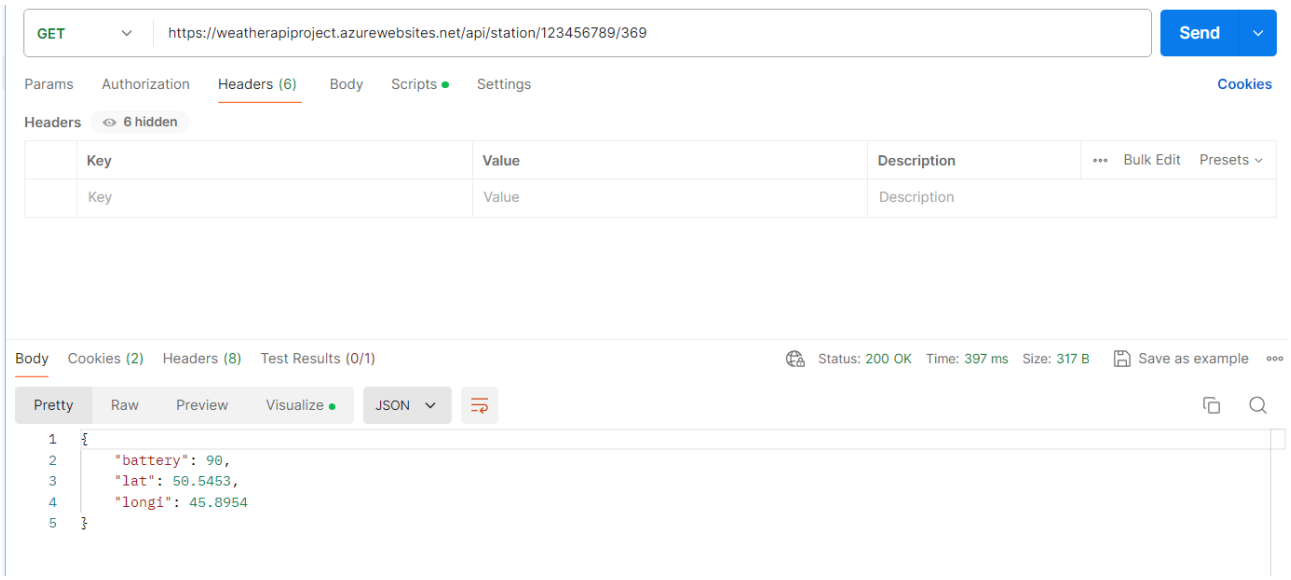
By użyć metody GetLast50Weather należy wysłać zapytanie http na adres: weatherapiproject.azurewebsites.net/api/50/{nr tel}/{kod}



Rysunek 11: Metoda działa poprawnie

5.5.4 Metoda GetStation

By użyć metody GetStation należy wysłać zapytanie http na adres: weatherapiproject.azurewebsites.net/api/station/{nr tel}/{kod}



Rysunek 12: Metoda działa poprawnie

6 Aplikacja mobilna

W tym rozdziale znajdują się szczegółowe informacje na temat części zawierającej aplikację mobilną. Opisano szczegółowo wszystkie ważniejsze funkcjonalności.

6.1 Opis

Aplikacja ma za zadanie w przystępny sposób przedstawiać dane zebrane podczas pracy stacji pogodowych. Dodawanie do listy urządzeń odbywa się poprzez podanie numeru telefonu (moduł GSM) oraz hasła, które jest generowane i stałe dla danego urządzenia (podane np. na obudowie). Dodane urządzenia są przechowywane na telefonie przy pomocy bazy danych SQLite. Aplikacja dostaje informacje o temperaturze, ciśnieniu, wilgotności powietrza oraz lokalizacji tylko dodanych stacji poprzez API. Całość napisana jest w języku Kotlin, korzystając ze środowiska Android Studio.

6.2 Funkcjonalności

1. Możliwość przemieszczania się po aplikacji za pomocą BottomNavigationView.
2. Weryfikacja istnienia stacji poprzez API.
3. Dodawanie istniejącej stacji do bazy danych SQLite.
4. Możliwość wybrania stacji z listy w celu sprawdzenia szczegółów.
5. Możliwość wybrania stacji na mapie (MapView - GoogleMaps) w celu sprawdzenia szczegółów.
6. Zwizualizowanie danych dla danej stacji pobranych przez API (tj. aktualna temperatura, ciśnienie itp.) w szczegółach stacji.
7. Zwizualizowanie danych dla danej stacji poprzez wykres, na podstawie kilkunastu (50) ostatnich zapisów (API).
8. Obsługa trybu ciemnego.

6.3 Szczegółowy opis poszczególnych funkcjonalności

W tym podrozdziale znajdują się szczegółowe opisy wszystkich ważniejszych funkcjonalności.

