Wrocław University of Science and Technology
Faculty of Information and Communication Technology

Field of study: **TEL**

BACHELOR THESIS

Web weather service

Sergei Vorobev

Supervisor
PhD, Pawel Gluchowski

Keywords: Raspberry Pi, weather service, web server, database, python

WROCŁAW 2022

Synopsis

The aim of this thesis is to create a model and Python implementation of a web service providing data from a weather station. The scope of work was determined based on an analysis of systems available on the market, offering solutions to similar problems. The article presents the process of creating a weather station based creating weather station hardware, creating a database with a web server and a web application to display weather data. The article ends with a summary that concludes that the conclusions drawn from the code development and server integration work.

The weather station and web application for data visualization that is the subject of this thesis work is implemented as part of an individual engineering project by the author of this research.

Abstract

Collecting weather data allows for analysis, which in due course can be used to create forecasts. In addition, up-to-date weather data are important for various economic activities, beginning with agriculture to the building industry. Collecting this data in automatic remote mode is becoming increasingly important. This reduces the human error factor and makes the process cheaper. The purpose of this paper is to propose a solution with the creation of a weather station to collect weather data and transmit them in the form of an intuitive web interface.

The range of work has been defined based on an analysis of the solutions available on the market providing tools that solve similar problems. A requirements analysis was carried out which, in turn, helped in verifying the completeness of the application.

Agenda

1. Introduction

1.1. Solution context

The weather station and web application for data visualization that is the subject of this engineering paper is implemented as part of an individual engineering project by the author of this research. The complete application consists of a Raspberry Pi 4 Model B [1] microcomputer with connected weather sensors, a database, a web server and a web application.

Since the introduction of the Internet of Things (IoT)[2] concept, people have been able to collect weather data and transmit it via the Internet. This has greatly simplified data collection and helped to reduce the costs associated with the process. This area of IoT application has contributed to the growing demand for weather stations using IoT in the last twenty years. Currently, there are quite a lot of such weather stations on the market. This prompted the author to create a similar weather station with web application.

For the basis of the created weather station was chosen Rasberry Pi microcomputer of the latest version - 4. The choice of this component was due to its low cost, popularity on the market and as a consequence its wide support. In addition the choice of this component was due to the use of the programming language Python [3]. Which in turn is widely used in the creation of IoT networks.

Among other components used by the author were weather sensors. For the creation of the database was used sqlite3 [4] library. As a framework for writing the web server application and creating a web application the author used Flask [5].

1.2. Purpose and scope

The main purpose of the work is to create a weather station to collect weather data and transfer them as a web interface.

The scope of work includes the following questions:

- Overview of how the data transfer from the sensors works;
- Creating a working schematic diagram of connecting sensors to a microcomputer;
- Writing Python code for collecting data from the sensors and transferring it into a database;
- Creating a database for collecting data from sensors;

- Creating a web server using the Flask web framework that takes HTTP requests from a web browser and returns HTTP responses, along with the data from the sensors;
- Creating a web application to illustrate the sensor data.

1.3. System architecture

The text of the work is divided into chapters with the following content:

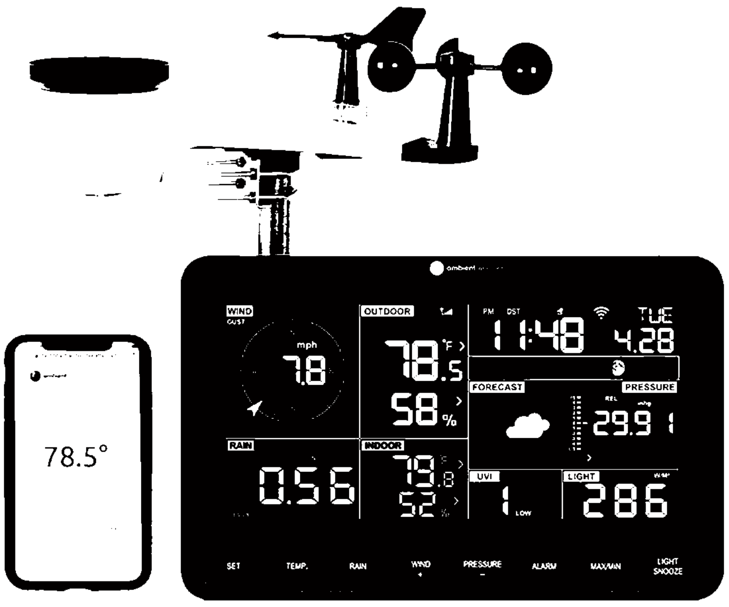
- Chapter 1 presents the solution context, purpose and scope and system architecture;
- Chapter 2 presents analysis of existing solutions;
- Chapter 3 includes a specification and analysis of functional and non-functional requirements;
- Chapter 4 describes overall solution architecture, database model and also describes libraries used in the solution;
- Chapter 5 presents implementation of hardware connection, database creation, web server creation and web application creation;
- Chapter 6 provides functional and non-functional testing of the solution;
- Chapter 7 describes achieved results, advantages and disadvantages of the solution and also what was learned in the process of preparing the solution.

2. Analysis of existing solutions

This chapter presents the existing weather stations on the market, which the author of this thesis was able to find. An attempt has been made to describe in detail the features of the existing solutions. This chapter describes advantages and disadvantages of each of the presented solutions. And in the final section a brief summary of the analysis.

2.1. Ambient Weather Station WS-2902

Ambient Weather's newest Personal Weather Station WS-2902 allows to monitor home and backyard weather conditions with a LCD [ ] color display. Monitor indoor and outdoor conditions, including wind speed, wind direction, rainfall, UV, solar radiation, barometric pressure, indoor/outdoor temperature (F and C), indoor/outdoor humidity, and more. The weather station also calculates dew point, wind chill, and heat index.



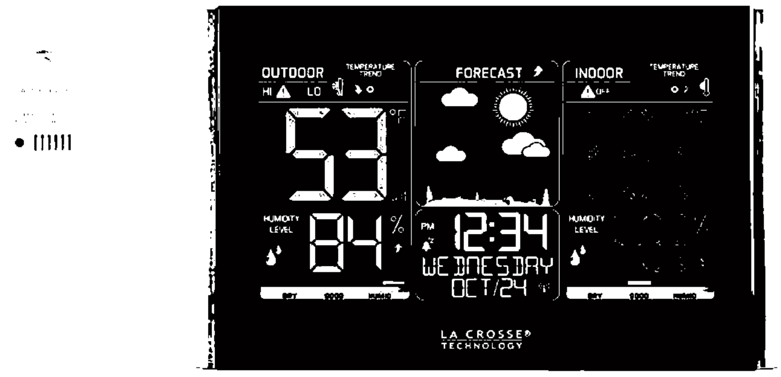
Picture 2.1: Ambient Weather Station WS-2902

Table 2.: Advantages and disadvantages of Ambient Weather Station WS-2902

Advantages	**Disadvantages**
Additional sensors can be added	Display has limited viewing angles
Supports voice assistant services	Complicated setup for sending data to internet
Mobile application available in the Apple App Store and Google Play	Console does not display additional sensors

2.2. La Crosse Technology C85845-INT Weather Station

The La Crosse Technology C85845 weather station includes indoor and outdoor temperature and humidity sensors and a barometer, which allows it to offer a rough preview of your upcoming weather. It doesn't provide the same level of detail as more expensive weather stations, but it is enough to warn a user to grab an umbrella on the way out the door.



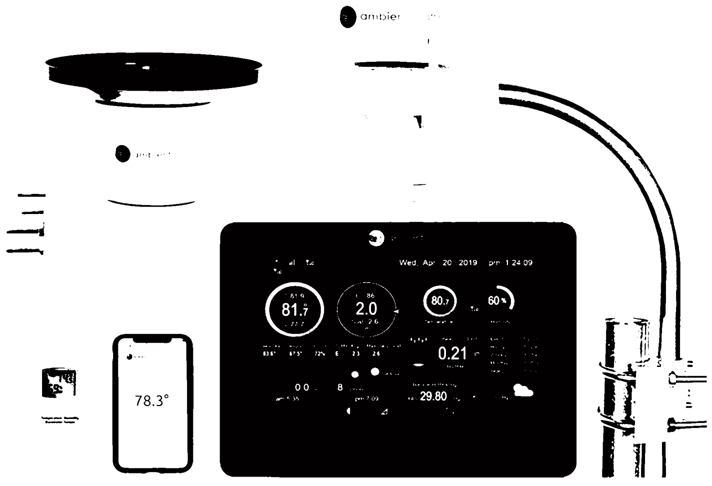
Picture 2.2: La Crosse Technology C85845-INT Weather Station

Table 2.2: Advantages and disadvantages of La Crosse Technology C85845-INT

Advantages	Disadvantages
Color display has a big viewing angles User-friendly setup Low price	Does not support additional sensors Presents only two weather parameters

2.3. Ambient Weather WS-5000

The Ambient Weather WS-5000 is a serious weather station for those who are serious about tracking the weather. This system comes with a full suite of sensors out of the box, including an ultrasonic wind sensor that's faster and more accurate than traditional anemometers, plus a huge variety of add-on sensors, including up to eight thermo-hygrometers and soil moisture sensors, as well as air quality, leak, and lightning detectors. Most of the sensors that come with the WS-5000 are separate units, allowing to place each in its ideal location



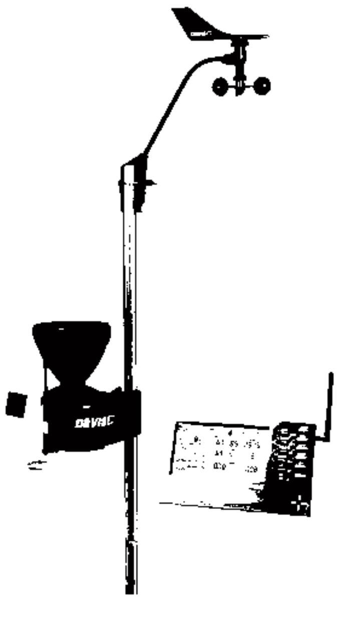
Picture 2.3: Ambient Weather WS-5000

Table 2.3: Advantages and disadvantages of Ambient Weather WS-5000

Advantages	**Disadvantages**
Has many weather sensors	Complicated setup
Support additional sensors	Presents only two weather parameters
Color display has a big viewing angles	High price
Mobile application available in the Apple App Store and Google Play	

2.4. Davis Wireless Vantage Pro2

The Davis Wireless Vantage Pro2 straddles the line between hobbyist home weather stations and the sort of weather stations used in agricultural and commercial applications. While this weather station is more complicated to set up than most of the other options, it offers the precision and reliability that serious hobbyists need and professionals can rely on. It includes separate wind, rain, temperature, and humidity sensors, allowing you to place each sensor exactly where it needs to be for the most accurate readings.



