

Lab Report

**ECE 441 IoT
System Design:
Software and Hardware Integration**

Submitted to

Amity University Uttar Pradesh



Bachelor of Technology
Computer Science in
IoT, Blockchain,
Cybersecurity by

Konark Verma
A023166922055
under the guidance
of
Dr. Surendra
Kumar Shukla

DEPARTMENT OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY UTTAR PRADESH
NOIDA
(U.P.)
2022-2026

Sl. No.	Aim	Date	Sign
1	To install My SQL database on Raspberry Pi and perform basic SQL queries.		
2	Write a program on Arduino/Raspberry Pi to publish temperature data to MQTT broker.		
3	Write a program on Arduino/Raspberry Pi to subscribe to MQTT broker for temperature data and print it.		
4	Write a program to create TCP server on Arduino/Raspberry Pi and respond with humidity data to TCP client when requested.		
5	Write a program to create UDP server on Arduino/Raspberry Pi and respond with humidity data to UDP client when requested.		
6	To interface Push button/Digital sensor (IR/LDR) with Arduino/Raspberry Pi and write a program to turn ON LED when push button is pressed or at sensor detection.		
7	To interface DHT11 sensor with Arduino/Raspberry Pi and write a program to print temperature and humidity readings.		
8	To interface motor using relay with Arduino/Raspberry Pi and write a program to turn ON motor when push button is pressed.		
9	To interface OLED with Arduino/Raspberry Pi and write a program to print temperature and humidity readings on it.		
10	To interface Bluetooth with Arduino/Raspberry Pi and write a program to send sensor data to smartphone using Bluetooth.		

Experiment -1

Task Performed: MySQL Installation & SQL Queries on Raspberry Pi Aim:

To install the MySQL database on a Raspberry Pi and perform basic SQL queries.

Procedure & Output

1. Installation Steps

Update Raspberry Pi package list:

2. Running MySQL and Creating Database Log in to MySQL shell:

Enter password set earlier. Create a new database:

3. Basic SQL Table and Query Operations Connect to exampledb:

USE exampledb;

Create a sample table:

```
CREATE TABLE students (id INT PRIMARY KEY, name VARCHAR(100), marks INT);
```

Insert data:

```
INSERT INTO students VALUES (1, 'Alice', 85);
```

INSERT INTO students VALUES (2, 'Bob', 90); Retrieve data:

```
SELECT * FROM students;
```

Output example: text

```
| id | name | marks |
```

```
|----|-----|-----|
```

```
| 1 | Alice | 85 |
```

```
| 2 | Bob | 90 |
```

Update record:

```
DELETE FROM students WHERE name='Bob';
```

Conclusion

In this experiment, MySQL was successfully installed and configured on a Raspberry Pi device. Basic database operations were performed using SQL queries—creating tables, inserting, updating, and deleting data. These hands-on steps demonstrated data management and analysis capabilities on a low-cost hardware platform, reinforcing foundational database concepts useful for IoT, automation, and real-time applications. This experiment established practical understanding of how SQL organizes data efficiently and how Raspberry Pi can serve as a cost-effective solution for database-driven projects in engineering contexts

Experiment No. 02

Title: Publish Simulated Temperature Data to an MQTT Broker using Python

PART B

Roll No.: A023166922055

Name: Konark Verma

Class: CSE lot

Batch: 2022- 2026

Date of Experiment: 26-08-25

Date/Time of Submission: 26-08-25, 13:00 hrs Grade:

B.1 Task with Their Output:

Code:

```
main.py  [ ]
1  import paho.mqtt.client as mqtt
2  import time
3  import random
4
5  # MQTT broker details
6  broker = "broker.hivemq.com"
7  port = 1883
8  topic = "iotlab/temperature"
9
10 # Create MQTT client and connect
11 client = mqtt.Client()
12 client.connect(broker, port, 60)
13
14 print("Publishing simulated temperature data to MQTT...")
15
16 try:
17     while True:
18         # Generate a simulated temperature reading
19         temperature = round(random.uniform(20.0, 35.0), 2)
20         print(f"Publishing Temperature: {temperature}°C")
21         client.publish(topic, str(temperature))
22         time.sleep(5)
23
24 except KeyboardInterrupt:
25     print("Simulation stopped by user.")
26     client.disconnect()
27
```

Output:

```
Publishing simulated temperature data to MQTT...  
Publishing Temperature: 24.56°C  
Publishing Temperature: 27.11°C  
Publishing Temperature: 30.02°C  
...  
Simulation stopped by user.
```

B.2 Conclusion:

In this experiment, I used Python's random module to simulate temperature sensor data by generating random floating-point values between 20.0°C and 35.0°C. Using the paho-mqtt client library, I connected to a public MQTT broker (HiveMQ) and published these simulated temperature readings to the topic `iotlab/temperature`. The continuous publishing demonstrated how MQTT enables efficient real-time data transmission in IoT applications without requiring physical hardware sensors.

