**Serverless IoT Data Processing**

**Phase 5: Project Documentation and Submission**

**Introduction:**

Completely automated smart homes are on the way to becoming a well-established reality. The ever increasing number of smart objects in the smart home environment requires the definition of standardized and flexible protocols for state information exchange, besides optimal strategies for processing large number of commands. New technologies such as serverless computing and ad hoc communication protocols can be leveraged to manage a large fleet of smart objects, while also ensuring good accessibility and intuitive user interfaces. Serverless computing is a cloud computing model in which the cloud provider manages the infrastructure and automatically allocates resources as needed. This means that businesses can focus on building their applications and processing their data without worrying about managing and scaling their infrastructure.

**Project Objective:**

The objective of the project is to transform a home into smart living space using IBM Cloud Functions for data processing. The goal is to collect data from various smart devices, process in real-time, and automate routines for energy efficiency and home security. This involves designing the smart home setup, implementing data collection and processing, and leveraging IBM Cloud for storage and analysis.

**Design thinking:**

**Data Integration:**

Identify and integrate smart devices such as thermostats, motion sensors, and cameras into the smart home ecosystem.

**Data Collection:**

Set up data collection from these devices, utilizing loT protocols. Real-time Processing: Implement real-time data processing using IBM Cloud Functions.

**Real-time processing:**

The real-time data collected by the sensors are transmitted to the cloud database platform like aws, IBM Cloud functions etc., It is to implement real-time processing.

**Automation:**

Develop automated routines for energy efficiency (e.g., adjustingthermostat settings) and home security (e.g., sending alerts on motion detection)

**Storage and Analysis:**

Store data in IBM Cloud Object Storage and analyze it to gain insights into energy consumption, security events, and patterns.

**Innovation:**

**Step 1 - Selection of IoT Devices**

IoT devices like Smart lighting, Smart Thermostats and smart sensors likemotion sensors /smoke detectors are chosen which suits our requirements. It is ensured that is supports data communication protocols like MQTT [Message Queuing Telemetry Transport] or HTTP [Hypertext Transfer Protocol].

**Step 2 – Setting up Cloud Service**

IBM Cloud provides the IBM Watson IoT platform, which can be used to connect and manage IoT devices securely. It supports MQTT protocol and HTTP protocol, making it suitable for IoT data communication.

Function as a service [Faas]

 IBM cloud Functions is IBM’s serverless computing platform.

 It allows you to create and deploy serverless functions, including IoT data arriving from devices.

**Step 3 – Data Storage**

Storage options like cloud object storage and IBM Db2 for storing and managing IoT data. These can be used to store historical data and making it accessible for analytics.

**Step 4 – IoT Device Integration**

The IoT devices are configured to send data to the IBM Db2 cloud Platform. The data are stored in the cloud platform.

**Step 5 – Creating Serverless Functions**

The code for these functions is written to analyse, transform or respond to the data as needed. Separate functions like detecting motion, temperature analysing etc are created.

**Step 6 – Configuring Trigger setup**

Trigger is created for specific serverless functions based on the occurrence of the event or the data arrival. A function is triggered based on the event occurrence like motion sensor, temperature sensor etc.With these steps we can achieve a Serverless IoT Data Processing which transforms your Home into a Smart Living Space using IBM cloud Functions. Thus experiencing the convenience and peace of mind of a serverless smart Home!

**Development Part 1:**

In this phase, solution is built for serverless home automation using cloud functions and device integration. Smart devices are chosen and integrated and setup for data collection is made.

**Components used:**

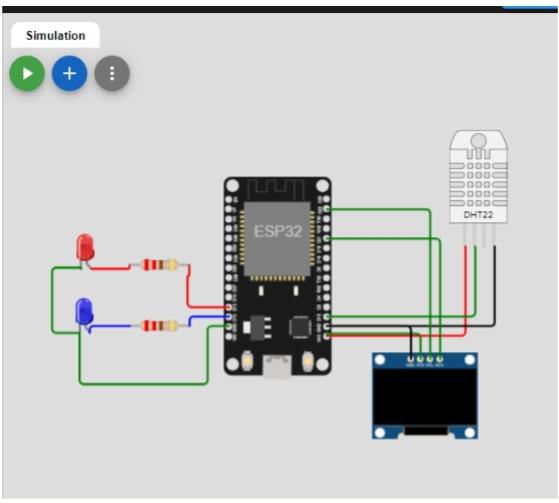
1. ESP32

2. DHT22

3. LED

4. Resistor

5. SSD1306 OLED display



**PROGRAM:**

**Main.py**

from machine import Pin, PWM, I2C

from umqtt.simple import MQTTClient

import ujson

import network

import utime as time

import dht

from led\_pwm import LED

import ssd1306

from device\_traits import \*

# Device Setup

DEVICE\_ID = "wokwi001"

# WiFi Setup

WIFI\_SSID = "Wokwi-GUEST"

WIFI\_PASSWORD = ""

# MQTT Setup

MQTT\_BROKER = "7706f65c1d3b40428155631e41b1cd5f.s1.eu.hivemq.cloud"

MQTT\_CLIENT = DEVICE\_ID

MQTT\_TELEMETRY\_TOPIC = 'iot/device/{0}/telemetry'.format(DEVICE\_ID)

MQTT\_CONTROL\_TOPIC = 'iot/device/{0}/control'.format(DEVICE\_ID)

MQTT\_MASTER\_TELEMETRY\_TOPIC = 'iot/telemetry'.format(DEVICE\_ID)

MQTT\_MASTER\_CONTROL\_TOPIC = 'iot/control'.format(DEVICE\_ID)

# DHT Sensor Setup

DHT\_PIN = Pin(15)

dht\_sensor = dht.DHT22(DHT\_PIN)

# LED/LAMP Setup

RED\_LED = LED(12)

BLUE\_LED = LED(13)

FLASH\_LED = Pin(2, Pin.OUT)

i2c = I2C(0, scl=Pin(22), sda=Pin(21))

oled\_width = 128

oled\_height = 64

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

# Turn On LED

RED\_LED.on()

BLUE\_LED.on()

# Methods

def did\_recieve\_callback(topic, message):

print('\n\nData Recieved! \ntopic = {0}, message = {1}'.format(topic, message))

if topic == MQTT\_CONTROL\_TOPIC.encode():

#Get the command message from json command.

command\_message = ujson.loads(message.decode())["command"]

if command\_message == "lamp/red/on":

RED\_LED.on()

send\_led\_status()

elif command\_message == "lamp/red/off":

RED\_LED.off()

send\_led\_status()

elif command\_message == "lamp/blue/on":

BLUE\_LED.on()

send\_led\_status()

elif command\_message == "lamp/blue/off":

BLUE\_LED.off()

send\_led\_status()

elif command\_message == "lamp/on":

RED\_LED.on()

BLUE\_LED.on()

send\_led\_status()

elif command\_message == "lamp/off":

RED\_LED.off()

BLUE\_LED.off()

send\_led\_status()

elif command\_message == "status":

mqtt\_client\_publish(MQTT\_TELEMETRY\_TOPIC, get\_sensor\_json\_data())

send\_led\_status()

elif len(command\_message.split('/')) == 4 and command\_message.split('/')[2] ==

"brightness":

# "lamp/red/brightness/34"

brightness\_commands = command\_message.split('/')

brightness\_value = float(brightness\_commands[3])

if(brightness\_commands[1] == "red"):