Picture. 2.4: Davis Wireless Vantage Pro2

Table 2.4: Advantages and disadvantages of Davis Wireless Vantage Pro2

Advantages	**Disadvantages**
Has many weather sensors Support additional sensors	Complicated setup Monochrome display High price

2.5. Conclusion

We can see that there are many ready solutions on the weather station market. Some solutions provide the ability to monitor only two weather data, while others show much more data. Not all these stations propose any web service for monitoring weather data.

Most of the commercial proposals mentioned in this dissertation have a high cost, relative to the cost of the weather sensors used in them. The cheapest of these is the La Crosse Technology C85845-INT weather station. It only measures air temperature and humidity. Without deconstructing the station, it can be assumed that it uses a single sensor. This sensor could be the DHT22 temperature and relative humidity sensor [6]. The market value of this sensor does not exceed 20 percent of the stated cost of the entire system. Which indicates the high cost of this weather station, relative to the cost of probably the sensor used in it.

This prompted the author of this thesis to make his own station. A weather station that will feature several advantages:

- Low cost. This will be due to the fact that the proposal will not be commercial, being developed for the market;
- Scalability. A weather station, no matter how large [clarification], cannot compete in scalability with a weather station created using the Raspberry Pi4 microcomputer. This microcomputer can process data up to 30 simultaneously connected devices [7]. This makes it possible to connect all weather sensors available on the market and to submite these data into web application.

3. Specification and analysis of requirements

3.1. System general description

The developed solution is a system consisting of a microcomputer with connected sensors, a database, a web server and a web application to display the weather data.

This solution represents in my opinion a cheap and not complicated approach in its performance. A solution which can be implemented even by a robotics enthusiast. In addition the programming language Python and Sqlite library also have a low level for beginner developers. All this makes the proposed solution "user-friendly" for beginners.

The goal of the work is to achieve the collection, storage and transmission and presentation of real-time weather data in the form of a web application interface. The data collection should be done by sensors. Data storage, reading and transmission should be stable. The web application interface should be intuitive. A detailed list of the functional and non-functional requirements for the solution, together with their descriptions, can be found later in this chapter, in sections 2 and 3 respectively of the Chapter 3.

3.2. Functional requirements

Due to the nature of the software being designed, user stories, presented in the form of lists, were used to describe the functional requirements for the solution. They were grouped according to the underlying functionalities they comprise. Next to each requirement, its priority was also written down, in accordance with the MoSCoW methodology [8] The MoSCoW method is a prioritization technique used in management, business analysis, project management, and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement; it is also known as MoSCoW prioritization or MoSCoW analysis.

The term Moscow itself is an acronym derived from the first letter of each of four prioritization categories: M - Must have, S - Should have, C - Could have, W - Won't have.

Related to general description of the solution of user stories are:

- **Collecting weather data from sensors**

As a user, I would like the data to be collected from the sensors. – Must

As a user, I would like the data to be collected without data loss. – Must

As a user I would like the data to be collected at regular intervals. – Should

- **Save the data to a database**

As a user I would like the data to be saved for later use. - Must

As a user I would like the data to be stored in a single, defined place in the memory of the computer. - Should

As a user I would like to have a safe place to store the data in the database. - Should

- **Read data from the database**

As a user I would like the reading of the data from the database to take not more than 30 ms. - Should

As a user I want to read data from the database without errors and distortions - Must

- **Passing data to the web application interface**

As a user, I would like the data to be transferred to the web application interface. - Must

As a user, I would like the data transfer to the web application interface to take place no longer than 30 ms. - Should

As a user, I would like the transfer of data to the web application interface to be free of errors and distortion. - Must

- **Web application**

As a user, I would like the interface to display sensor data. - Must

As a user, I would like the web page to load no longer than 100 ms. - Should

As a user, I would like the interface to display data from all installed sensors. - Could

As a user, I would like the interface to be intuitive. - Should

3.3. Non-functional requirements

The non-functional requirements for the solution were developed in accordance with the FURPS methodology [9].

3.3.1. Limitations.

Limitations for the microcomputer used:

- Raspbian Buster desktop or Raspbian Buster lite operating system;
- A quad-core processor clocked at 1.5 GHz;
- The minimum amount of microcomputer RAM is 1 GB. The recommended amount of microcomputer RAM is 4 GB.

Limitations for the web application used:

- A web browser from the Chrome, Firefox, or Safari families
- Any computer or mobile operating system capable of

support a web browser.

3.3.2 Transparency and reusability.

The developed solution must be user-friendly for development and use simple templates.

3.3.3. Localization.

The web interface must be available in English, due to the wide spread of the latter.

3.3.4. Consistency.

The developed solution must have similarity between interface elements designed for the same purpose.

3.3.5. Security

The connections between the electronic components in the system must be reliable. This would prevent loss of signal from the sensors.

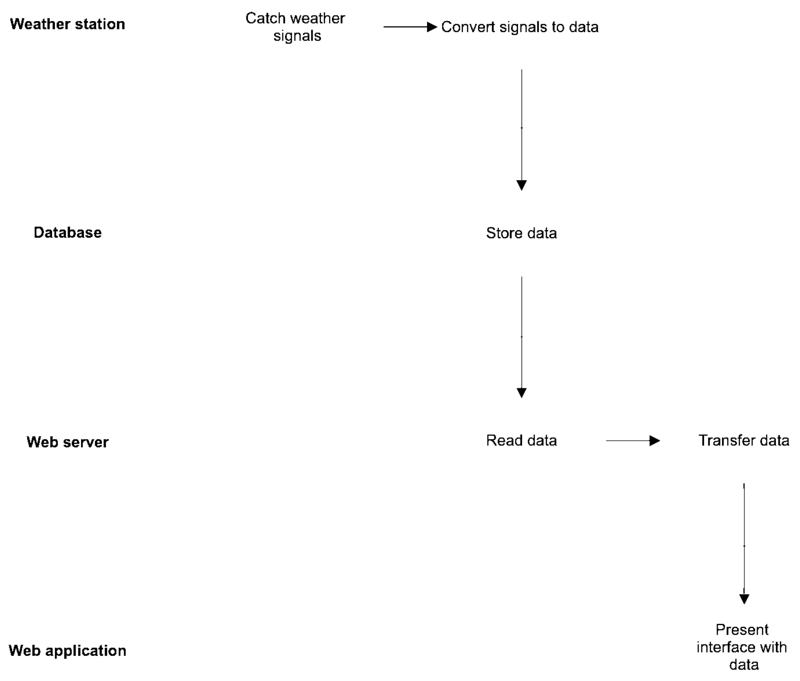
4. Design of the solution

4.1. Overall solution architecture

When developing the project, it was decided to divide it into three components:

- creation of the hardware of the weather station;
- writing software for collecting and storing data from sensors;
- database creation;
- web server development;
- web application development.

The general logic of the solution process is shown in the illustration below.

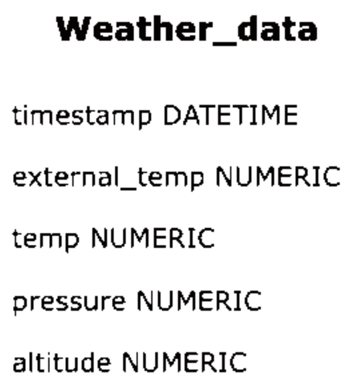


Picture 4.1: Swimline diagram of the solution [10]

4.2. Database model

Database “sensorData.db” will consist of only one “Weather_data” table. It contained the next fields:

- “timestamp” datetime filed to store time data;
- “external_temp” numeric field to store temperature data from external temperature sensor;
- “temp” numeric field to store air temperature data;
- “pressure” numeric field to store pressure data;
- “altitude” numeric field to store altitude data.



Weather_data

timestamp DATETIME

external_temp NUMERIC

temp NUMERIC

pressure NUMERIC

altitude NUMERIC

Picture 4.2: Weather_data table of database

4.3. Libraries used of the solution

In the process of development libraries were used:

- flask - to creating webserver application;
- sqlite3 - to creation database;
- w1thermsensor – for setup ds18b20 temperature sensor;
- bmpsensor – to connect BMP180 temperature sensor;
- matplotlib – to plot graphs.

5. Implementation of the solution

5.1 Connecting sensors to the microcomputer

To implement the hardware part of the project, the author chose the cheapest and most available sensors for measuring temperature and pressure. And the Raspberry Pi4 was chosen as the microcomputer.