6.3.1 Obsługa menu

W celu ułatwienia przemieszczania się po aplikacji użyto BottomNavigationView w MainActivity. Do menu dodano cztery fragmenty, które reprezentują kolejno listę stacji, mapę, ustawienia oraz informacje. Dobrano do nich pasujące ikony dostępne w Android Studio. Obsługa wywoływania fragmentów w MainActivity:

```
1       val bottomNav = findViewById<BottomNavigationView>(R.id.
           BottomNavMenu)
2           bottomNav.setOnNavigationItemSelectedListener { item ->
3               when (item.itemId) {
4                   R.id.devices -> showFragment(DevicesFragment(), "
                       DevicesFragment")
5                   R.id.map -> showFragment(MapFragment(), "MapFragment")
6                   R.id.settings -> showFragment(SettingsFragment(), "
                       SettingsFragment")
7                   R.id.info -> showFragment(InfoFragment(), "InfoFragment
                       ")
8               }
9               true
```

```
10          }
```

```
1           private fun showFragment(fragment: Fragment, tag: String) {
2           supportFragmentManager
3               .beginTransaction()
4               .replace(R.id.frameLayout, fragment, tag)
5               .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN)
6               .commit()
7       }
```

6.3.2 Weryfikowanie istnienia stacji

We fragmencie odpowiedzialnym za wyświetlenie listy stacji dodano przycisk, który przenosi do ekranu odpowiedzialnego za dodanie nowej stacji. Należy tam podać kolejno nazwę stacji (Dowolną, informacja dla użytkownika), numer oraz hasło. Po kliknięciu w przycisk następuje wysłanie zapytania http GET z parametrami. W celu obsługi zapytań i asynchroniczności użyto gotowej biblioteki Volley.
Następnie napisano odpowiednią obsługę wyniku na podstawie odpowiedzi z serwera. Jeśli zapytanie powiedzie się urządzenie jest dodawane do bazy SQLite. W przypadku błędu rozróżniane są dwie sytuacje: brak połączenia z Internetem oraz reszta błędów wynikająca z API bądź złego wprowadzenia danych. Wyniki są sygnalizowane za pomocą Toastów. Po otrzymaniu wyniku aplikacja wraca do głównego ekranu.
Dodatkowo dodano okienko z informacją dla użytkownika (dialog), które pojawia się, gdy zapytanie zostanie wysłane i znika po otrzymaniu odpowiedzi. Kod odpowiedzialny za weryfikację stacji:

```
1           fun addStation() {
2           val stationName = findViewById<EditText>(R.id.
               EditTextStationName).text.toString()
3           val number = findViewById<EditText>(R.id.EditTextStationNumber)
               .text.toString()
4           val password = findViewById<EditText>(R.id.
               EditTextStationPassword).text.toString()
5           val url = "https://weatherapiproject.azurewebsites.net/api/
               station/$number/$password"
6           val loadingDialog = Dialog(this)
7
8           //LoadingDialog w oczekiwaniu na odpowiedz
9
10          loadingDialog.setCancelable(false)
11          loadingDialog.setContentView(R.layout.loading_layout)
12          loadingDialog.window!!.setLayout(LinearLayout.LayoutParams.
               WRAP_CONTENT, LinearLayout.LayoutParams.WRAP_CONTENT)
13          loadingDialog.show()
14
15          //Wywolanie API i obsluga odpowiedzi
16
17          val reqQueue: RequestQueue = Volley.newRequestQueue(this)
18          val request = JsonObjectRequest(Request.Method.GET, url, null,
               {result ->
19              val dbRecord = DBWeatherStation(
20                  id = null,
21                  name = stationName,
```

```
                battery = result.getString("battery"),
                mobileNumber = number,
                password = password,
                longitude = result.getString("longi"),
                latitude = result.getString("lat"),
                lastUpdate = "No information"
            )
            Log.i("USER_LOG", result.toString())
            db.insertData(dbRecord)
            Toast.makeText(this, "Device added.", Toast.LENGTH_SHORT).
               show()

            loadingDialog.dismiss()
            finish()

        }, {error ->

            // Obsluga bledow

            Log.e("USER_LOG", error.toString())
            if (error.toString().contains("NoConnectionError")) {
                Toast.makeText(this, "No connection. Check your
                   Internet connection.", Toast.LENGTH_LONG).show()
            }
            else {
                Toast.makeText(this, "Device not found or something
                   went wrong. Try again later.", Toast.LENGTH_LONG).
                   show()
            }
            loadingDialog.dismiss()
            finish()
        })
        reqQueue.add(request)
    }
```