RED\_LED.set\_brightness(brightness\_value)

if(brightness\_commands[1] == "blue"):

BLUE\_LED.set\_brightness(brightness\_value)

send\_led\_status()

else:

return

# MQTT\_MASTER\_CONTROL\_TOPIC is used for Google Home Integration.

if topic == MQTT\_MASTER\_CONTROL\_TOPIC.encode():

#Get the command message from json command.

received\_message = ujson.loads(message.decode())

should\_acknowledge = {}

command\_data = {}

if 'type' in received\_message:

if received\_message['type'] == "command":

if "data" in received\_message and DEVICE\_ID in received\_message["data"]:

print(received\_message)

command\_data = received\_message["data"]

should\_acknowledge = received\_message["acknowledge"]

process\_commands(command\_data[DEVICE\_ID], should\_acknowledge)

else:

print("Message is not for this device")

elif received\_message['type'] == "ping" and received\_message['id'] == DEVICE\_ID:

print("PING Message Received")

else:

print("Message not of type COMMAND or PING")

else:

print("Message type invalid, do not process.")

def mqtt\_connect():

print("Connecting to MQTT broker ...", end="")

mqtt\_client = MQTTClient(MQTT\_CLIENT, MQTT\_BROKER, user="iotsmarthome",

password="Smarthome1", ssl=True, ssl\_params={'server\_hostname':MQTT\_BROKER})

mqtt\_client.set\_callback(did\_recieve\_callback)

mqtt\_client.connect()

print("Connected.")

mqtt\_client.subscribe(MQTT\_CONTROL\_TOPIC)

# subscribe to master topics for Google Home Control

mqtt\_client.subscribe(MQTT\_MASTER\_CONTROL\_TOPIC)

return mqtt\_client

def create\_control\_json\_data(command, command\_id):

#import ujson

data = ujson.dumps({

"device\_id": DEVICE\_ID,

"command\_id": command\_id,

"command": command

})

return data

def get\_sensor\_json\_data():

data = ujson.dumps({

"device\_id": DEVICE\_ID,

"temp": dht\_sensor.temperature(),

"humidity": dht\_sensor.humidity(),

"type": "sensor"

})

return data

def get\_all\_parts\_settings():

data = {

"device\_id": DEVICE\_ID,

"temp": dht\_sensor.temperature(),

"humidity": dht\_sensor.humidity(),

"humidity\_ison": humiditySetpointPercentOn,

"set\_temp": thermostatTemperatureSetpoint,

"set\_humidity": humiditySetpointPercent,

"red\_led": True if RED\_LED.value() == 1 else False,

"blue\_led": True if BLUE\_LED.value() == 1 else False,

"red\_led\_brightness": RED\_LED.get\_brightness(),

"blue\_led\_brightness": BLUE\_LED.get\_brightness()

}

global parts\_settings

parts\_settings = data

return data

def create\_master\_json\_data():

data = ujson.dumps(get\_all\_parts\_settings())

return data

def mqtt\_client\_publish(topic, data):

try:

print("\nUpdating MQTT Broker...")

mqtt\_client.publish(topic, data)

print(data)

except:

print("MQTT client may not be initialized.")

def send\_led\_status():

data = ujson.dumps({

"device\_id": DEVICE\_ID,

"red\_led": "ON" if RED\_LED.value() == 1 else "OFF",

"blue\_led": "ON" if BLUE\_LED.value() == 1 else "OFF",

"red\_led\_brightness": RED\_LED.get\_brightness(),

"blue\_led\_brightness": BLUE\_LED.get\_brightness(),

"type": "lamp"

})

mqtt\_client\_publish(MQTT\_TELEMETRY\_TOPIC, data)

def get\_part\_by\_name(name):

if name == "red\_led":

return RED\_LED

if name == "blue\_led":

return BLUE\_LED

def send\_ack\_data(data):

mqtt\_client\_publish(MQTT\_MASTER\_TELEMETRY\_TOPIC, data)

def process\_commands(commands, acknowledge):

if acknowledge:

data = ujson.dumps({

"gatewayId": DEVICE\_ID,

"data": commands,

})

send\_ack\_data( data)

for command in commands:

part = command["deviceId"].split("::")[1]

command\_actions = command["commands"]

if part == "red\_led" or part == "blue\_led":

if 'on' in command\_actions:

get\_part\_by\_name(part).set\_value(0 if command\_actions["on"] == False else 1)

if 'brightness' in command\_actions:

get\_part\_by\_name(part).set\_brightness(command\_actions["brightness"])

if part == "sensor\_temp" and "thermostatTemperatureSetpoint" in command\_actions:

global thermostatTemperatureSetpoint

thermostatTemperatureSetpoint = command\_actions["thermostatTemperatureSetpoint"]

print("\n\nTEMPERATURE SENSOR = " +

str(command\_actions["thermostatTemperatureSetpoint"]) )

if part == "sensor\_humidity":

if 'on' in command\_actions:

global humiditySetpointPercentOn

humiditySetpointPercentOn = command\_actions["on"]

if "thermostatTemperatureSetpoint" in command\_actions:

global humiditySetpointPercent

humiditySetpointPercent = command\_actions["thermostatTemperatureSetpoint"]

print("\n\nHUMIDITY SENSOR = " +

str(command\_actions["thermostatTemperatureSetpoint"]))

oled\_print()

def oled\_print():

oled.fill(0)

oled.show()

oled.text('CUR TMP: '+ str(dht\_sensor.temperature()), 0, 0)

oled.text('SET TMP: '+ str(thermostatTemperatureSetpoint), 0, 10)

oled.text('CUR HUM: '+ str(dht\_sensor.humidity()), 0, 20)

oled.text('SET HUM: '+ str(humiditySetpointPercent if humiditySetpointPercentOn == True

else "OFF"), 0, 30)

oled.text('RED: '+ str(RED\_LED.get\_brightness()) + '%', 0, 40)

oled.text('BLUE: ' + str(BLUE\_LED.get\_brightness()) + '%', 0, 50)

oled.show()

def mqtt\_ping():

data = ujson.dumps({

"device\_id": DEVICE\_ID,

"id": DEVICE\_ID,

"type": "ping",

"devices": []

})

mqtt\_client\_publish(MQTT\_MASTER\_CONTROL\_TOPIC, data)

# Application Logic

# Connect to WiFi

wifi\_client = network.WLAN(network.STA\_IF)

wifi\_client.active(True)

print("Connecting device to WiFi")

wifi\_client.connect(WIFI\_SSID, WIFI\_PASSWORD)

# Wait until WiFi is Connected

while not wifi\_client.isconnected():

print("Connecting")

time.sleep(0.1)

print("WiFi Connected!")

print(wifi\_client.ifconfig())

# Connect to MQTT

mqtt\_client = mqtt\_connect()

# RED\_LED.off()

# BLUE\_LED.off()

mqtt\_client\_publish(MQTT\_CONTROL\_TOPIC, create\_control\_json\_data('lamp/off',

'DEVICE-RESET-00'))

# read dht\_sensor and register device.

dht\_sensor.measure()

time.sleep(0.2)

# Set default settings

parts\_settings = get\_all\_parts\_settings()

register\_settings = ujson.dumps(

get\_settings(DEVICE\_ID, "register\_settings", parts\_settings)

)

mqtt\_client\_publish(MQTT\_MASTER\_CONTROL\_TOPIC, register\_settings)

master\_data\_old = ""

oled\_print()

pause\_time = 1500000

start\_time = time.ticks\_ms()

sleep\_time = pause\_time

while True:

mqtt\_client.check\_msg()

print(". ", end="")

try:

FLASH\_LED.on()

dht\_sensor.measure()

time.sleep(0.2)

FLASH\_LED.off()

except:

pass

master\_data\_new = create\_master\_json\_data()

if master\_data\_new != master\_data\_old:

mqtt\_client\_publish(MQTT\_TELEMETRY\_TOPIC, get\_sensor\_json\_data())

send\_led\_status()

master\_data\_old = master\_data\_new

oled\_print() #print out to OLED

# update device settings in cloud.

all\_settings = ujson.dumps(

get\_settings(DEVICE\_ID, "update\_settings", get\_all\_parts\_settings())

)

mqtt\_client\_publish(MQTT\_MASTER\_CONTROL\_TOPIC, all\_settings)

time.sleep(0.1)

sleep\_time = sleep\_time - (time.ticks\_ms() - start\_time)

if sleep\_time < 1:

start\_time = time.ticks\_ms()

sleep\_time = pause\_time

mqtt\_ping()

**Diagram.json**

{

"version": 1,

"author": "Francis Okechukwu",

"editor": "wokwi",

"parts": [

{

"type": "wokwi-esp32-devkit-v1",

"id": "esp",

"top": 52.79,

"left": -4.24,

"attrs": { "env": "micropython-20220618-v1.19.1" }

},

{

"type": "wokwi-led",

"id": "led1",

"top": 103.38,

"left": -159.88,

"attrs": { "color": "red" }

},

{

"type": "wokwi-resistor",

"id": "r1",

"top": 136.19,

"left": -97.24,

"attrs": { "value": "220" }

},

{

"type": "wokwi-led",

"id": "led2",

"top": 166.42,

"left": -161.39,

"attrs": { "color": "blue" }

},

{

"type": "wokwi-resistor",

"id": "r2",

"top": 196.13,

"left": -96.93,

"attrs": { "value": "220" }

},

{

"type": "wokwi-dht22",

"id": "dht1",

"top": 8.53,

"left": 218.4,

"attrs": { "humidity": "69", "temperature": "24" }

},

{ "type": "board-ssd1306", "id": "oled1", "top": 228.03, "left": 141.58, "attrs": {} }

],

"connections": [

[ "esp:TX0", "$serialMonitor:RX", "", [] ],

[ "esp:RX0", "$serialMonitor:TX", "", [] ],

[ "esp:D12", "r1:2", "red", [ "h-40.94", "v-30.82", "h-2.48" ] ],

[ "esp:D13", "r2:2", "blue", [ "h0" ] ],

[ "r1:1", "led1:A", "red", [ "v0" ] ],

[ "led2:A", "r2:1", "blue", [ "v0" ] ],

[ "led2:C", "esp:GND.2", "green", [ "v50.57", "h125.97", "v-58.87" ] ],

[ "led1:C", "led2:C", "green", [ "v-1.07", "h-28.6", "v62.59" ] ],

[ "dht1:VCC", "esp:3V3", "red", [ "v0" ] ],

[ "dht1:SDA", "esp:D15", "green", [ "v0" ] ],

[ "dht1:GND", "esp:GND.1", "black", [ "v0" ] ],

[ "esp:3V3", "oled1:VCC", "green", [ "v-2.15", "h90.57" ] ],

[ "oled1:GND", "esp:GND.1", "black", [ "v0" ] ],

[ "oled1:SCL", "esp:D22", "green", [ "v0" ] ],

[ "oled1:SDA", "esp:D21", "green", [ "v0" ] ]

],

"dependencies": {}

}

**Led\_pwm.py:**

from machine import Pin, PWM

class LED:

def \_\_init\_\_(self, pin\_num, freq=500):

self.pin\_num = pin\_num

self.freq = freq

self.pwm = PWM(Pin(pin\_num), freq=freq)

self.is\_on = False # Track the LED state

self.min\_percent = 6

self.percent = 0 # Track the brightness state

self.duty = int(self.min\_percent / 100 \* 1023) # Track the duty state

def on(self):

self.pwm.duty(self.duty) # Set duty cycle to 1023 (maximum value) to fully turn on the

LED

self.is\_on = True # Update LED state

def off(self):

self.pwm.duty(0) # Set duty cycle to 0 to fully turn off the LED

self.is\_on = False # Update LED state

def set\_brightness(self, percentage):

# Convert percentage to a duty cycle value between 0 and 1023 (100%)

pvalue = percentage if percentage > self.min\_percent else self.min\_percent

self.percent = percentage

self.duty = int(pvalue / 100 \* 1023)

if self.is\_on == True:

self.pwm.duty(self.duty) # Set duty cycle to the calculated value

def get\_brightness(self):

return self.percent

def set\_value(self, state):

if state == 0:

self.off()

elif state == 1:

self.on()

else:

raise ValueError("Invalid state value. Use 0 for off and 1 for on.")

def value(self):

return 1 if self.is\_on == True else 0 # Return current LED state (1 if on, 0 if off)

def deinit(self):

self.pwm.deinit() # Deinitialize the PWM to clean up

**Ssd1306.py**

#MicroPython SSD1306 OLED driver, I2C and SPI interfaces created by Adafruit

import time

import framebuf

# register definitions

SET\_CONTRAST = const(0x81)

SET\_ENTIRE\_ON = const(0xa4)

SET\_NORM\_INV = const(0xa6)

SET\_DISP = const(0xae)

SET\_MEM\_ADDR = const(0x20)

SET\_COL\_ADDR = const(0x21)

SET\_PAGE\_ADDR = const(0x22)

SET\_DISP\_START\_LINE = const(0x40)

SET\_SEG\_REMAP = const(0xa0)

SET\_MUX\_RATIO = const(0xa8)

SET\_COM\_OUT\_DIR = const(0xc0)

SET\_DISP\_OFFSET = const(0xd3)

SET\_COM\_PIN\_CFG = const(0xda)

SET\_DISP\_CLK\_DIV = const(0xd5)

SET\_PRECHARGE = const(0xd9)

SET\_VCOM\_DESEL = const(0xdb)

SET\_CHARGE\_PUMP = const(0x8d)

class SSD1306:

def \_\_init\_\_(self, width, height, external\_vcc):

self.width = width

self.height = height

self.external\_vcc = external\_vcc

self.pages = self.height // 8

# Note the subclass must initialize self.framebuf to a framebuffer.

# This is necessary because the underlying data buffer is different

# between I2C and SPI implementations (I2C needs an extra byte).

self.poweron()

self.init\_display()

def init\_display(self):

for cmd in (

SET\_DISP | 0x00, # off

# address setting

SET\_MEM\_ADDR, 0x00, # horizontal

# resolution and layout

SET\_DISP\_START\_LINE | 0x00,

SET\_SEG\_REMAP | 0x01, # column addr 127 mapped to SEG0

SET\_MUX\_RATIO, self.height - 1,

SET\_COM\_OUT\_DIR | 0x08, # scan from COM[N] to COM0

SET\_DISP\_OFFSET, 0x00,

SET\_COM\_PIN\_CFG, 0x02 if self.height == 32 else 0x12,

# timing and driving scheme

SET\_DISP\_CLK\_DIV, 0x80,

SET\_PRECHARGE, 0x22 if self.external\_vcc else 0xf1,

SET\_VCOM\_DESEL, 0x30, # 0.83\*Vcc

# display

SET\_CONTRAST, 0xff, # maximum

SET\_ENTIRE\_ON, # output follows RAM contents

SET\_NORM\_INV, # not inverted

# charge pump

SET\_CHARGE\_PUMP, 0x10 if self.external\_vcc else 0x14,

SET\_DISP | 0x01): # on

self.write\_cmd(cmd)

self.fill(0)

self.show()

def poweroff(self):

self.write\_cmd(SET\_DISP | 0x00)

def contrast(self, contrast):

self.write\_cmd(SET\_CONTRAST)

self.write\_cmd(contrast)

def invert(self, invert):

self.write\_cmd(SET\_NORM\_INV | (invert & 1))

def show(self):

x0 = 0

x1 = self.width - 1

if self.width == 64:

# displays with width of 64 pixels are shifted by 32

x0 += 32

x1 += 32

self.write\_cmd(SET\_COL\_ADDR)

self.write\_cmd(x0)

self.write\_cmd(x1)

self.write\_cmd(SET\_PAGE\_ADDR)

self.write\_cmd(0)

self.write\_cmd(self.pages - 1)

self.write\_framebuf()

def fill(self, col):

self.framebuf.fill(col)

def pixel(self, x, y, col):

self.framebuf.pixel(x, y, col)

def scroll(self, dx, dy):

self.framebuf.scroll(dx, dy)

def text(self, string, x, y, col=1):

self.framebuf.text(string, x, y, col)

class SSD1306\_I2C(SSD1306):

def \_\_init\_\_(self, width, height, i2c, addr=0x3c, external\_vcc=False):

self.i2c = i2c

self.addr = addr

self.temp = bytearray(2)

# Add an extra byte to the data buffer to hold an I2C data/command byte

# to use hardware-compatible I2C transactions. A memoryview of the

# buffer is used to mask this byte from the framebuffer operations

# (without a major memory hit as memoryview doesn't copy to a separate

# buffer).

self.buffer = bytearray(((height // 8) \* width) + 1)

self.buffer[0] = 0x40 # Set first byte of data buffer to Co=0, D/C=1

self.framebuf = framebuf.FrameBuffer1(memoryview(self.buffer)[1:], width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.temp[0] = 0x80 # Co=1, D/C#=0

self.temp[1] = cmd

self.i2c.writeto(self.addr, self.temp)

def write\_framebuf(self):

# Blast out the frame buffer using a single I2C transaction to support

# hardware I2C interfaces.

self.i2c.writeto(self.addr, self.buffer)

def poweron(self):

pass

class SSD1306\_SPI(SSD1306):

def \_\_init\_\_(self, width, height, spi, dc, res, cs, external\_vcc=False):

self.rate = 10 \* 1024 \* 1024

dc.init(dc.OUT, value=0)

res.init(res.OUT, value=0)

cs.init(cs.OUT, value=1)

self.spi = spi

self.dc = dc

self.res = res

self.cs = cs

self.buffer = bytearray((height // 8) \* width)

self.framebuf = framebuf.FrameBuffer1(self.buffer, width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.low()

self.cs.low()

self.spi.write(bytearray([cmd]))

self.cs.high()

def write\_framebuf(self):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.high()

self.cs.low()

self.spi.write(self.buffer)

self.cs.high()

def poweron(self):

self.res.high()

time.sleep\_ms(1)

self.res.low()

time.sleep\_ms(10)

self.res.high()

**Device\_traits.py**

thermostatTemperatureSetpoint = 25.5

humiditySetpointPercent = 35

humiditySetpointPercentOn = True

parts\_settings = {}

# def setTemp(temp):

# global thermostatTemperatureSetpoint, parts\_settings

# thermostatTemperatureSetpoint = temp

# parts\_settings["temp"] = temp

# def setHumidity(humidity):

# global humiditySetpointPercentOn, parts\_settings

# parts\_settings["humidity"] = humidity

def get\_settings(gateway\_id, setting\_type, curr\_settings):

global parts\_settings

parts\_settings = curr\_settings

return {

"id": gateway\_id,

"type": setting\_type,

"devices": [red\_led\_trait(), blue\_led\_trait(), temp\_trait(), humidity\_trait()]

}

def red\_led\_trait():

return {

"id": "red\_led",

"name": "Red LED",

"type": "action.devices.types.LIGHT",

"nicknames": ["Red LED"],

"default\_names": ["LED", "Red LED"],

"traits": [

{

"name": "OnOff",

"values": {"on": parts\_settings["red\_led"]}

},

{

"name": "Brightness",

"values": {"brightness": parts\_settings["red\_led\_brightness"]}

}

],

"deviceInfo": {

"manufacturer": "IoT Master Class",

"model": "LED-BULB",

"hwVersion": "1.0",

"swVersion": "2.2.3"

}

}

def blue\_led\_trait():

return {

"id": "blue\_led",

"name": "Blue LED",

"type": "action.devices.types.LIGHT",

"nicknames": ["Blue LED"],

"default\_names": ["LED", "Blue LED"],

"traits": [

{

"name": "OnOff",

"values": {"on": parts\_settings["blue\_led"]}

},

{

"name": "Brightness",

"values": {"brightness": parts\_settings["blue\_led\_brightness"]}

}

],

"deviceInfo": {

"manufacturer": "IoT Master Class",

"model": "LED-BULB",

"hwVersion": "1.0",

"swVersion": "2.2.3"

}

}

def temp\_trait():

return {

"id": "sensor\_temp",

"name": "Temperature Sensor",

"type": "action.devices.types.THERMOSTAT",

"nicknames": ["Temperature"],

"default\_names": ["SENSOR", "Temperature"],

"traits": [

{

"name": "TemperatureSetting",

"values": {

"thermostatMode": "heat",

"activeThermostatMode": "heat",

"thermostatTemperatureSetpoint": parts\_settings["temp"],

"thermostatTemperatureAmbient": parts\_settings["temp"],

"thermostatHumidityAmbient": 90,

"thermostatTemperatureSetpointLow": 0,

"thermostatTemperatureSetpointHigh": 100

}

}

],

"deviceInfo": {

"manufacturer": "IoT Master Class",

"model": "LED-BULB",

"hwVersion": "1.0",

"swVersion": "2.2.3"

}

}

# parts\_settings["humidity\_ison"]

# parts\_settings["humidity"]

def humidity\_trait():

return {

"id": "sensor\_humidity",

"name": "Humidity Sensor",

"type": "action.devices.types.THERMOSTAT",

"nicknames": ["Humidity"],

"default\_names": ["SENSOR", "Humidity"],

"traits": [

{

"name": "TemperatureSetting",

"values": {

"thermostatMode": "cool",

"activeThermostatMode": "cool",

"thermostatTemperatureSetpoint": parts\_settings["humidity"],

"thermostatTemperatureAmbient": parts\_settings["humidity"],

"thermostatHumidityAmbient": 90,

"thermostatTemperatureSetpointLow": 0,

"thermostatTemperatureSetpointHigh": 100

}

}

],

"deviceInfo": {

"manufacturer": "IoT Master Class",

"model": "LED-BULB",

"hwVersion": "1.0",

"swVersion": "2.2.3"