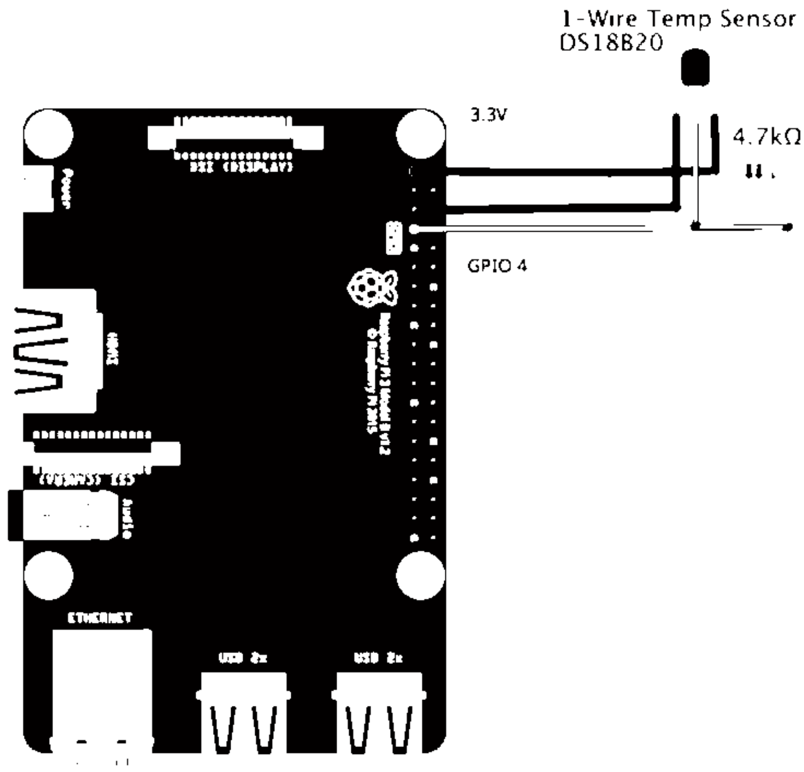
5.1.1 Connecting temperature sensor

The DS18B20 sensor was chosen as the temperature sensor. It is very useful for capturing temperature in wet conditions, for example on humid soil. The sensor is isolated and can take measurements until 125°C.

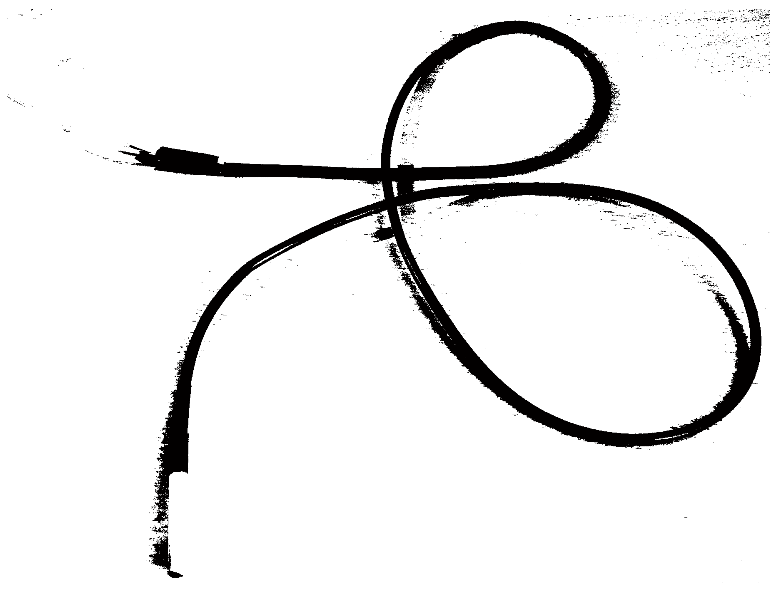
The sensor works from 3.0 to 5.0V, which means that it can be powered directly from the 3.3V provided by one of the Raspberry pins (1 or 17).

The sensor has 3 wires:

- Black: GND;
- Red: VCC;
- Yellow: 1-Wire Data.



Picture 5.1: DS18B20 sensor connection

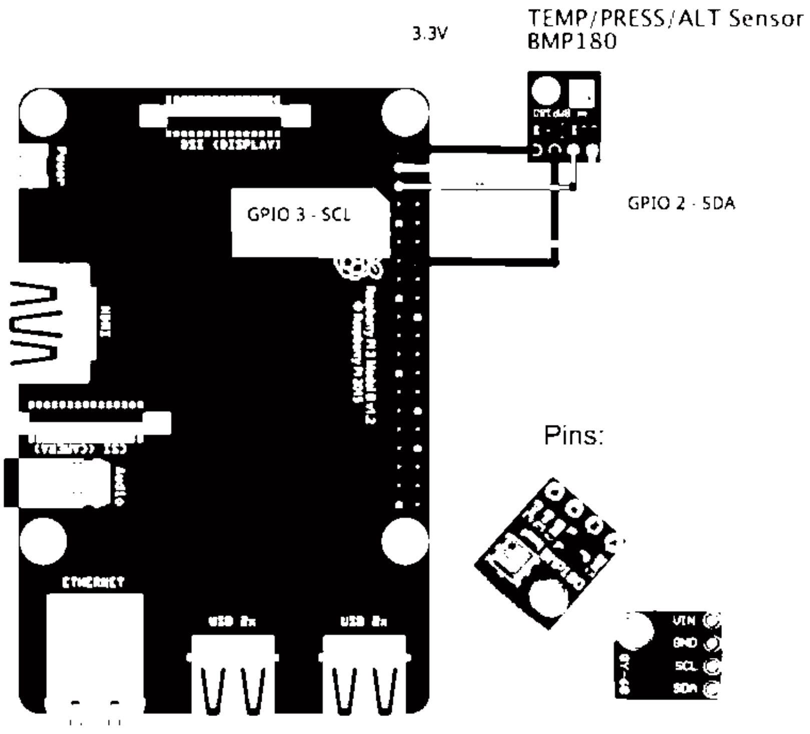


Picture 5.2: DS18B20 sensor

5.1.2. Connecting temperature pressure and altitude sensor

The BMP180 sensor [11] was chosen as the sensor for measuring.

The BMP180 is the successor of the BMP085, a new generation of high precision digital pressure sensors for consumer applications. The ultra-low power, low voltage electronics of the BMP180 is optimized for use in mobile phones, PDAs, GPS navigation devices and outdoor equipment. With a low altitude noise of merely 0.25m at fast conversion time, the BMP180 offers superior performance. The I2C interface allows for easy system integration with a microcontroller. The BMP180 is based on piezo-resistive technology for EMC robustness, high accuracy, and linearity as well as long-term stabil

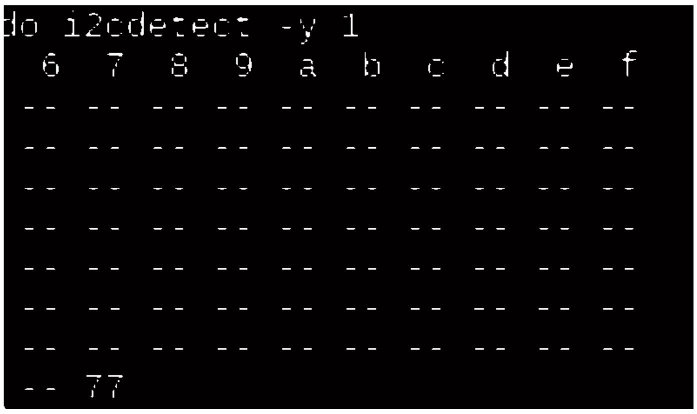


Picture 5.3: BMP180 sensor connection

According to the picture above the next connections were made:

- Vin ==> 3.3V;
- GND ==> GND;
- SCL ==> GPIO 3;
- SDA ==> GPIO 2

To check if the Raspberry Pi sees your BMP180 in the terminal window run the command: “sudo i2cdetect -y 1”. Terminal window will show that the BMP180 is on channel '77'.



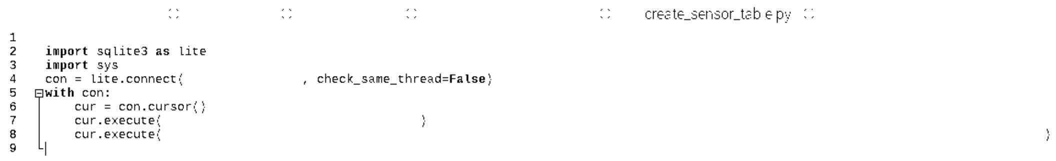
```
do i2cdetect -y 1
  6  7  8  9  a  b  c  d  e  f
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- -- -- -- -- -- -- -- -- --
 -- 77
```

Picture 5.4: Terminal window presents BMP180 on channel '77'

That is mean that Raspberry Pi detect BMP180 sensor.

5.2 Database creation

For storing weather in Raspberry Pi was created a local database. It was created with using SQLite language library. The script is illustrated below.

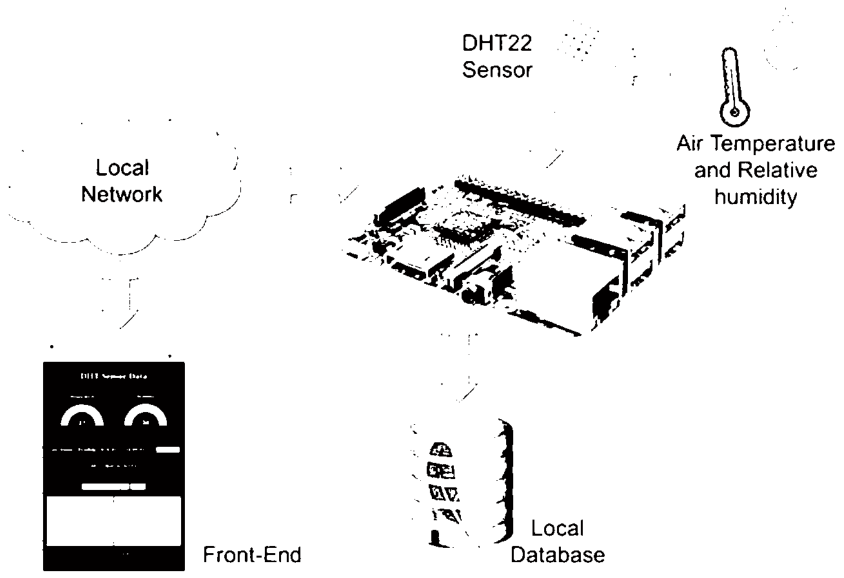


create_sensor_tab e py

```

import sqlite3 as lite
import sys
con = lite.connect(                  , check_same_thread=False)
with con:
    cur = con.cursor()
    cur.execute(                                )
    cur.execute(                                                                   )