6.3.3 Baza danych SQLite

W celu stworzenia bazy danych napisano odpowiednią klasę, która obsługuje funkcje CRUD.

```
    class DBHandler(context: Context) : SQLiteOpenHelper(context,
       DATABASE_NAME, null, DATABASE_VERSION)  {
    companion object {
        private const val DATABASE_NAME = "weather_stations.db"
        private const val DATABASE_VERSION = 1
        private const val TABLE_NAME = "weather_stations"
        private const val COLUMN_ID = "id"
        private const val COLUMN_NAME = "name"
        private const val COLUMN_MOBILE_NUMBER = "mobile_number"
        private const val COLUMN_PASSWORD = "password"
        private const val COLUMN_BATTERY = "battery"
        private const val COLUMN_LONGITUDE = "longitude"
        private const val COLUMN_LATITUDE = "latitude"
        private const val COLUMN_LAST_UPDATE = "last_update"
    }
```

```
15
16      //Tworzenie tabeli
17
18      override fun onCreate(db: SQLiteDatabase?) {
19          val createTableQuery =
20              "CREATE TABLE $TABLE_NAME " +
21                      "($COLUMN_ID INTEGER PRIMARY KEY, " +
22                      "$COLUMN_NAME TEXT, " +
23                      "$COLUMN_MOBILE_NUMBER TEXT, " +
24                      "$COLUMN_PASSWORD TEXT, " +
25                      "$COLUMN_BATTERY TEXT," +
26                      "$COLUMN_LONGITUDE TEXT, " +
27                      "$COLUMN_LATITUDE TEXT, " +
28                      "$COLUMN_LAST_UPDATE TEXT)"
29          db?.execSQL(createTableQuery)
30      }
31
32      //Aktualizacja struktury tabeli
33
34      override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int,
            newVersion: Int) {
35          val dropTableQuery = "DROP TABLE IF EXISTS $TABLE_NAME"
36          db?.execSQL(dropTableQuery)
37          onCreate(db)
38      }
39
40      //Dodatnie danych do tabeli
41
42      fun insertData(dbWeatherStation: DBWeatherStation) {
43          val db = writableDatabase
44          val values = ContentValues().apply {
45              put(COLUMN_NAME, dbWeatherStation.name)
46              put(COLUMN_MOBILE_NUMBER, dbWeatherStation.mobileNumber)
47              put(COLUMN_PASSWORD, dbWeatherStation.password)
48              put(COLUMN_BATTERY, dbWeatherStation.battery)
49              put(COLUMN_LONGITUDE, dbWeatherStation.longitude)
50              put(COLUMN_LATITUDE, dbWeatherStation.latitude)
51              put(COLUMN_LAST_UPDATE, dbWeatherStation.lastUpdate)
52          }
53          db.insert(TABLE_NAME, null, values)
54          db.close()
55      }
56
57      //Zwracanie wszystkich stacji z tabeli
58
59      fun getAllStations() : List<DBWeatherStation> {
60          val factsList = mutableListOf<DBWeatherStation>()
61          val db = readableDatabase
62          val query = "SELECT * FROM $TABLE_NAME"
63          val cursor = db.rawQuery(query, null)
64
65          while (cursor.moveToNext()) {
66              val id = cursor.getInt(cursor.getColumnIndexOrThrow(
```

```
                COLUMN_ID))
67              val name = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_NAME))
68              val number = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_MOBILE_NUMBER))
69              val password = cursor.getString(cursor.
                    getColumnIndexOrThrow(COLUMN_PASSWORD))
70              val battery = cursor.getString(cursor.getColumnIndexOrThrow
                    (COLUMN_BATTERY))
71              val long = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LONGITUDE))
72              val lat = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LATITUDE))
73              val last = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LAST_UPDATE))
74
75              val dbRecord = DBWeatherStation(id, name, number,password,
                    battery, long, lat, last)
76              factsList.add(dbRecord)
77          }
78          cursor.close()
79          db.close()
80          return factsList
81      }
82
83      //Usuwanie calej zawartosci
84
85      fun clearDatabase() {
86          val db = writableDatabase
87          val clearQuery = "DELETE FROM $TABLE_NAME"
88          db.execSQL(clearQuery)
89      }
90
91      //Zwracanie specyficznej stacji
92
93      fun getRecord(stationID: String) : DBWeatherStation? {
94          val db = readableDatabase
95          val query = "SELECT * FROM $TABLE_NAME WHERE $COLUMN_ID = ?"
96          val cursor = db.rawQuery(query, arrayOf(stationID))
97
98          if (cursor.moveToFirst()) {
99              val id = cursor.getInt(cursor.getColumnIndexOrThrow(
                    COLUMN_ID))
100             val name = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_NAME))
101             val number = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_MOBILE_NUMBER))
102             val password = cursor.getString(cursor.
                    getColumnIndexOrThrow(COLUMN_PASSWORD))
103             val battery = cursor.getString(cursor.getColumnIndexOrThrow
                    (COLUMN_BATTERY))
104             val long = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LONGITUDE))
```

```
105             val lat = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LATITUDE))
106             val last = cursor.getString(cursor.getColumnIndexOrThrow(
                    COLUMN_LAST_UPDATE))
107
108             cursor.close()
109             db.close()
110
111             return DBWeatherStation(id, name, number, password, battery
                    , long, lat, last)
112         } else {
113             cursor.close()
114             db.close()
115             return null
116         }
117     }
118
119     //Usuwanie specyficznej stacji
120
121     fun deleteStation(id: Int?) {
122         val db = writableDatabase
123         val where = "$COLUMN_ID = ?"
124         val args = arrayOf(id.toString())
125         db.delete(TABLE_NAME, where, args)
126         db.close()
127     }
128
129     //Zaktualizowanie informacji specyficznej stacji
130
131     fun updateStation(id: Int?, bat: String, long: String, lat: String,
            time: String) {
132         val db = writableDatabase
133         val values = ContentValues().apply {
134             put(COLUMN_BATTERY, bat)
135             put(COLUMN_LONGITUDE, long)
136             put(COLUMN_LATITUDE, lat)
137             put(COLUMN_LAST_UPDATE, time)
138         }
139         val where = "$COLUMN_ID = ?"
140         val args = arrayOf(id.toString())
141         db.update(TABLE_NAME, values, where, args)
142         db.close()
143     }
144 }
```