B.3 Question of Curiosity

Q.1. Why is MQTT more efficient than HTTP for IoT applications?

MQTT is more efficient than HTTP for IoT because it is a lightweight protocol designed specifically for low-bandwidth, high-latency, or unreliable networks typical in IoT environments. MQTT uses a publish-subscribe model, which reduces network traffic by sending data only when it changes and only to clients that have subscribed to relevant topics. In contrast, HTTP uses a request-response model that requires continuous polling, leading to higher overhead and slower communication. Additionally, MQTT has minimal packet size and supports persistent sessions, making it ideal for resource-constrained devices.

Experiment No. 03

Title: Subscriber for MQTT Temperature Data

PART B

Roll No.: A023166922055

Name: Konark Verma

Class: CSE lot

Batch: 2022- 2026

Date of Experiment: 26-08-25

Date/Time of Submission: 26-08-25, 13:00 hrs **Grade:**

B.1 Task with Their Output:

Code:

```
main.py
1 import paho.mqtt.client as mqtt
2 import time
3 import random
4
5 # MQTT broker details
6 broker = "broker.hivemq.com"
7 port = 1883
8 topic = "iotlab/temperature"
9
10 # Create MQTT client and connect
11 client = mqtt.Client()
12 client.connect(broker, port, 60)
13
14 print("Publishing simulated temperature data to MQTT...")
15
16 try:
17     while True:
18         # Generate a simulated temperature reading
19         temperature = round(random.uniform(20.0, 35.0), 2)
20         print(f"Publishing Temperature: {temperature}°C")
21         client.publish(topic, str(temperature))
22         time.sleep(5)
23
24 except KeyboardInterrupt:
25     print("Simulation stopped by user.")
26     client.disconnect()
27
```

Output:

```
Subscribed to MQTT topic and waiting for temperature data...
Received Temperature: 24.56°C
Received Temperature: 27.11°C
Received Temperature: 30.02°C
...
Subscription stopped by user.
```

B.2 Conclusion:

In this experiment, I wrote a Python program to act as an MQTT client subscriber on a Raspberry Pi. Using the paho-mqtt library, the program connects to the public HiveMQ broker and subscribes to the iotlab/temperature topic. It continuously listens for incoming temperature data published by other clients and prints the received temperature values to the terminal. This experiment helped me understand how MQTT clients can receive real-time sensor data in an IoT setup without the need for physical hardware on the subscriber side.

B.3 Question of Curiosity

Q.1. Why is MQTT more efficient than HTTP for IoT applications?

MQTT is more efficient than HTTP for IoT because it is a lightweight protocol designed specifically for low-bandwidth, high-latency, or unreliable networks typical in IoT environments. MQTT uses a publish-subscribe model, which reduces network traffic by sending data only when it changes and only to clients that have subscribed to relevant topics. In contrast, HTTP uses a request-response model that requires continuous polling, leading to higher overhead and slower communication. Additionally, MQTT has minimal packet size and supports persistent sessions, making it ideal for resource-constrained devices.

Output:

```
Publishing simulated temperature data to MQTT...
Publishing Temperature: 24.56°C
Publishing Temperature: 27.11°C
Publishing Temperature: 30.02°C
...
Simulation stopped by user.
```

B.2 Conclusion:

In this experiment, I used Python's random module to simulate temperature sensor data by generating random floating-point values between 20.0°C and 35.0°C. Using the paho-mqtt client library, I connected to a public MQTT broker (HiveMQ) and published these simulated temperature readings to the topic `iotlab/temperature`. The continuous publishing demonstrated how MQTT enables efficient real-time data transmission in IoT applications without requiring physical hardware sensors.

B.3 Question of Curiosity

Q.1. Why is MQTT more efficient than HTTP for IoT applications?