```

Picture 5.5: The part of code to create database table

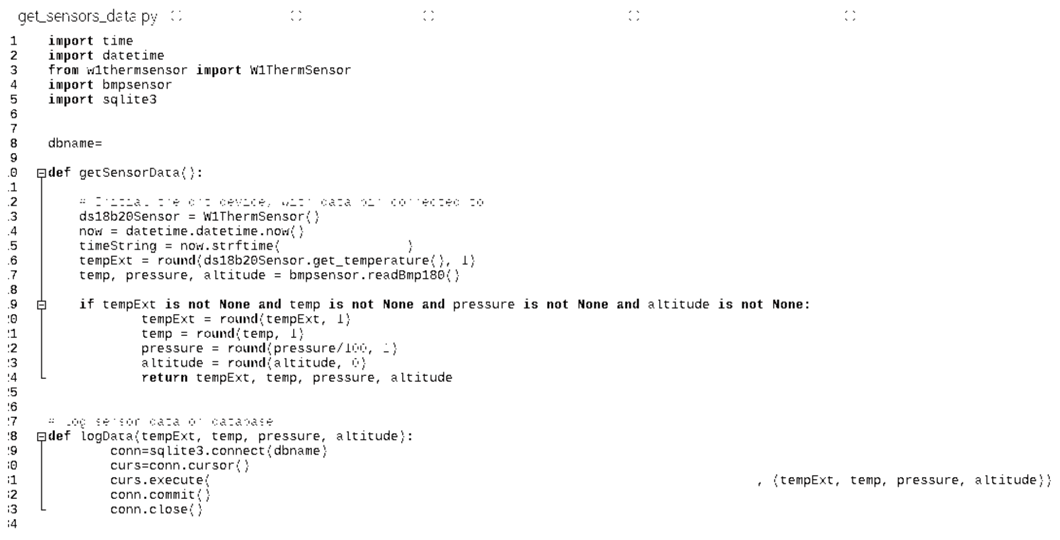
The script above create database with name “sensorData.db”. Then it creates the table with name “Weather_data” with fields: timestamp, external_temp, temp, pressure, altitude. “timestamp” datatime field stores data with current time, “external_temp” for storing temperature from DS18B20 sensor, “temp”, “pressure” and “altitude” are storing data from BMP180 sensor about air temperature, atmospheric pressure and altitude.



Picture 5.6: The general scheme of the weather service

5.3. Database storage

For retrieving and storing data into created database was created the next script below.



get_sensors_data.py

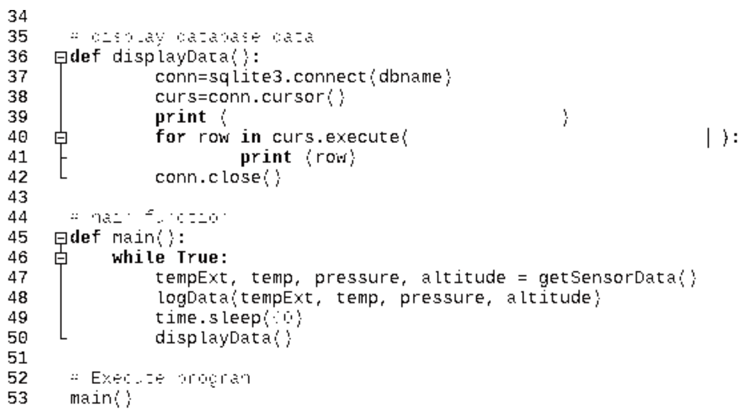
```
import time
import datetime
from w1thermsensor import W1ThermSensor
import bmpsensor
import sqlite3


dbname=

def getSensorData():

    # [illegible]
    ds18b20Sensor = W1ThermSensor()
    now = datetime.datetime.now()
    timeString = now.strftime(                )
    tempExt = round(ds18b20Sensor.get_temperature(), 1)
    temp, pressure, altitude = bmpsensor.readBmp180()

    if tempExt is not None and temp is not None and pressure is not None and altitude is not None:
            tempExt = round(tempExt, 1)
            temp = round(temp, 1)
            pressure = round(pressure/100, 1)
            altitude = round(altitude, 0)
            return tempExt, temp, pressure, altitude


# log sensor data on database
def logData(tempExt, temp, pressure, altitude):
        conn=sqlite3.connect(dbname)
        curs=conn.cursor()
        curs.execute(                                                    , (tempExt, temp, pressure, altitude))
        conn.commit()
        conn.close()
```

Picture 5.7: The part of code to insert data into database table

From this picture we can wee the "getSensorData" function to retrieve weather data from connected DS18B20 and BMP180 sensors. For both of them also were installed "w1thermsensor" and "bmpsensor" libraries appropriately.

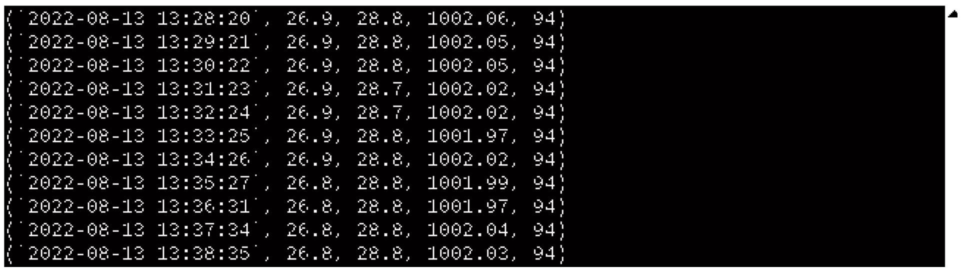
The function "logData" is using to insert data to database. The function "displayData" is for displaying the data which were added to the database.

The script above stores weather data into database and display it with 1 minute period.



```
# display database data
def displayData():
        conn=sqlite3.connect(dbname)
        curs=conn.cursor()
        print (                              )
        for row in curs.execute(                                    | ):
                print (row)
        conn.close()

# main function
def main():
    while True:
        tempExt, temp, pressure, altitude = getSensorData()
        logData(tempExt, temp, pressure, altitude)
        time.sleep(10)
        displayData()

# Execute program
main()
```

Picture 5.8: The part of code to retrieve data from database table



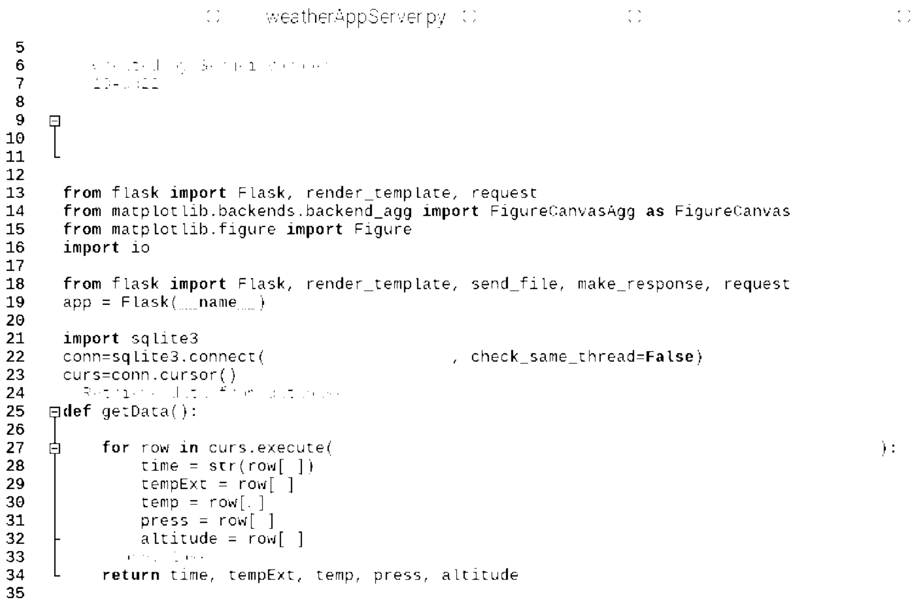
```
('2022-08-13 13:28:20', 26.9, 28.8, 1002.06, 94)
('2022-08-13 13:29:21', 26.9, 28.8, 1002.05, 94)
('2022-08-13 13:30:22', 26.9, 28.8, 1002.05, 94)
('2022-08-13 13:31:23', 26.9, 28.7, 1002.02, 94)
('2022-08-13 13:32:24', 26.9, 28.7, 1002.02, 94)
('2022-08-13 13:33:25', 26.9, 28.8, 1001.97, 94)
('2022-08-13 13:34:26', 26.9, 28.8, 1002.02, 94)
('2022-08-13 13:35:27', 26.8, 28.8, 1001.99, 94)
('2022-08-13 13:36:31', 26.8, 28.8, 1001.97, 94)
('2022-08-13 13:37:34', 26.8, 28.8, 1002.04, 94)
('2022-08-13 13:38:35', 26.8, 28.8, 1002.03, 94)
```

Picture 5.9: Terminal window with retrieving from database table data

5.4. Creating web server application

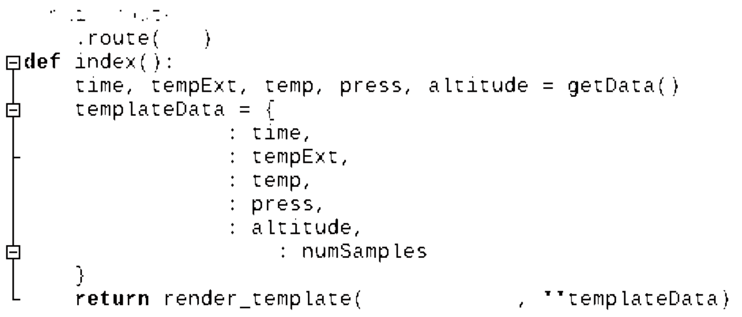
A web server is computer software and underlying hardware that accepts requests via HTTP (the network protocol created to distribute web content) [12]. Web server application was created with using Flask web framework.

The first step for creating webserver was installing Flask framework on Raspberry Pi. It was accomplished with command: “sudo apt-get install python3-flask”.