Baza danych przechowuje informacje o stacji takie jak nazwa, numer, hasło, data ostatniej aktualizacji, stan baterii. Są to informacje przydatne do wyświetlenia na liście oraz potrzebne do pozostałych zapytań http. Po napisaniu podstawowych funkcji CRUD do obsługi bazy, dodałem kilka dodatkowych, które przydają się na kolejnych ekranach.

6.3.4 Lista stacji

Listę stacji obsługuje element RecyclerView. Do odpowiedniego wyświetlania jego elementów napisano osobną klasę. Dodatkowo do każdego elementu dodano możliwość kliknięcia, które

przekierowuje do kolejnego ekranu ze szczegółami, wysyłając do niego informacje o ID naciśniętej stacji. Fragment z listą stacji:

```
class DevicesFragment : Fragment(), DBAdapter.OnItemClickListener {

private lateinit var recyclerView: RecyclerView
private lateinit var dbAdapter: DBAdapter
private lateinit var db: DBHandler

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    db = DBHandler(requireContext())
    dbAdapter = DBAdapter(db.getAllStations(), this)
}

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_devices,
       container, false)
    context?.theme?.applyStyle(R.style.Theme_WeatherStationApp,
       true)

    recyclerView = view.findViewById(R.id.stationsRecyclerView)
    recyclerView.layoutManager = LinearLayoutManager(requireContext
       ())

    updateListData()

    val buttonAddStation = view.findViewById<View>(R.id.
       buttonAddStationActivity)
    buttonAddStation.setOnClickListener {
        addStationActivity(it)
    }

    return view
}

override fun onResume() {
    super.onResume()
    dbAdapter.refreshData(db.getAllStations())
}

//Zaktualizowanie RecyclerView

private fun updateListData() {
    recyclerView.adapter = dbAdapter
    Log.i("USER_LOG", "List updated")
}

override fun onItemClick(station: DBWeatherStation) {
```

```
48          val intent = Intent(requireContext(), StationActivity::class.
               java).apply {
49              putExtra("STATION_ID", station.id.toString())
50          }
51          startActivity(intent)
52      }
53
54      //Przekierowanie do Activity obslugujacego dodanie stacji
55
56      private fun addStationActivity(view: View) {
57          val intent = Intent(requireContext(), AddStationActivity::class
               .java)
58          startActivity(intent)
59      }
60  }
```

6.3.5 Fragment z mapą - Google Maps

W celu prawidłowego działania MapView trzeba było uzyskać API Key Google. Po odpowiedniej implementacji można oglądać gotową mapę ze znacznikami. Dodano możliwość kliknięcia w znacznik, który przenosi Użytkownika do ekranu ze szczegółami, podobnie jak z elementami z listy. Kod fragmentu z mapą:

```
1       class MapFragment : Fragment(), OnMapReadyCallback {
2
3       private lateinit var db: DBHandler
4       private var mMapView: MapView? = null
5
6       override fun onCreateView(
7           inflater: LayoutInflater, container: ViewGroup?,
8           savedInstanceState: Bundle?
9       ): View? {
10          val rootView = inflater.inflate(R.layout.fragment_map,
               container, false)
11          mMapView = rootView.findViewById(R.id.mapView)
12          mMapView?.onCreate(savedInstanceState)
13          mMapView?.getMapAsync(this)
14
15          db = DBHandler(requireContext())
16
17          return rootView
18      }
19
20      override fun onResume() {
21          super.onResume()
22          mMapView?.onResume()
23      }
24
25     override fun onPause() {
26          super.onPause()
27          mMapView?.onPause()
28      }
29
```

```
30      override fun onDestroy() {
31          super.onDestroy()
32          mMapView?.onDestroy()
33      }
34
35      override fun onLowMemory() {
36          super.onLowMemory()
37          mMapView?.onLowMemory()
38      }
39
40      override fun onMapReady(googleMap: GoogleMap) {
41          val stations : List<DBWeatherStation> = db.getAllStations()
42          val builder = LatLngBounds.Builder()
43
44          //Dodanie znacznikow wszystkich stacji
45
46          if (stations.isNotEmpty()) {
47              for (station in stations) {
48                  val location = LatLng(station.latitude.toDouble(),
                        station.longitude.toDouble())
49                  val markerOptions =
50                      MarkerOptions().position(location).title(station.id
                            .toString()).snippet(station.name)
51                  googleMap.addMarker(markerOptions)
52                  builder.include(location)
53              }
54
55              //Optymalne ustawienie kamery
56              googleMap.moveCamera(CameraUpdateFactory.newLatLngBounds(
                    builder.build(), 300))
57
58              //Ustawienie mozliwosci nacisniecia na przycisk
59
60              googleMap.setOnMarkerClickListener { marker ->
61                  val intent = Intent(requireContext(), StationActivity::
                        class.java).apply {
62                      putExtra("STATION_ID", marker.title)
63                  }
64                  startActivity(intent)
65                  true
66              }
67          }
68      }
69  }
```