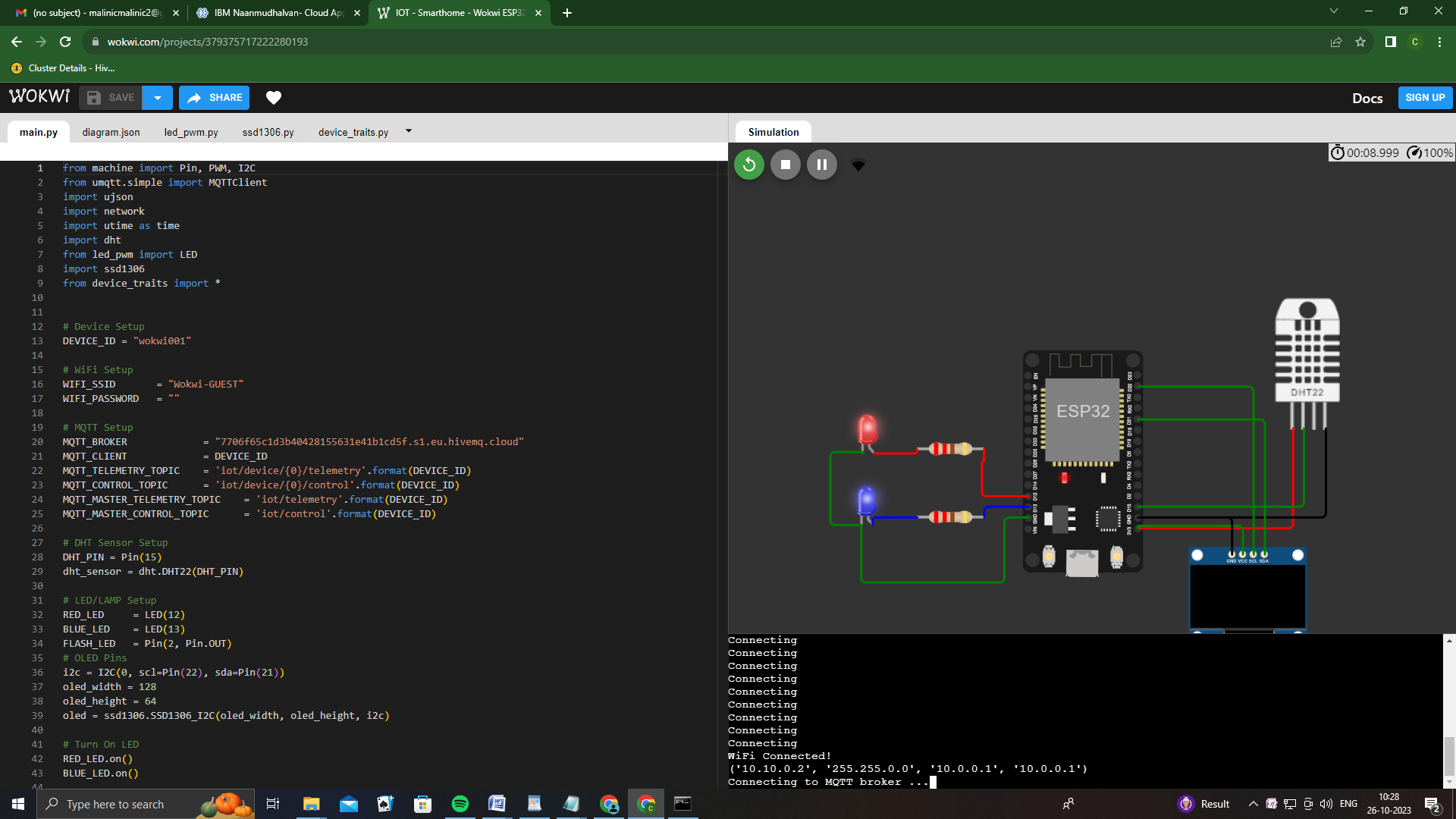
}

}

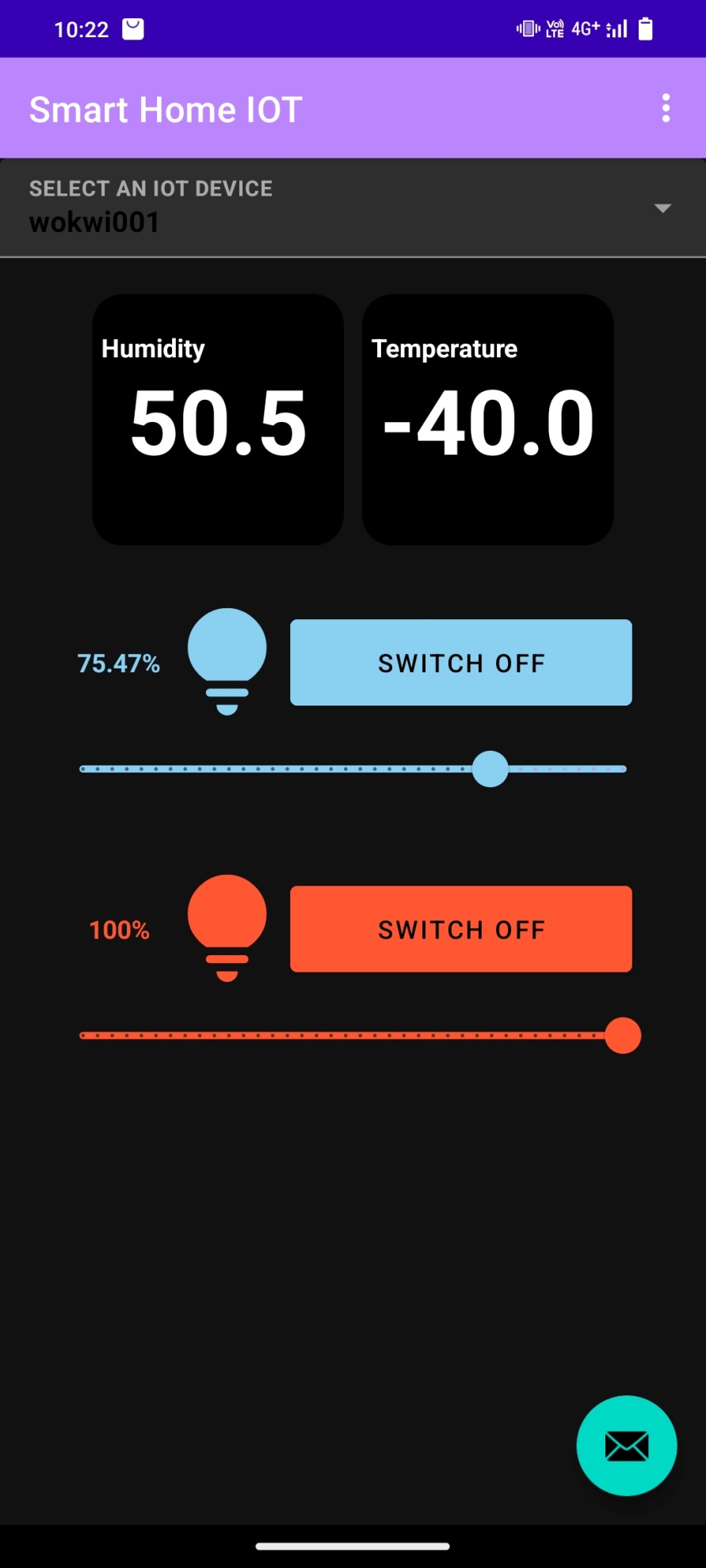
**Development Part 2:**

In this phase, Continue building the solution by implementing real-time data processing, automation, and storage. Use IBM Cloud Functions to process data and trigger automated routines.Store processed data in IBM Cloud Object Storage for analysis.