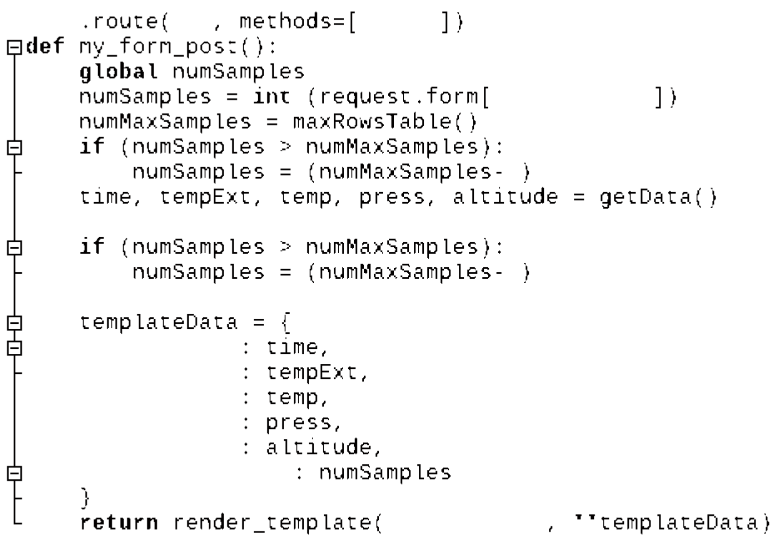
```
weatherAppServer.py

    [illegible]
    [illegible]




from flask import Flask, render_template, request
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
import io

from flask import Flask, render_template, send_file, make_response, request
app = Flask(__name__)

import sqlite3
conn=sqlite3.connect(                    , check_same_thread=False)
curs=conn.cursor()
  [illegible]
def getData():

    for row in curs.execute(                                              ):
        time = str(row[ ])
        tempExt = row[ ]
        temp = row[ ]
        press = row[ ]
        altitude = row[ ]
      [illegible]
    return time, tempExt, temp, press, altitude
```

Picture 5.10: The part of code to retrieve data from database in web server application



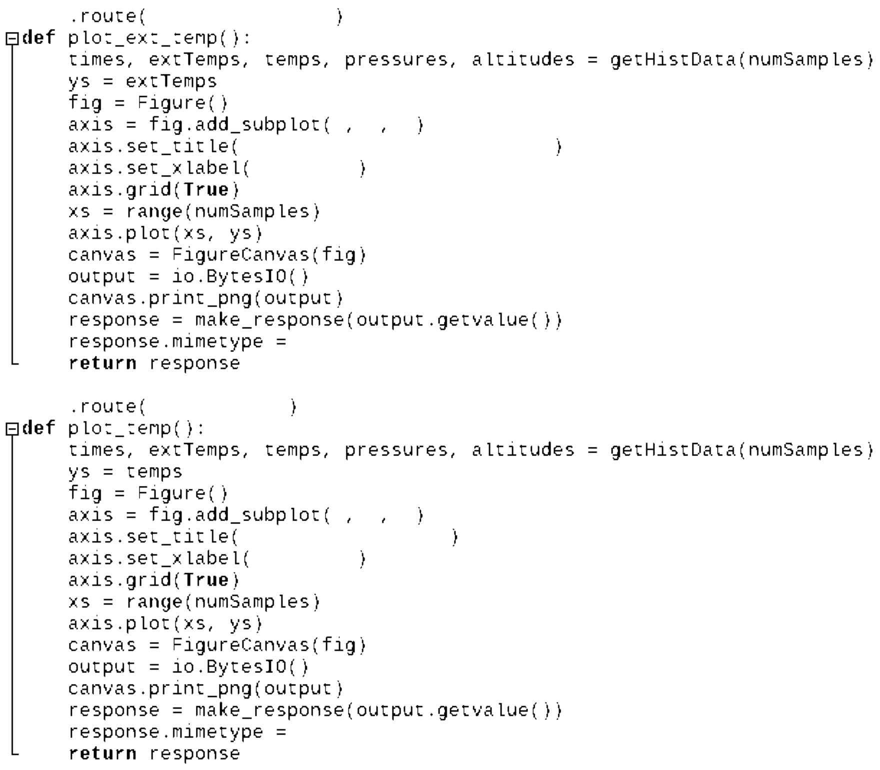
```
   [illegible]
    .route(    )
def index():
    time, tempExt, temp, press, altitude = getData()
    templateData = {
                : time,
                : tempExt,
                : temp,
                : press,
                : altitude,
                    : numSamples
    }
    return render_template(            , **templateData)
```

Picture 5.11: The part of code to retrieve the main web page



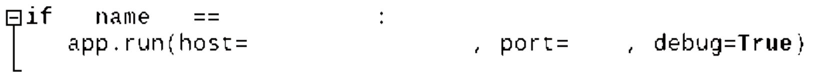
```
    .route(    , methods=[      ])
def my_form_post():
    global numSamples
    numSamples = int (request.form[             ])
    numMaxSamples = maxRowsTable()
    if (numSamples > numMaxSamples):
        numSamples = (numMaxSamples- )
    time, tempExt, temp, press, altitude = getData()

    if (numSamples > numMaxSamples):
        numSamples = (numMaxSamples- )

    templateData = {
                : time,
                : tempExt,
                : temp,
                : press,
                : altitude,
                    : numSamples
    }
    return render_template(            , **templateData)
```

Picture 5.12: The part of code to retrieve the main web page



```
    .route(                )
def plot_ext_temp():
    times, extTemps, temps, pressures, altitudes = getHistData(numSamples)
    ys = extTemps
    fig = Figure()
    axis = fig.add_subplot( ,  ,  )
    axis.set_title(                          )
    axis.set_xlabel(         )
    axis.grid(True)
    xs = range(numSamples)
    axis.plot(xs, ys)
    canvas = FigureCanvas(fig)
    output = io.BytesIO()
    canvas.print_png(output)
    response = make_response(output.getvalue())
    response.mimetype =
    return response

    .route(           )
def plot_temp():
    times, extTemps, temps, pressures, altitudes = getHistData(numSamples)
    ys = temps
    fig = Figure()
    axis = fig.add_subplot( ,  ,  )
    axis.set_title(                   )
    axis.set_xlabel(         )
    axis.grid(True)
    xs = range(numSamples)
    axis.plot(xs, ys)
    canvas = FigureCanvas(fig)
    output = io.BytesIO()
    canvas.print_png(output)
    response = make_response(output.getvalue())
    response.mimetype =
    return response
```

Picture 5.13: The part of code to retrieve graphs in the main web page



```
if   name   ==              :
    app.run(host=                    , port=    , debug=True)
```

Picture 5.14: The part of code with the main function of webserver application

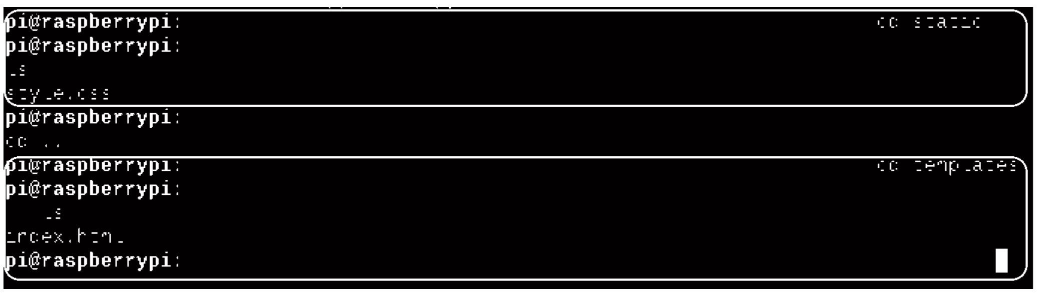
5.5. Creating web application

In order to create frontend of the web application static and template were created. These folders were created in the same directory with webserver application- “weatherAppServer.py”

```
pi@raspberrypi:                                                        ls
                 weatherAppServer.py
pi@raspberrypi:
```

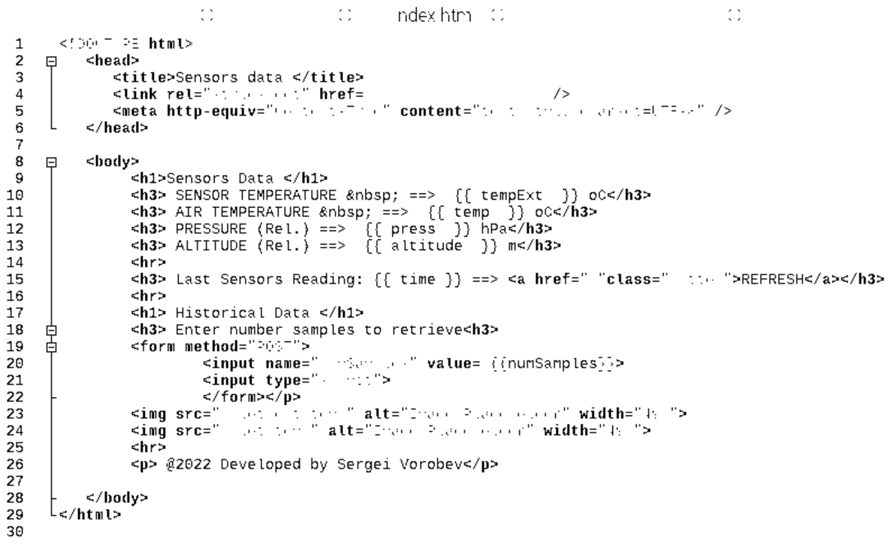
Picture 5.15: The terminal window with webserver directory

“static” folder was used to store style.css file and “templates” was used to store “index.html” files.