6.3.6 Wizualizacja danych

Po stworzeniu ekranu ze szczegółami danej stacji aplikacja szuka w bazie stacji o otrzymanym ID. Następnie na podstawie informacji takich jak numer oraz hasło wysyła zapytanie http. Tutaj też występuje okienko (dialog) informujące użytkownika o trwaniu pobierania danych. Podobnie jak poprzednio rozróżniam sytuację braku połączenia oraz innych błędów. W przypadku błędu aplikacja cofa nas do głównego ekranu. Jeśli dane zostaną pobrane są one wyświetlane (temperatura etc.) oraz aktualizowane w bazie (stan baterii, pozycja). Dodatkowo dodano ikony w

różnych kolorach, które mają symbolizować dane (temperatura, ciśnienie, wilgotność). Funkcja odpowiedzialna za pobranie informacji z API:

```
    fun getStationInfo() {
        val url = "https://weatherapiproject.azurewebsites.net/api/${
           station.mobileNumber}/${station.password}"
        val reqQueue: RequestQueue = Volley.newRequestQueue(this)
        val loadingDialog = Dialog(this)

        //LoadingDialog w oczekiwaniu na odpowiedz

        loadingDialog.setCancelable(false)
        loadingDialog.setContentView(R.layout.loading_layout)
        loadingDialog.window!!.setLayout(LinearLayout.LayoutParams.
           WRAP_CONTENT, LinearLayout.LayoutParams.WRAP_CONTENT)
        loadingDialog.show()

        //Wyslanie zapytania i obsluga odpowiedzi

        val request = JsonObjectRequest(Request.Method.GET, url, null,
           { result ->

            val localDateTime = LocalDateTime.parse(result.getString("
               time"))
            val formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:
               mm:ss")
            val outputDate = formatter.format(localDateTime)

            var pressure = result.getString("press").toDouble()
            pressure *= 0.01


            findViewById<TextView>(R.id.stationTextView).text = station
               .name
            findViewById<TextView>(R.id.textViewTemperature).text = "${
               String.format("%.2f", result.getString("temp").toDouble
               ())}\u00B0C"
            findViewById<TextView>(R.id.textViewPressure).text = "${
               String.format("%.2f", pressure)} hPa"
            findViewById<TextView>(R.id.textViewHumidity).text = "${
               String.format("%.2f", result.getString("humi").toDouble
               ())}%"
            findViewById<TextView>(R.id.lastUpdateTextView).text = "
               Last update:\n$outputDate"
            findViewById<TextView>(R.id.batteryStatusTextView).text = "
               Battery:\n${result.get("battery")}%"

            db.updateStation(station.id, result.getString("battery"),
               result.getString("longi"), result.getString("lat"),
               outputDate)

            //Przekazanie informacji o stacji do fragmentu z wykresem

```

```
            val plotFragment = PlotFragment()
            val bundle = Bundle().apply {
                putString("STATION_NUMBER", station.mobileNumber)
                putString("STATION_PASSWORD", station.password)
            }
            plotFragment.arguments = bundle

            supportFragmentManager.beginTransaction()
                .replace(R.id.fragmentContainerView2, plotFragment)
                .commit()

            Log.i("USER_LOG", result.toString())
            loadingDialog.dismiss()

        }, {error ->

            //Obsluga bledow

            Log.e("USER_LOG", error.toString())
            if (error.toString().contains("NoConnectionError")) {
                Toast.makeText(this, "No connection. Check your
                    Internet connection.", Toast.LENGTH_LONG).show()
            }
            else {
                Toast.makeText(this, "Something went wrong. Try again
                    later.", Toast.LENGTH_LONG).show()
            }
            loadingDialog.dismiss()
            finish()
        })
        reqQueue.add(request)
    }
```