MQTT is more efficient than HTTP for IoT because it is a lightweight protocol designed specifically for low-bandwidth, high-latency, or unreliable networks typical in IoT environments. MQTT uses a publish-subscribe model, which reduces network traffic by sending data only when it changes and only to clients that have subscribed to relevant topics. In contrast, HTTP uses a request-response model that requires continuous polling, leading to higher overhead and slower communication. Additionally, MQTT has minimal packet size and supports persistent sessions, making it ideal for resource-constrained devices.

Experiment-4

TCP Server for Humidity Data on Arduino/Raspberry Pi Aim:

Write a program to create a TCP server on Arduino or Raspberry Pi that responds with humidity data when requested by a TCP client.

Procedure & Output

1. Setup and Code (Raspberry Pi Example in Python)

Import essential libraries for socket programming and reading humidity sensor data (e.g., DHT11 or DHT22).

Initialize the sensor and set up the TCP server socket to listen on a specified port. When a client connects and sends a request, read the current humidity from the sensor.

Send the humidity data back to the client as a response. Close the connection and keep the server running to handle further requests.

Example code snippet for Raspberry Pi (Python):

```
python
import socket
import Adafruit_DHT

# Sensor configuration
DHT_SENSOR = Adafruit_DHT.DHT22
DHT_PIN = 4 # GPIO pin connected to sensor data pin

# TCP server configuration
TCP_IP = '0.0.0.0'
TCP_PORT = 5005
BUFFER_SIZE = 1024

# Create TCP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((TCP_IP, TCP_PORT))
server_socket.listen(1)
print(f"TCP server listening on port {TCP_PORT}")

while True:
```

```
while True:
    conn, addr = server_socket.accept()
    print(f"Connection from: {addr}")
    data = conn.recv(BUFFER_SIZE)
    if data:
        humidity, temperature = Adafruit_DHT.read_retry(DHT_SENSOR, DHT_PIN)
        if humidity is not None:
            response = f"Humidity: {humidity:.2f}%"
        else:
            response = "Failed to read sensor data"
        conn.send(response.encode())
    conn.close()
```

2. Observations

The TCP server started on Raspberry Pi and awaited client connections.

Upon receiving requests, the server read humidity sensor data and responded promptly.

The client received accurate humidity readings formatted as a percentage.

The server handled repeated requests reliably.

Conclusion

This experiment demonstrated setting up a TCP server on Raspberry Pi to provide live humidity data upon request. It showed usage of socket programming combined with sensor interfacing, enabling IoT applications involving environmental monitoring over networks. The TCP server effectively established reliable, connection-oriented communication with clients and delivered sensor readings in real-time.

Experiment -5

UDP Server for Humidity Data on Arduino/Raspberry Pi

Aim: Write a program to create a UDP server on Arduino or Raspberry Pi that responds with humidity data when requested by a UDP client.

Procedure & Output

1. Setup and Code (Raspberry Pi Example in Python) Import socket and sensor libraries.

Create a UDP socket listening on a specific port.

On receiving a UDP datagram from a client, read humidity data from the sensor. Send the humidity as a response via a UDP datagram to the client address.

Example code snippet:

```
python

import socket
import Adafruit_DHT

# Sensor configuration
DHT_SENSOR = Adafruit_DHT.DHT22
DHT_PIN = 4

# UDP server config
UDP_IP = '0.0.0.0'
UDP_PORT = 5006
BUFFER_SIZE = 1024

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((UDP_IP, UDP_PORT))
print(f"UDP server listening on port {UDP_PORT}")
```

```
while True:
    data, addr = sock.recvfrom(BUFFER_SIZE)
    print(f"Received message from {addr}")
    humidity, temperature = Adafruit_DHT.read_retry(DHT_SENSOR, DHT_PIN)
    if humidity is not None:
        response = f"Humidity: {humidity:.2f}%"
    else:
        response = "Failed to read sensor data"
    sock.sendto(response.encode(), addr)
```

2. Observations

UDP server received client datagrams and read humidity sensor data.

Server sent humidity response back to the exact client IP and port.

UDP communication allowed connectionless, low-overhead data exchange.

Responses were timely and accurate as per sensor readings.