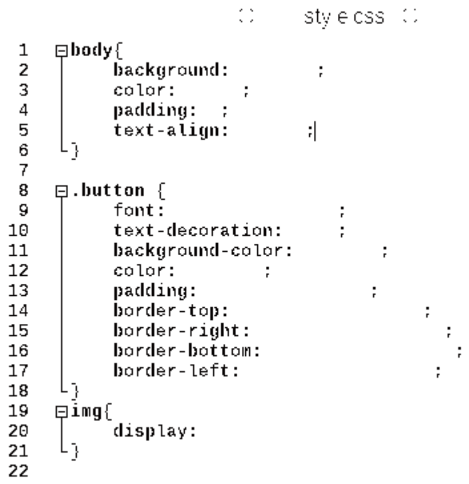
```
pi@raspberrypi:                                                   cd static
pi@raspberrypi:
ls
style.css
pi@raspberrypi:
cd ..
pi@raspberrypi:                                                   cd templates
pi@raspberrypi:
    ls
index.html
pi@raspberrypi:
```

Picture 5.16: The terminal window with static and templates content



```
<!DOCTYPE html>
   <head>
      <title>Sensors data </title>
      <link rel="[illegible]" href=                          />
      <meta http-equiv="[illegible]" content="[illegible]" />
   </head>

   <body>
         <h1>Sensors Data </h1>
         <h3> SENSOR TEMPERATURE &nbsp; ==>  {{ tempExt  }} oC</h3>
         <h3> AIR TEMPERATURE &nbsp; ==>  {{ temp  }} oC</h3>
         <h3> PRESSURE (Rel.) ==>  {{ press  }} hPa</h3>
         <h3> ALTITUDE (Rel.) ==>  {{ altitude  }} m</h3>
         <hr>
         <h3> Last Sensors Reading: {{ time }} ==> <a href=" "class="  [illegible] ">REFRESH</a></h3>
         <hr>
         <h1> Historical Data </h1>
         <h3> Enter number samples to retrieve<h3>
         <form method="POST">
                <input name="  [illegible]  " value= {{numSamples}}>
                <input type="[illegible]">
                </form></p>
         <img src="  [illegible]  " alt="[illegible]" width="[illegible]">
         <img src="  [illegible]  " alt="[illegible]" width="[illegible]">
         <hr>
         <p> @2022 Developed by Sergei Vorobev</p>

   </body>
</html>
```

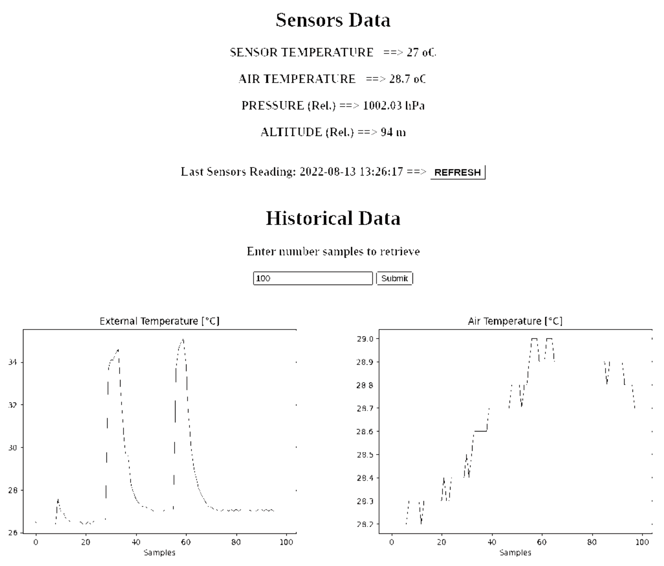
Picture 5.17: Code of index.html



```
body{
     background:        ;
     color:       ;
     padding:   ;
     text-align:        ;
}

.button {
     font:                ;
     text-decoration:     ;
     background-color:        ;
     color:        ;
     padding:                 ;
     border-top:                     ;
     border-right:                     ;
     border-bottom:                     ;
     border-left:                     ;
}
img{
     display:
}
```

Picture 5.18: Code of style.css

The main page of web weather service is presented in the picture below.


Sensors Data

SENSOR TEMPERATURE ==> 27 oC

AIR TEMPERATURE ==> 28.7 oC

PRESSURE (Rel.) ==> 1002.03 hPa

ALTITUDE (Rel.) ==> 94 m

Last Sensors Reading: 2022-08-13 13:26:17 ==> REFRESH

Historical Data

Enter number samples to retrieve

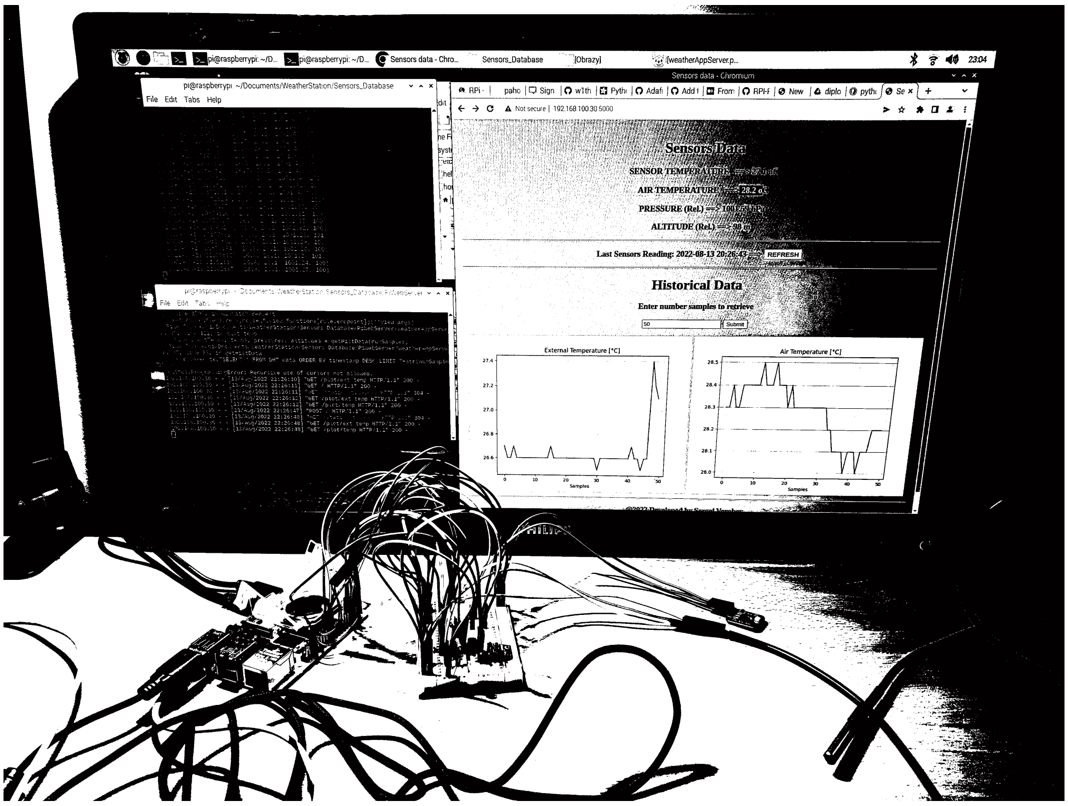
100 Submit

Picture 5.19: The main page interface

Where samples - number samples of weather data. According to the program the data updates every 60 seconds.

In the picture below are presented:

1. the web application.
2. terminal window with running script to retrieve data from the sensors;
3. terminal window with running script to run webserver;
4. RaspberryPi4 with connected sensors.



Picture 5.20: Implemented web application, webserver, Raspberry Pi4 with connected sensors

In the picture above also presented two more sensors - humidity and UV sensors. But they were not included to the final solution.

6. Testing of the solution

6.1. Functional testing

Functional testing is one of the types of testing aimed at checking whether the functional requirements of the software correspond to its actual characteristics. The main

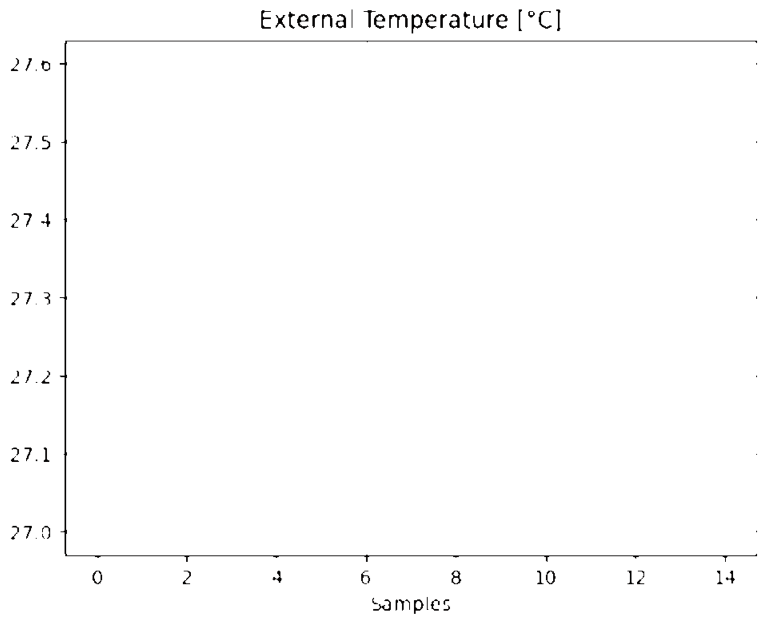
The main task of functional testing is to confirm that the product being developed has all the functionality required of it.Functional testing of sensors, database and web server were performed.

6.1.1 Functional testing of the sensors

6.1.1.1. Functional testing DS18B20 sensor data

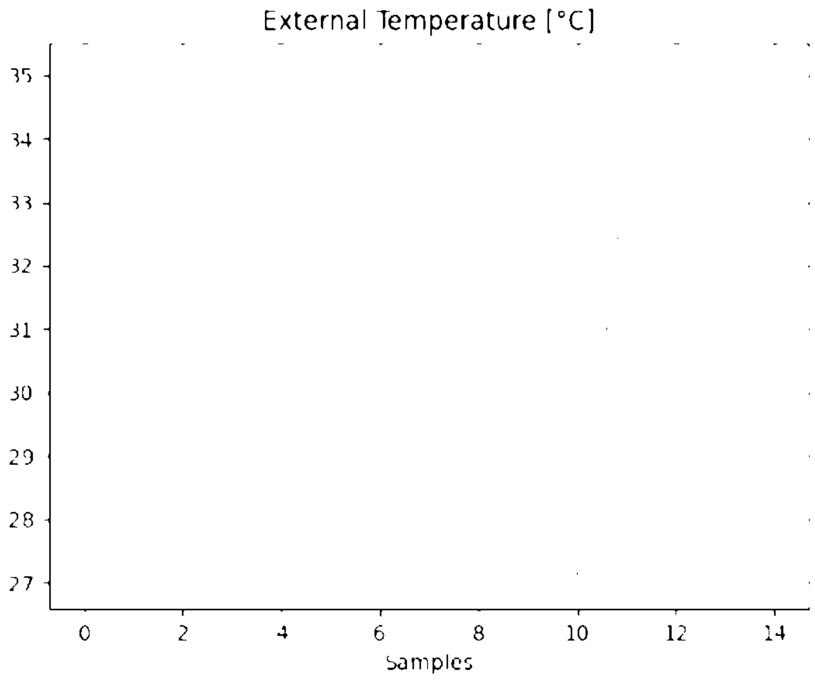
Table 6.1: Functional testing DS18B20 sensor data

Number	Activity	Conditions
	Monitoring temperature graph with data from the DS18B20 sensor	Raspberry PI is running, Sensor connected, server app is running, data from the sensor is showing in the web application
	Execution	**Results**
1	Clutching external temperature sensor in the hand. Observe the temperature change in the graph for 5 minutes.	The graph showed increasing temperature from 27.1°C up to 35.1°C
2	Unclenching the hand with the sensor and placing it in the original environment. Observe the temperature change in the graph for 5 minutes.	The graph showed decreasing temperature from 35.1°C up to 28.1°C



Picture 6.1: External temperature graph at the beginning of the test number 1.