Do wyświetlenia wykresu zastosowano oddzielny fragment, który zawarty jest na ekranie. Przy tworzeniu fragmentu przekazywany jest do niego numer oraz hasło stacji. Następnie fragment pobiera listę z kolejnego zapytania http kilkunastu ostatnich zapisów. Każdy z nich jest przechowywany w specjalnej klasie.

```
    private fun getStationData(view: View) {
        val url = "https://weatherapiproject.azurewebsites.net/api/50/$
            {stationNumber}/${stationPassword}"

        //Wyslanie zapytania i obsluga odpowiedzi

        val reqQueue: RequestQueue = Volley.newRequestQueue(
            requireContext())
        val request = JsonArrayRequest(Request.Method.GET, url, null, {
            result ->

            //Tworzenie listy z danymi

            for (i in 0 until result.length()) {
                val station = result.getJSONObject(i)
                val temp = station.getString("temp")
```

```
                var pressure = station.getString("press").toDouble()
                pressure *= 0.01
                val humi = station.getString("humi")
                val time = station.getString("time")

                val newData = DBStationData(temp, pressure.toString(),
                    humi, time)
                dataList.add(newData)
            }

            dataList.reverse()

            //Dodanie mozliwosci klikniecia w przycisk w celu zmiany
                wykresu

            view.findViewById<Button>(R.id.buttonPlotTemperature).
                setOnClickListener {
                 setTemperature()
            }
            view.findViewById<Button>(R.id.buttonPlotPressure).
                setOnClickListener {
                 setPressure()
            }
            view.findViewById<Button>(R.id.buttonPlotHumi).
                setOnClickListener {
                 setHumi()
            }

            setTemperature()
            Log.i("USER_LOG", result.toString())

        }, {error ->
            Log.e("USER_LOG", error.toString())
        })
        reqQueue.add(request)
    }
```

Wykres jest realizowany za pomocą gotowej biblioteki MPAndroidChart. Przykład tworzenia wykresu z temperaturą:

```
    private fun setTemperature() {
        val entries = mutableListOf<Entry>()
        var index = 0

        //Dodanie danych do 'wejsc' wykresu

        for (data in dataList) {
            index++
            val temperature = data.temp.toFloatOrNull()
            if (temperature != null) {
                entries.add(Entry(index.toFloat(), temperature))
            }
        }
```

```
14
15          //Kolor tesktu na wykresie uwzgledniajacy tryb ciemny
16
17          val textColor = if (darkMode == 2) Color.WHITE else Color.BLACK
18
19          //Konfiguracja danych i opcji wykresu
20
21          val dataSet = LineDataSet(entries, "Temperature [°C]").apply {
22              axisDependency = YAxis.AxisDependency.LEFT
23              color = resources.getColor(R.color.red, null)
24              setCircleColor(resources.getColor(R.color.red, null))
25              valueTextColor = textColor
26          }
27
28          val lineData = LineData(dataSet)
29          lineChart.data = lineData
30
31          lineChart.xAxis.textColor = textColor
32          lineChart.axisLeft.textColor = textColor
33          lineChart.axisRight.textColor = textColor
34          lineChart.legend.textColor = textColor
35
36          lineChart.invalidate()
37      }
```

Dodano przyciski, które odpowiadają za przełączanie informacji na wykresie pomiędzy temperaturą, ciśnieniem, wilogtnością.
Dodatkowo na dole ekranu dodano przycisk, który umożliwia usunięcie danej stacji. Po jego naciśnięciu wyświetla się okno z potwierdzeniem (dialog) Po potwierdzeniu stacja jest usuwana z bazy, a aplikacja powraca na ekran główny.

6.3.7 Tryb ciemny

Tryb ciemny jest zapisywany poprzez SharedPreferences. Zmienia on wygląd wszystkich ekranów oraz fragmentów aplikacji. Dodano odpowiedni kolor tekstu na wykresie w zależnośći od trybu ciemnego, ponieważ nie zmieniał się on automatycznie. Funkcja sprawdzająca stan zapisanych preferencji:

```
1          private fun isDarkModeEnabled(context: Context): Boolean {
2          val sharedPreferences: SharedPreferences = context.
               getSharedPreferences("user_preferences", Context.
               MODE_PRIVATE)
3          return sharedPreferences.getBoolean("dark_mode", false)
4      }
```

Funckja zapisująca preferencje:

```
1          private fun saveDarkModePreference(context: Context, isEnabled:
               Boolean) {
2          val sharedPreferences: SharedPreferences = context.
               getSharedPreferences("user_preferences", Context.
               MODE_PRIVATE)
3          val editor = sharedPreferences.edit()
4          editor.putBoolean("dark_mode", isEnabled)
5          editor.apply()
```

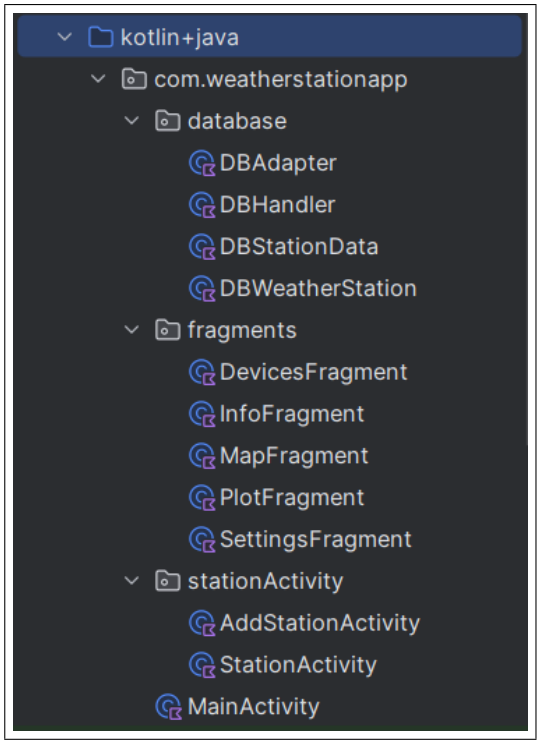
```
    }
```

Funkcja obsługująca przełącznik do preferencji:

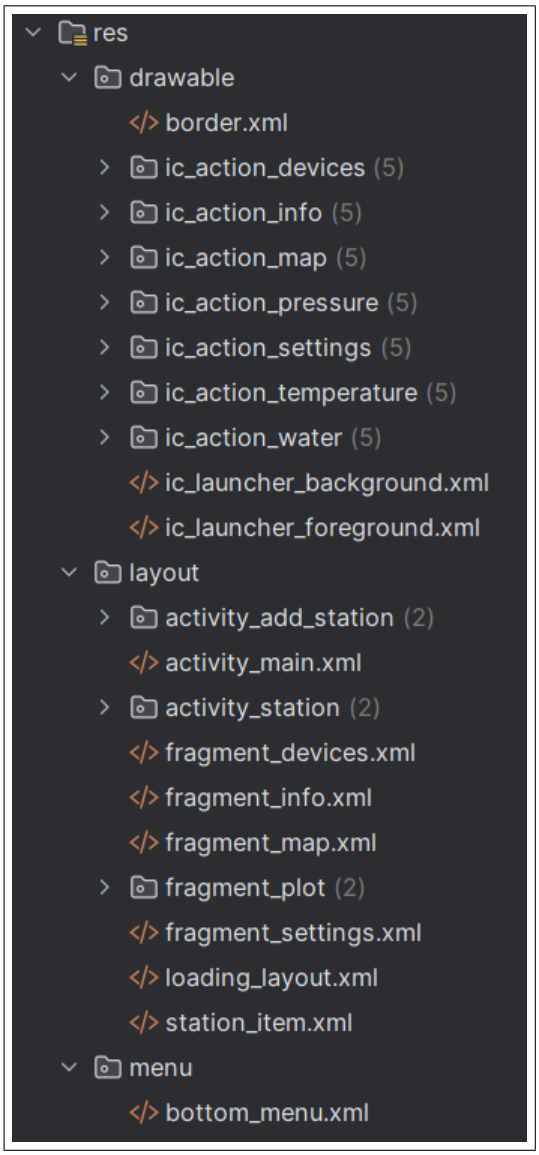
```
        private fun toggleDarkMode(context: Context, isEnabled: Boolean
            ) {
        if (isEnabled) {
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.
                MODE_NIGHT_YES)
        } else {
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.
                MODE_NIGHT_NO)
        }
        saveDarkModePreference(context, isEnabled)
    }
```

6.4 Lista klas oraz szablonów

Poszczególne klasy oraz szablony można wyczytać ze zrzutów ekranu poniżej:



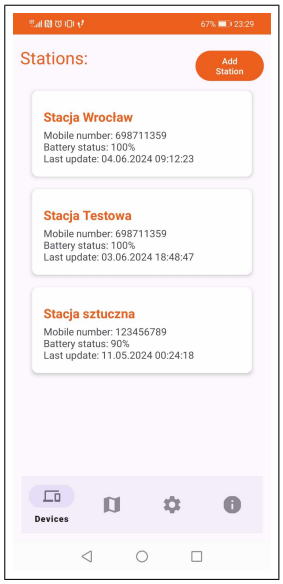
(a) 1. Lista klas



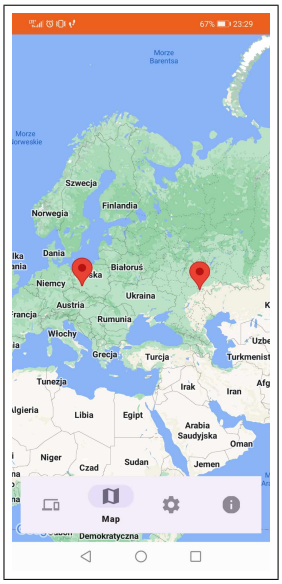
(b) 2. Lista szablonów

6.5 Wygląd aplikacji

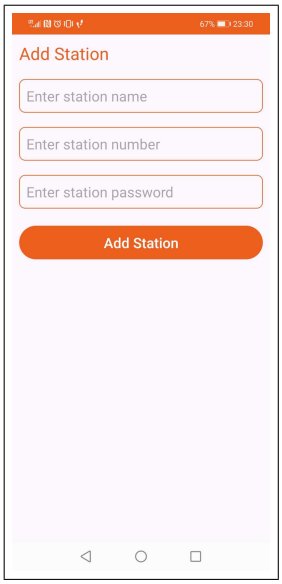
Przykłady ważniejszych ekranów i fragmentów aplikacji (zrzuty ekranu z fizycznego telefonu):