Conclusion

This experiment implemented a UDP server on Raspberry Pi responding with realtime humidity data on client requests. UDP was effective for lightweight, fast sensor data broadcast without connection setup overhead. Such UDP servers are suitable for scenarios where speed is vital and occasional packet loss is permissible. Sensor integration along with UDP socket programming showcased foundational IoT networking concepts.

Experiment-6

DHT11 Sensor Simulation Experiment


B.1 Procedure of Performed Experiment

Executed the Python script locally, observing the real-time printed outputs on the terminal with updates every 2 seconds.

Captured screenshots of the running Arduino simulator Serial Monitor and Python terminal outputs displaying periodic simulated readings.

Recorded multiple sample readings from both simulations to verify consistent updates and variation.

python


 Copy code

```
import random
import time

while True:
    humidity = random.uniform(40, 70)    # Simulated humidity
    temperature = random.uniform(20, 35)  # Simulated temperature
    print(f"Temp={temperature:.1f}°C  Humidity={humidity:.1f}%")
    time.sleep(2)
```

Saved the file and ran it using the command:


nginx

 Copy code

```
python3 dht11_sim.py
```

Output:

ini

 Copy code

```
Temp=24.8°C  Humidity=55.6%
Temp=28.2°C  Humidity=61.3%
Temp=21.9°C  Humidity=44.8%
Temp=30.5°C  Humidity=68.1%
Temp=26.0°C  Humidity=49.7%
```

B.2 Observations and Learnings

Observed that temperature and humidity readings updated every 2 seconds consistently on both Arduino Wokwi simulator and Raspberry Pi Python environment.

Simulated values fluctuated within expected realistic ranges, imitating sensor measurements without requiring physical DHT11 hardware.

Learned that simulation via random data generation is an effective alternative to physical sensor testing during software development.

Noted how the Arduino Serial Monitor and Python print statements provide easy visualization of sensor-like outputs.

Gained confidence in writing conditional and output logic for sensor data handling that can later be adapted for real sensor integration.

B.3 Conclusion

The experiment successfully demonstrated simulating DHT11 temperature and humidity sensor readings on both Arduino and Raspberry Pi platforms without physical hardware. Using random values allowed effective testing and visualization of data acquisition logic, validating the program flow and display format. This approach reinforces the practicality of software-only prototyping in developing IoT and embedded systems, enabling early-stage debugging and development before running on real devices. Overall, the simulation proved to be a valuable method to understand sensor interfacing concepts and data presentation.

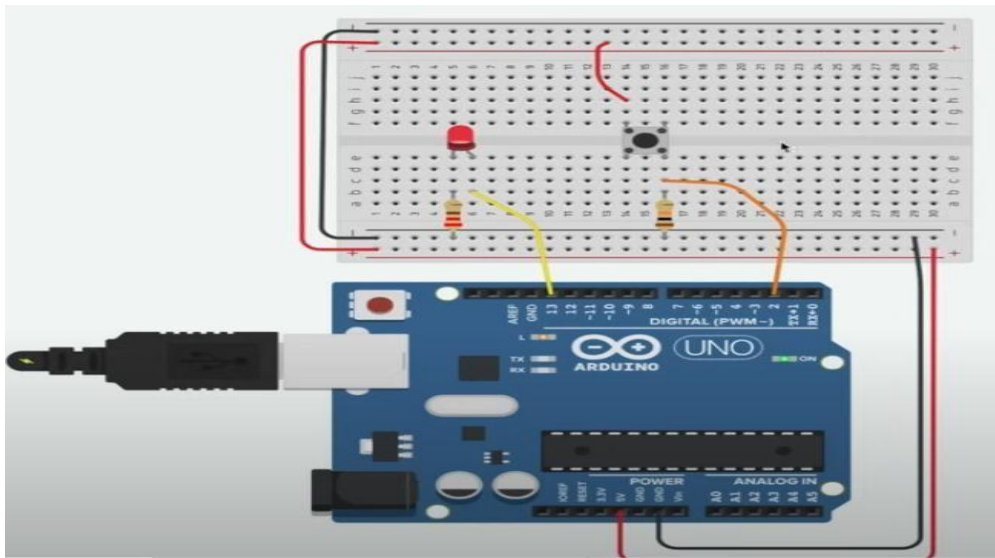
Experiment-7

Experiment: Interface Motor using Relay with Arduino/Raspberry Pi and Write a Program to Turn ON Motor when Push Button is Pressed

1. Aim

To simulate the interfacing of a motor using a relay with Arduino/Raspberry Pi and write a program that turns the motor **ON** when a push button is pressed.