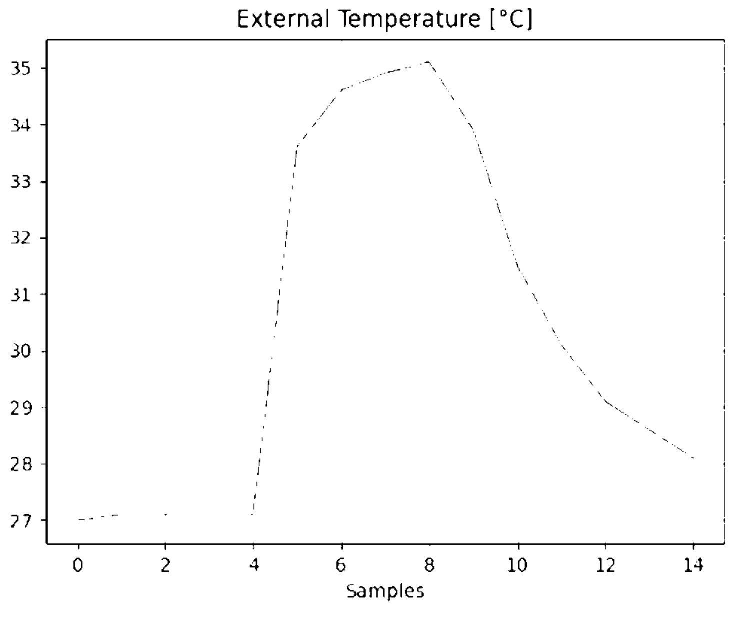
The graph below shows the temperature at the end of test number 1 and at the beginning of test number 2.



Picture 6.2: External temperature graph at the end of test number 1 and at the beginning of test number 2.

Both above graphs show the increase in temperature captured by the external temperature sensor.

The graph below presents the



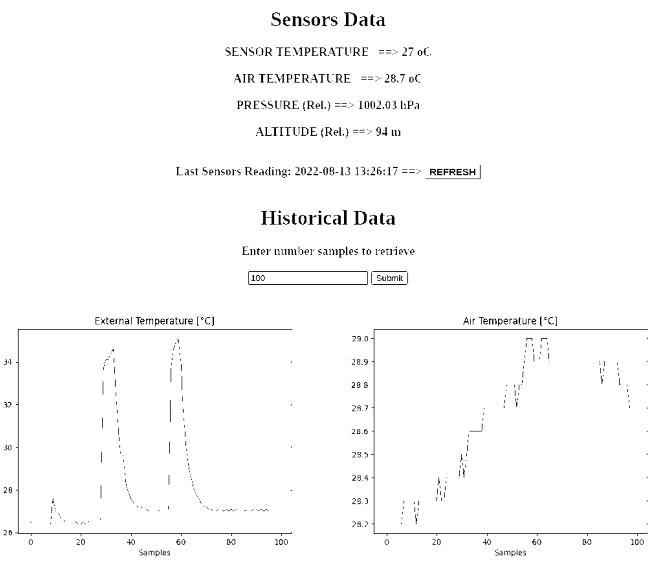
Picture 6.3: External temperature graph at the beginning of the test number 1.

6.1.1.2. Functional testing BMP180 sensor data

Table 6.2: Functional testing DS18B20 sensor data

Number	**Activity**	**Conditions**
	Monitoring temperature, pressure and altitude data from the BMP180 sensor	Raspberry PI is running, Sensor connected, server app is running, data from the sensor is showing in the web application
	Execution	**Results**
1	Click on the button "REFRESH" in the web application, make a notice with air temperature, pressure and altitude levels. Waiting for 5 minutes.	Web application presented all weather data with two temperature graphs
2	Click on the button "REFRESH" in the web application, make a notice with air temperature, pressure and altitude levels. Compare with weather data presented 5 minutes ago.	Air temperature and altitude are not changed. All other data changed. The temperature graphs also showed changes every 1 minute along 5 minutes of waiting

Despite the fact air temperature didn't change in test 1 and 2, the air temperature graph presented its dynamic changes along 5 minute test.


Sensors Data

SENSOR TEMPERATURE ==> 27 oC

AIR TEMPERATURE ==> 28.7 oC

PRESSURE (Rel.) ==> 1002.03 hPa

ALTITUDE (Rel.) ==> 94 m

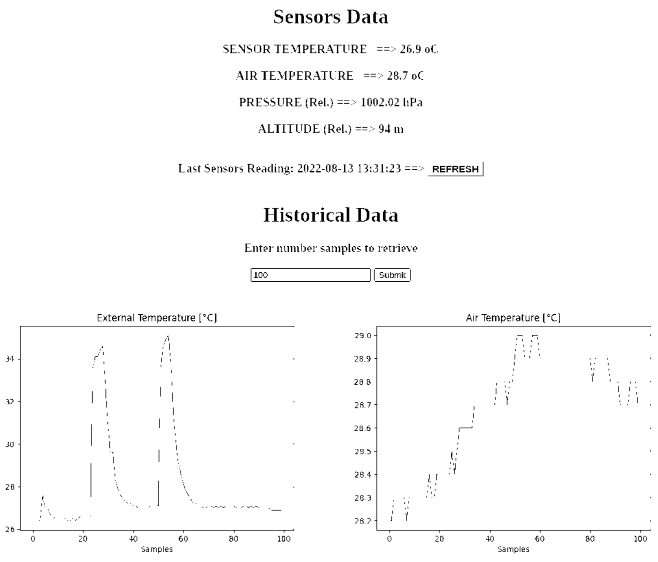
Last Sensors Reading: 2022-08-13 13:26:17 ==> REFRESH

Historical Data

Enter number samples to retrieve

100 Submit

Picture 6.4: Weather data after at the test number 1


Sensors Data

SENSOR TEMPERATURE ==> 26.9 oC

AIR TEMPERATURE ==> 28.7 oC

PRESSURE (Rel.) ==> 1002.02 hPa

ALTITUDE (Rel.) ==> 94 m

Last Sensors Reading: 2022-08-13 13:31:23 ==> REFRESH

Historical Data

Enter number samples to retrieve

100 Submit

Picture 6.5: Weather data after at the test number 2

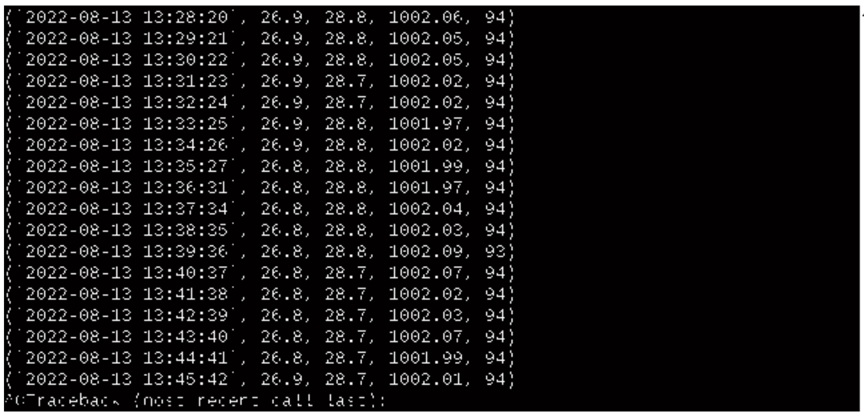
Although the air temperature did not change in tests 1 and 2, the graph of air temperature showed its dynamic change during the 5 minutes of the test.

6.1.2. Testing database

Functional testing validates that applications can access and update data in the database. Some testers prefer to validate database functionality by testing the application or applications that rely on the database. This approach poses a risk where the application tests fail to exercise all the fields and conditions within the database

Table 6.3: Functional testing of the database

Number	**Activity**	**Conditions**
	Monitoring terminal window in the directory with “get_sensors_data.py” python script	Raspberry PI is running, sensors connected
	Execution	**Results**
1	Run “get_sensors_data.py” python script with command “python3 get_sensors_data.py”. Waiting for 5 minutes.	Terminal window added into the database and presented 5 new rows with data.
2	Stop running “get_sensors_data.py” python script.	Terminal window stopped the process and showed the current directory.



```
('2022-08-13 13:28:20', 26.9, 28.8, 1002.06, 94)
('2022-08-13 13:29:21', 26.9, 28.8, 1002.05, 94)
('2022-08-13 13:30:22', 26.9, 28.8, 1002.05, 94)
('2022-08-13 13:31:23', 26.9, 28.7, 1002.02, 94)
('2022-08-13 13:32:24', 26.9, 28.7, 1002.02, 94)
('2022-08-13 13:33:25', 26.9, 28.8, 1001.97, 94)
('2022-08-13 13:34:26', 26.9, 28.8, 1002.02, 94)
('2022-08-13 13:35:27', 26.8, 28.8, 1001.99, 94)
('2022-08-13 13:36:31', 26.8, 28.8, 1001.97, 94)
('2022-08-13 13:37:34', 26.8, 28.8, 1002.04, 94)
('2022-08-13 13:38:35', 26.8, 28.8, 1002.03, 94)
('2022-08-13 13:39:36', 26.8, 28.8, 1002.09, 93)
('2022-08-13 13:40:37', 26.8, 28.7, 1002.07, 94)
('2022-08-13 13:41:38', 26.8, 28.7, 1002.02, 94)
('2022-08-13 13:42:39', 26.8, 28.7, 1002.03, 94)
('2022-08-13 13:43:40', 26.8, 28.7, 1002.07, 94)
('2022-08-13 13:44:41', 26.8, 28.7, 1001.99, 94)
('2022-08-13 13:45:42', 26.9, 28.7, 1002.01, 94)
^CTraceback (most recent call last):
```

Picture 6.6: Weather data after the test database number 1.