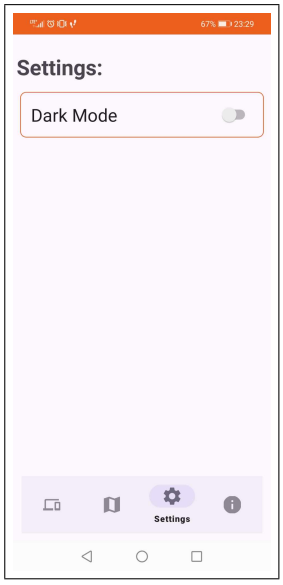
(a) 1. Lista



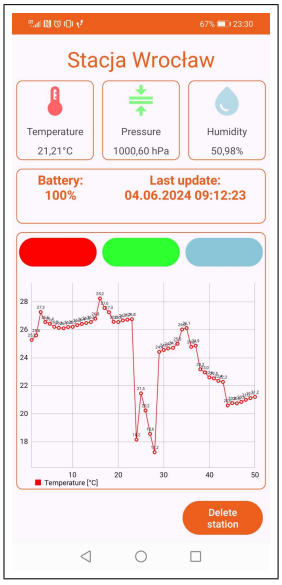
(b) 2. Mapa



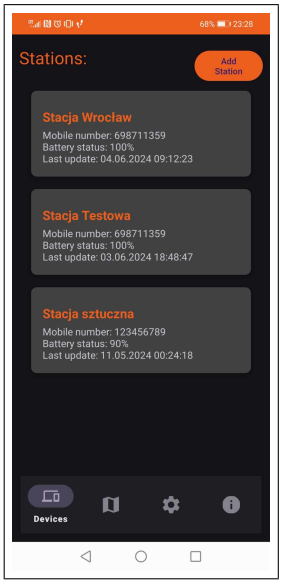
(c) 3. Dodawanie



(a) 4. Opcje



(b) 5. Szczegóły



(c) 6. Motyw ciemny

(a) 7. Szczegóły poziomo

7 Testy

Po podłączeniu gotowego urządzenia sprawdzano poprawność działania poszczególnych modułów. Temperatura, ciśnienie oraz wilgotność powietrza były poprawnie pobierane z czujników. Następnie zweryfikowano, czy moduł GSM wysyła dane do serwera.

Serwer poprawnie odbierał dane, co można było zaobserwować poprzez dodawanie ich do bazy MS SQL. Zweryfikowano działanie API poprzez wysyłanie przykładowych zapytań HTTP na zwykłej przeglądarce internetowej.

Następnie zweryfikowaliśmy działanie poszczególnych funkcjonalności aplikacji mobilnej. Wszystkie opcje można było łatwo sprawdzić poprzez zwykłe użytkowanie aplikacji na fizycznym telefonie.

Podsumowując cały projekt działa poprawnie i według założeń. Nie zaobserwowano żadnych niespójności oraz błędów podczas testów.

8 Podsumowanie

W trakcie prac nad projektem napotkaliśmy na różne wyzwania, które wymagały od nas elastyczności i innowacyjnego podejścia. Ostatecznie udało nam się stworzyć funkcjonalny system, który spełnia założone cele i może być rozszerzany o kolejne stacje pogodowe, dostosowując się do potrzeb użytkowników.

Projekt stacji pogodowej IoT jest przykładem zastosowania nowoczesnych technologii w praktyce, demonstrując, jak inteligentne rozwiązania mogą przyczynić się do poprawy codziennego życia. Mamy nadzieję, że nasza praca przyczyni się do dalszego rozwoju technologii IoT i znajdzie zastosowanie w wielu dziedzinach.

9 Bibliografia

Urządzenie:

- https://files.seeedstudio.com/wiki/Grove-AHT20_I2C_Industrial_Grade_Temperature_and_Hun datasheet-2020-4-16.pdf - AHT20 dokumentacja
- https://github.com/ciastkolog/BMP280_STM32 - bmp280 biblioteka
- https://www.digikey.jp/htmldatasheets/production/1833952/0/0/1/sim800-series-at-command-manual.html - komendy AT

API:

- https://medium.com/@syedhassam2001/creating-and-deploying-net-7-minimal-api-and-mysql-database-on-azure-91f3f6828fb8
- https://www.youtube.com/watch?v=PmDJIooZjBE

Aplikacja mobilna:

- https://www.youtube.com/watch?v=gb3EOEhudeo

- https://www.youtube.com/watch?v=ceqcalf5_Is
- https://www.youtube.com/watch?v=dy2F1DqBcKw
- https://www.youtube.com/watch?v=BVAslimaGSk
- https://stackoverflow.com/questions/49821027/cant-go-from-one-activity-to-another-in-recycler-view/49821315#49821315
- https://www.youtube.com/watch?v=_gpreGNtNCM
- https://developers.google.com/maps/documentation/android-sdk/marker?hl=pl
- https://www.youtube.com/watch?v=6IX2AIRFqLI
- https://stackoverflow.com/questions/59340099/how-to-set-confirm-delete-alertdialogue-box-in-kotlin