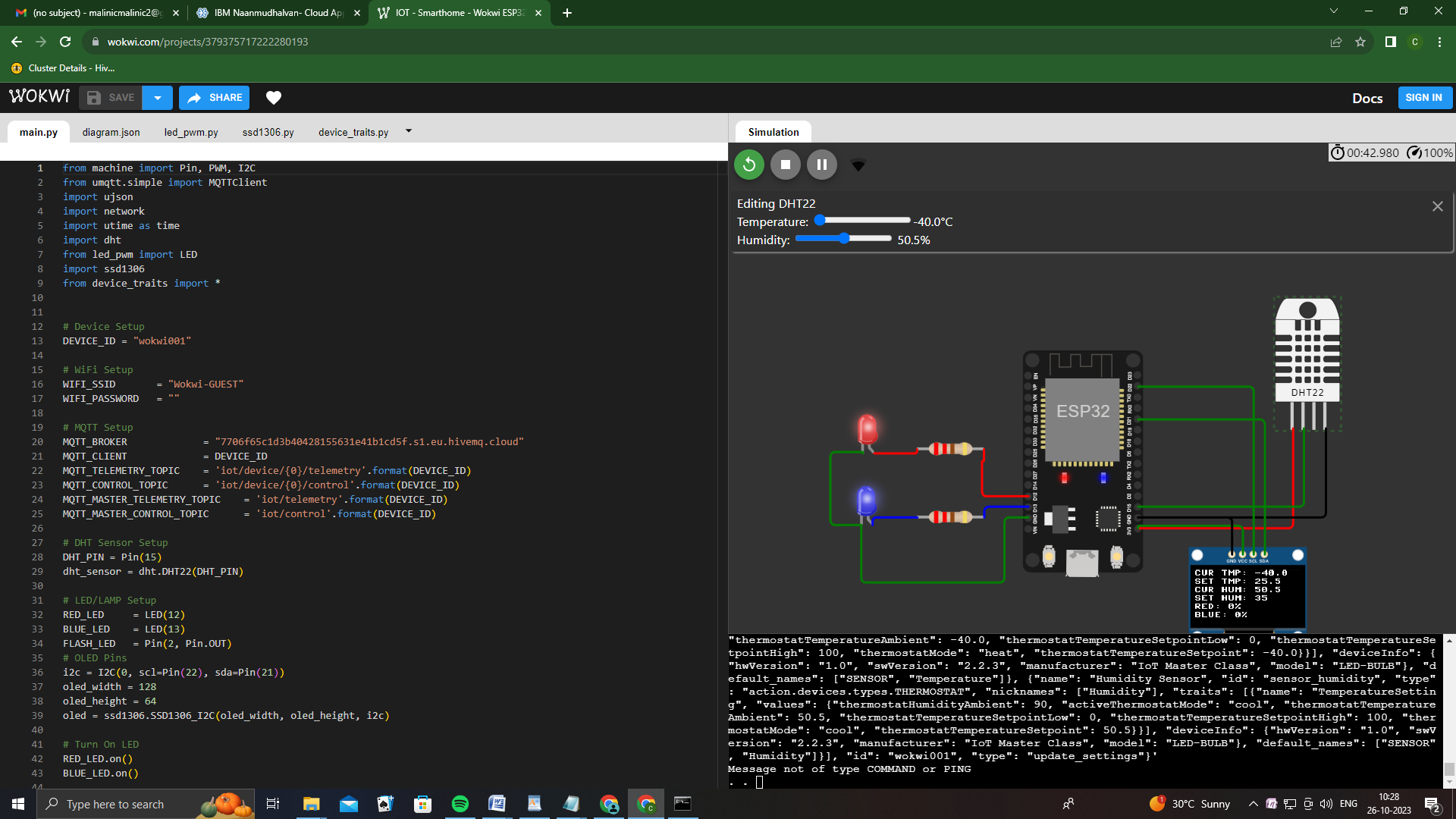
1. IMPLEMENTING REAL TIME DATA PROCESSING:



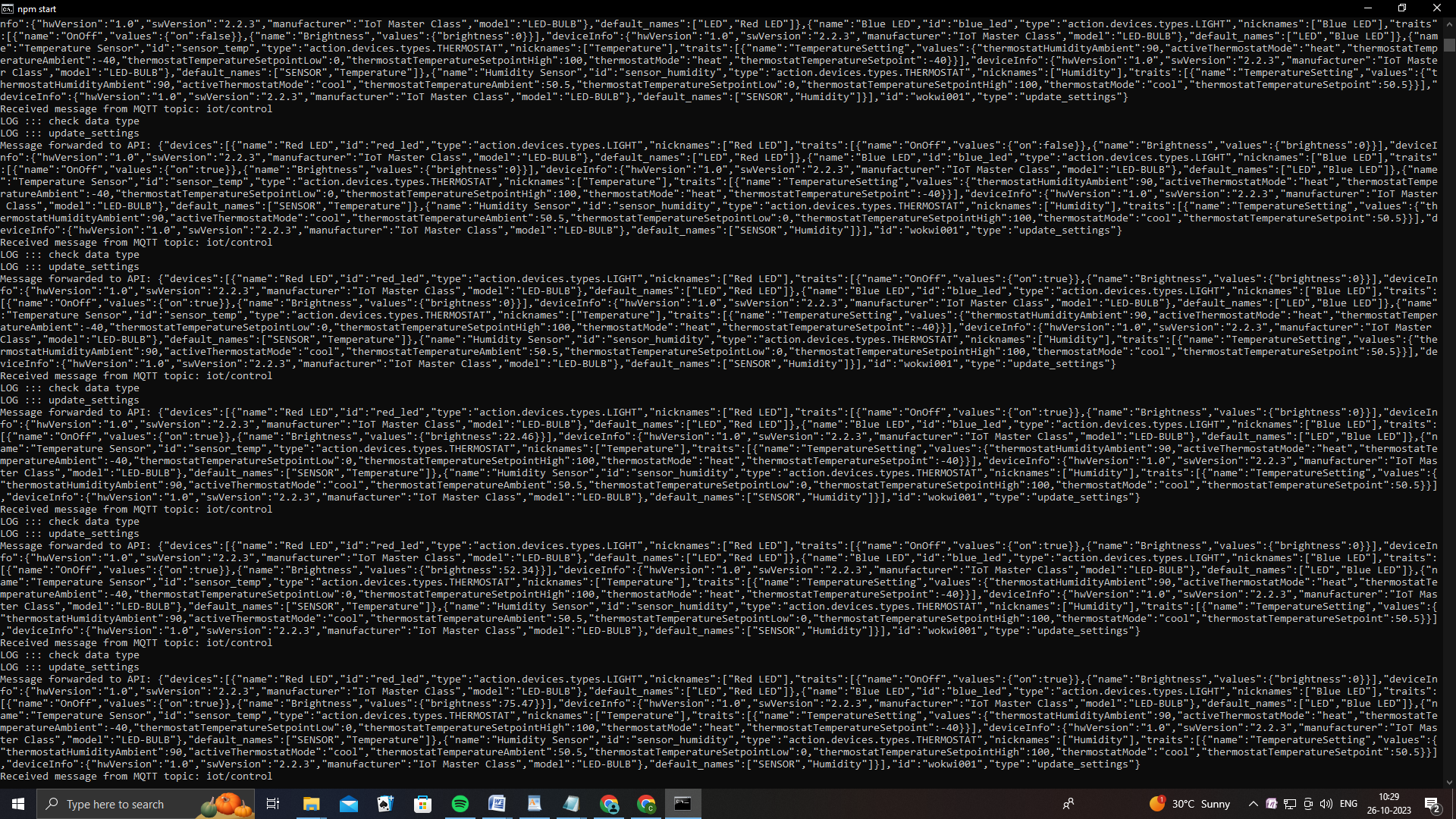
2.INPUT:



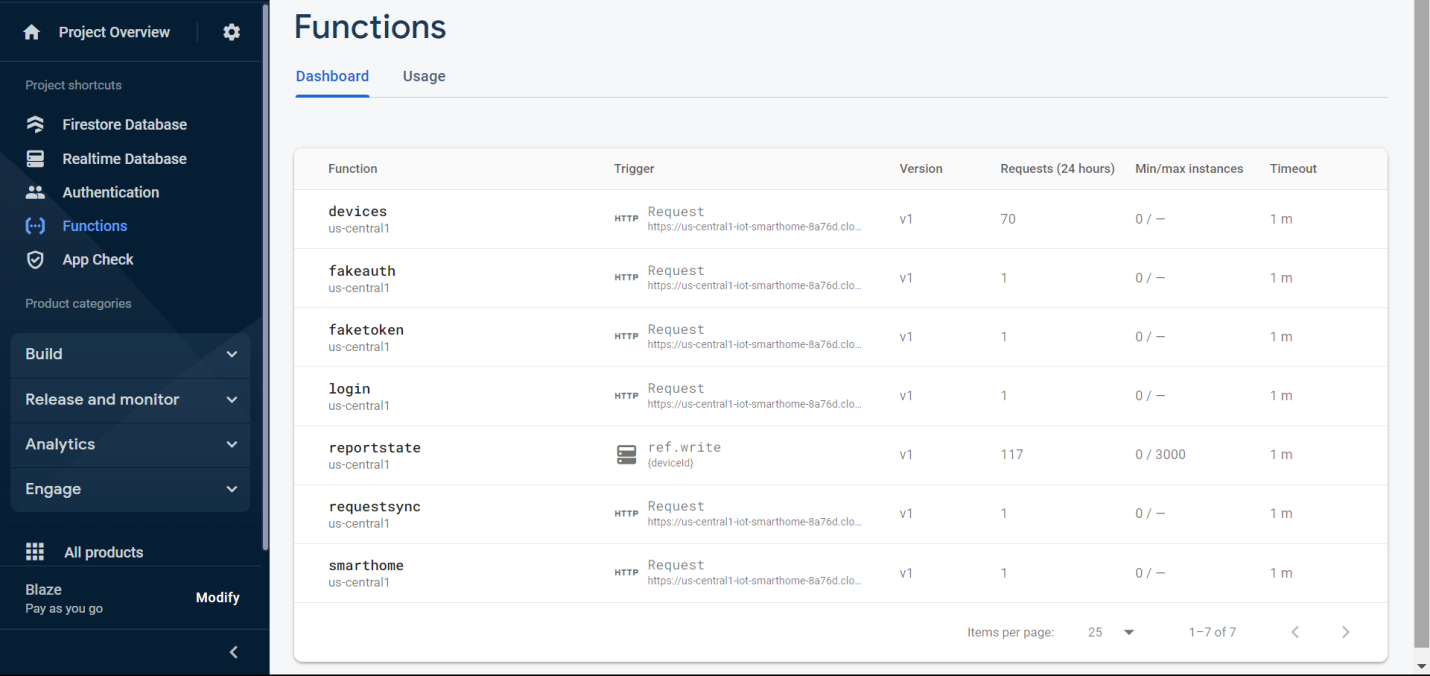
OUTPUT:

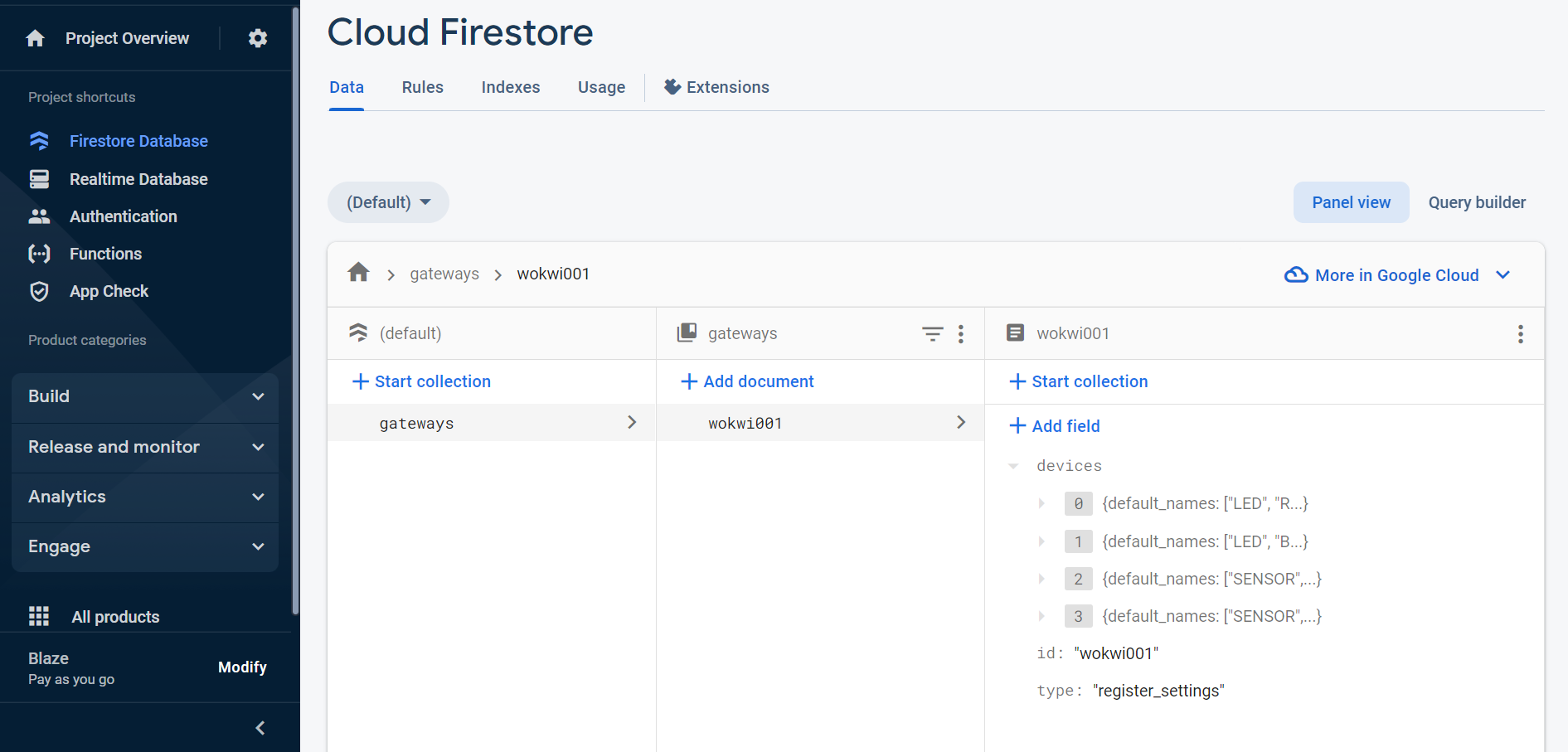


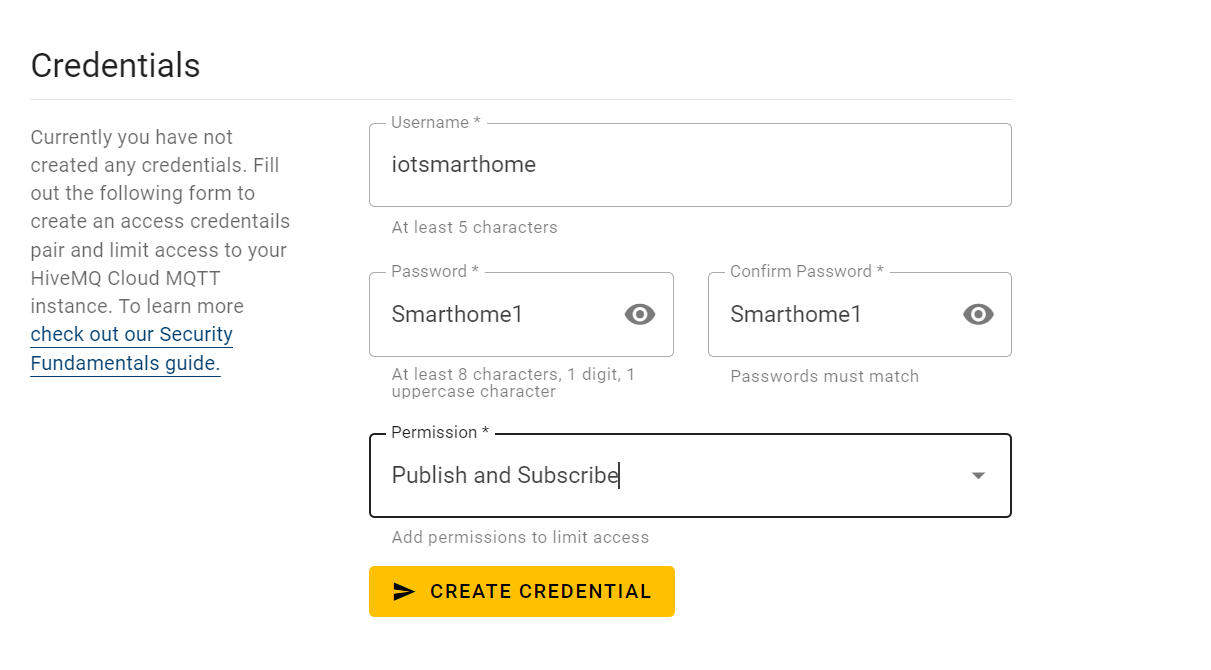
3.SERVER DATA CONNECTION:

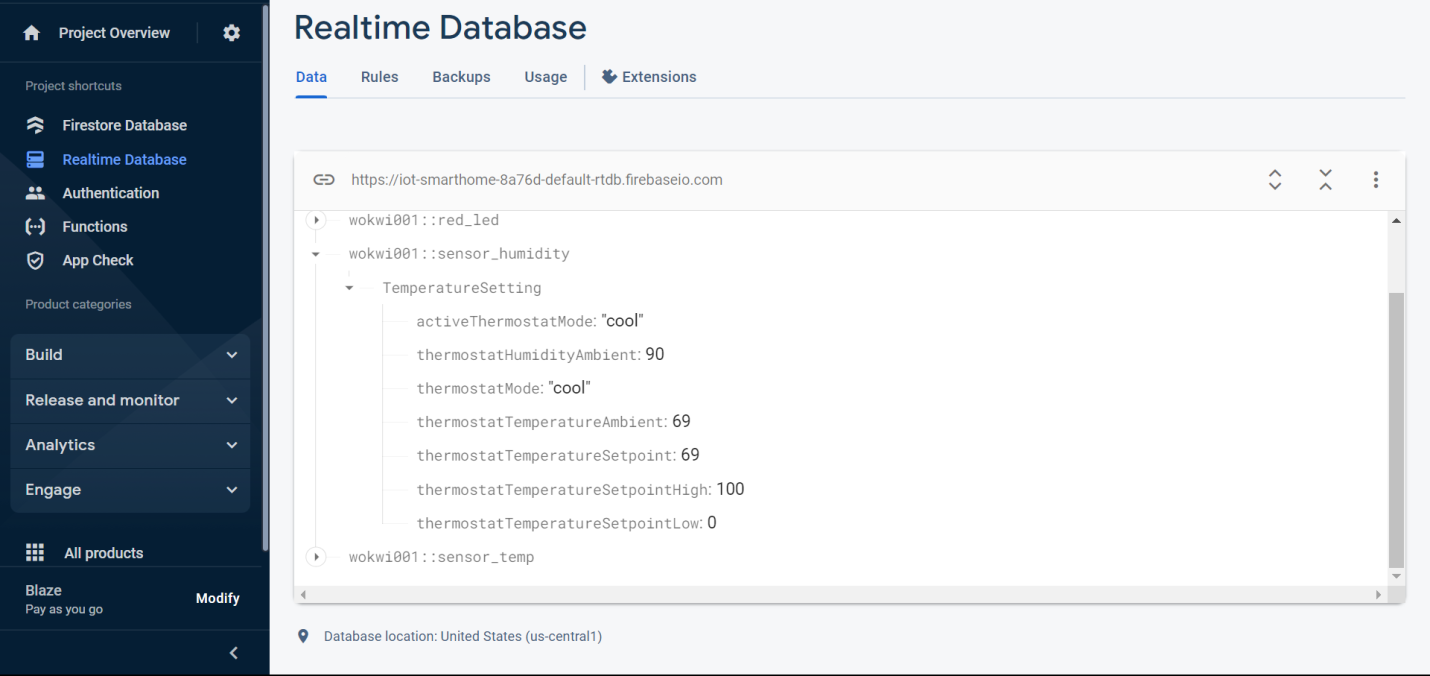


CLOUD DATA PROCESSING:









**Implementation details:**

● In this project, the implementation is done in wokwi simulator to on/off lights using our mobile application and to check the temperature and humidity.

● For this sensors like DHT and OLED board, ESP32 and Two LEDs are used.

● DHT sensor is used to monitor the temperature and humpty in the room.

● ESP32 which can perform as a complete standalone system or as a slave device to a host MCU, reducing communication stack overhead on the main application processor. Wi-Fi and Bluetooth functionality through its SPI/SDIO or 12C / UART interfaces.

● These devices are integrated in such a way that when simulation is done, LEDs can glow and temperature and humidity can be adjusted.

● The data generated in the real-time are made to store in the cloud database using cloud function that are written to the devices.

● The data stored at real-time can be used for future purpose.

● Separate code is written for the devices to connect to the cloud through wifi and for storing the data.

● A mobile application is developed and is connected to cloud to automate the devices whenever and wherever we want.

**Real-time processing and Data Storage:**

* For real-time processing, the simulator is started first.
* This establishes a connection to the cloud database through cloud functions.
* The connection between the application and cloud database is also established.
* Through the application, the lights can be made to on or off and temperature and humidity can be checked or maintained.
* The data that is simulated is stored to the database through cloud function at real-time.

**Conclusion:**

A generalized and flexible framework for smart object applications in a smart home environment was proposed. The framework consists of smart objects receiving messages from the cloud through a dedicated message-exchange protocol and interactive vocal interfaces, with serverless functions deployed on the cloud to monitor and control the objects. A practical use case based on this framework.