The implemented python script "get_sensors_data.py" inserts new data into the database and presents it in the terminal window every 60 seconds.

The output proves that the database was created correctly, the insertion of data into the database was successful and the retrieval of data from the database was also successful.

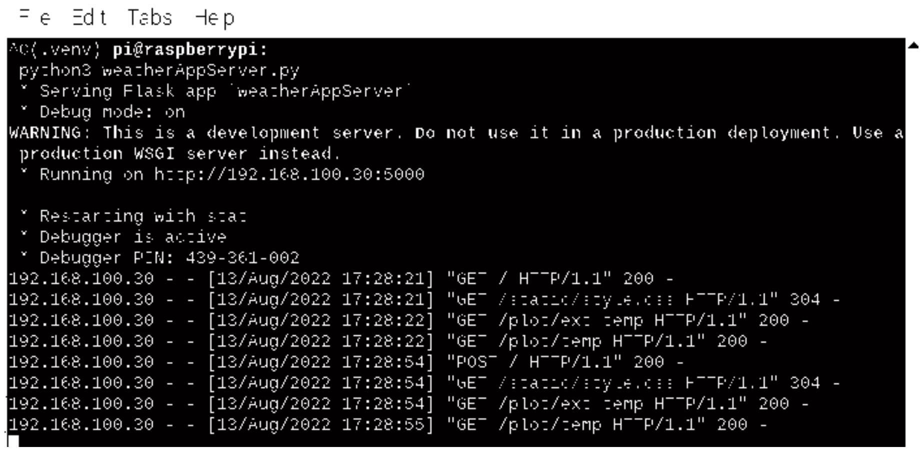
6.1.3 Testing web server and web interface of the solution

The web server application "weatherAppServer.py" was manually tested along with the weather sensor tests.

For testing web server application the first scenario was implemented:

1. Retrieving the main page;
2. Enter number of samples in the appropriate window in the main page;
3. Click on “Submit” button.

GET and POST CRUD[13] methods were involved during the implementation of this scenario. Both methods were successfully passed. This is represented in the figure below.



```
^C(.venv) pi@raspberrypi:
 python3 weatherAppServer.py
 * Serving Flask app 'weatherAppServer'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
 production WSGI server instead.
 * Running on http://192.168.100.30:5000

 * Restarting with stat
 * Debugger is active
 * Debugger PIN: 439-361-002
192.168.100.30 - - [13/Aug/2022 17:28:21] "GET / HTTP/1.1" 200 -
192.168.100.30 - - [13/Aug/2022 17:28:21] "GET /static/style.css HTTP/1.1" 304 -
192.168.100.30 - - [13/Aug/2022 17:28:22] "GET /plot/ext_temp HTTP/1.1" 200 -
192.168.100.30 - - [13/Aug/2022 17:28:22] "GET /plot/temp HTTP/1.1" 200 -
192.168.100.30 - - [13/Aug/2022 17:28:54] "POST / HTTP/1.1" 200 -
192.168.100.30 - - [13/Aug/2022 17:28:54] "GET /static/style.css HTTP/1.1" 304 -
192.168.100.30 - - [13/Aug/2022 17:28:54] "GET /plot/ext_temp HTTP/1.1" 200 -
192.168.100.30 - - [13/Aug/2022 17:28:55] "GET /plot/temp HTTP/1.1" 200 -
```

Picture 6.7: Running the web server application

For testing web server application the second scenario also was implemented:

1. Retrieving the main page;

3. Click on “REFRESH” button.

In particular, the GET method - to retrieve the main page with the index.html file. Method was successfully passed when loading the main page, as well as when updating it, as well as after clicking "REFRESH".

All the implemented CRUD methods along the first and the second scenarios were successfully passed.

6.1.4 Conclusion

All of the functional tests mentioned above have been successfully passed. The wiring diagram of the sensors and the microcomputer has been done correctly. All functions of the web application work properly. The weather data from the sensors are received in real time.

All elements of the web interface with data are displayed correctly. The database is updated every 60 seconds, and the webserver reads the updated data and transfers them to the main interface page without loss or distortion.

6.2. Non-functional testing

Section 3.3 of this research paper presents the non-functional requirements to the solution being developed. This subsection describes how these requirements are met.

6.2.1. Testing for limitation requirements

The solution used a Raspberry Pi4 microcomputer with the Chromium web browser, which met the technical requirements.

6.2.2. Testing for transparency and reusability

Simple patterns were used in this solution. They were used in the creation of the database and web server as well as in the connection of the sensors. In addition, inexpensive and popular sensors were used. The Raspberry PI4 microcomputers do not boast the same affordability. However, the widespread use of these devices, a big fan community, as well as the ease of connecting sensors to it made the choice in its favor.

6.2.3. Testing for localization

The code, all comments in the code and the web interface of the solution are written in English.

6.2.4. Testing for consistency

The main color of the font, as well as the background color of the interface are selected from close color gammas, but in such a way that they do not merge and create a pleasant overall perception. And the absence of a large number of elements makes the interface clear. The web interface follows the principles of the Material UI.

6.2.5. Testing for security

The connections between the sensors and the microcomputer were stable throughout the entire solution creation process. There was no data loss between them in this case.

7. Conclusion

The purpose of the work was to create a web service providing data from a weather station. However, in the course of the work, the weather station itself was additionally built. This research paper also described this process. This made it possible to operate with real weather data.

There is a huge choice of ready solutions of weather stations on the market. And the idea of the work was to develop the solution of this problem, which would be distinguished by its simplicity and would use cheap components.

The Raspberry Pi microcomputer was chosen as the main element of the weather station. This component was not easy to acquire. Perhaps because of the high demand for it. However, the use of this component did not raise doubts among the author of this work. Since this component could provide a successful solution to the task, to create a weather station. This microcomputer has a large fan base and broad community support.

Among other components, affordable and cheap components were selected.

All code was written in Python programming language, which is distinguished by its low entry level.

The presented solution is distinguished by its simplicity. In addition, during the creation of the weather station, all functional and non-functional requirements specified in Section 3 of the Chapter 3 were met. And in Chapter 6 all these requirements, in the opinion of the author of this project and this scientific work, have been successfully tested.

7.1. Advantages and disadvantages of the designed solution

The disadvantages of this solution include:

- Weak overall design of the weather station. Since the development of it, and in particular any external body, was not planned. This leads to the following disadvantage;

- Not reliable enough connections of the conductors. In addition to the lack of housing, this is also due to the lack of soldering of most of the contacts used. And this can lead to the following disadvantage;

- the possibility of an unexpected loss of signal transmission between the wires of the workstation, and as a consequence to the loss of data from the sensors.

Among the advantages are the following:

- The low final cost of the finished solution, due to the choice of the cheapest components;

- High scalability of the solution. The choice of Raspberry Pi microcomputer makes it possible to connect up to 30 devices simultaneously. This makes it possible to connect and process signals from all existing weather sensors;

- Using one programming language – Python;

- Clear and intuitive web interface.

7.2. What was learned from the work

The implementation of this work allowed to the author of this work to put into practice the knowledge gained in the Internet of Things course and the Advanced Web Technologies course. Particularly, the author of this thesis gained priceless experience in creating a complete project using a Raspberry Pi microcomputer. It was impressive. As well as creating a database and a webserver to transfer the data to the web application.

As a result, weather station with web service providing data from a weather station were created.

Bibliography

[1] *Datasheet : Raspberry Pi 4 Model B* , [online], Raspberrypi.org, June 2019, retrieved 16.04.2022 12:00am, internet access: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf

[2] Gillis Alexander, *What is internet of things (IoT)?*[online], IOT Agenda, 2021, retrieved 17.04.2022 11:00am, internet access: https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT

[3] *The Python Tutorial,* [online], Python Software Foundation, Aug 2001-2022, retrieved 19.04.2022 10:00am, internet access: https://docs.python.org/3/tutorial/index.html

[4] *About SQLite*, [online], Sqlite.org, May 2010, retrieved 19.04.2010 12:00am, internet access: https://www.sqlite.org/about.html

[5] Armin Ronacher, *Opening the Flask*, 2011, pages 18-33

[6] *DHT22 temperature-humidity sensor + extras*, [online], Adafruit Industries, LLC., retrieved 03.05.2010 10:00am, internet access: https://www.adafruit.com/product/385#technical-details

[7] *How many devices can be connected to RPI4 ?* [online]. Raspberri Pi official forum, 2019, retrieved 25.04.2022 13:00am, internet access: https://forums.raspberrypi.com/viewtopic.php?t=255156&sid=2f4ddd322443379282e6cfc5d2702086

[8] S. Robertson, J. Robertson, *Mastering the Requirements Process: Getting Requirements Right*, Addison-Wesley, 2012, chapter 17

[9] J. W. Kincaid, *Customer Relationship Management: Getting it Right!*, Prentice Hall Professional, 2012, section 21.2.2

[10] *What are swimlane diagrams?*, [online], Lucid Software Inc, LLC., retrieved 03.05.2010 11:00am, internet access: https://www.lucidchart.com/pages/tutorial/swimlane-diagram

[11] *BMP180 Barometric Pressure/Temperature/Altitude Sensor- 5V ready*, [online], Adafruit Industries, LLC., retrieved 14.05.2022 11:00am, internet access: https://www.adafruit.com/product/1603

[12] *What is a REST API?,* Red Hat, 2020, [online], retrieved 03.07.2022 11:00am, internet access: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[13] R. S. Richardson L., *RESTful Web APIs: Services for a Changing World*, O'Reilly Media, Inc., 2013.

List of Pictures

List of Tables