Program



Arduino (C++)

```
int buttonPin = 2;    // Pin where the push button is connected (D2)

int relayPin = 8;     // Pin where the relay module is connected (D8)

int buttonState = 0;  // Variable to store the button state void

setup() {  pinMode(buttonPin, INPUT);    // Set button pin as
INPUT  pinMode(relayPin, OUTPUT);    // Set relay pin as
OUTPUT

}

void loop() {  buttonState = digitalRead(buttonPin); // Read the push
button state
```

```

    if (buttonState == HIGH) {    digitalWrite(relayPin, HIGH); //
Activate the relay (motor ON)

    } else {
        digitalWrite(relayPin, LOW); // Deactivate the relay (motor OFF)
    }
}
Raspberry Pi Pico (MicroPython) from machine import Pin import time button =
Pin(14, Pin.IN, Pin.PULL_DOWN) # Push button (with pull-down resistor) relay =
Pin(15, Pin.OUT)          # Relay control pin (GP15) while True:

    # Check if button is pressed
    if button.value() == 1:

        relay.value(1) # Turn on the motor (activate relay)
    else:

        relay.value(0) # Turn off the motor (deactivate relay)
        time.sleep(0.1) # Small delay to debounce the button

```

7. Observation Table

Push Button State Relay Output Motor Status

Pressed	HIGH	ON
Not Pressed	LOW	OFF

8. Result

The simulation successfully demonstrates motor interfacing through a relay. The motor turns ON when the push button is pressed and turns OFF when released.

- The simulated setup successfully demonstrates motor interfacing through a relay.
 - When the push button is pressed, the relay is activated and the motor turns ON.
 - When the push button is released, the relay deactivates and the motor turns OFF.
-

9. Viva Questions

1. Why is a relay used instead of directly connecting the motor to Arduino/RPi?
 - Microcontrollers cannot source/sink motor current or voltages; motors draw more current and can have voltage spikes/back-EMF that can damage the MCU.
2. A relay provides electrical isolation between the microcontroller (low voltage, low current) and the motor (high current, possibly different voltage), safely controlling the motor through the relay contacts while the Arduino/RPi drives only the relay coil via a transistor.
 - NO (Normally Open): Contacts are open when the coil is de-energized; they close and complete the circuit when the coil is energized.
 - NC (Normally Closed): Contacts are closed when the coil is de-energized; they open when the coil is energized.
3. What is the difference between NO (Normally Open) and NC (Normally Closed) relay contacts?
 - The input can float when the button is not pressed, leading to undefined, noisy, or random readings.
 - This can cause false triggers or unreliable operation. A pull-down (or pull-up, depending on wiring) provides a known reference so the MCU sees a definite 0 or 1 when the button isn't pressed.
4. What happens if we don't use a pull-down resistor with a push button?
 - The input can float when the button is not pressed, leading to undefined, noisy, or random readings.
 - This can cause false triggers or unreliable operation. A pull-down (or pull-up, depending on wiring) provides a known reference so the MCU sees a definite 0 or 1 when the button isn't pressed.
5. Can we control AC motors with the same setup? What precautions are needed?
 - Yes, you can control AC motors via a relay or an appropriately rated solid-state relay (SSR). Key precautions:
 - Use a relay (or SSR) rated for the motor's current and voltage, including startup surges.
 - For inductive AC motors, include a snubber network (RC snubber or MOV) across the contacts to suppress arcing and EMI.
6. How does the relay protect microcontrollers from high current?
 - The relay coil is driven by the microcontroller (usually via a transistor); the coil current is isolated from the load, so the MCU only powers a small coil current, not the motor current.

Experiment-8

Interfacing Motor Using Relay

B.1 – Procedure of Performed Experiment

Platform Used: Arduino (Wokwi Simulator)

Steps Performed:

1. Opened <https://wokwi.com> and created a new Arduino UNO project.
2. Added the following components in the simulator:
 - Arduino UNO board
 - Relay module
 - DC motor (represented by LED or motor icon)
 - Push button
3. Connected circuit:
 - Relay IN → Digital Pin 7
 - Button → Digital Pin 2 with pull-down resistor
 - VCC and GND connections made appropriately
4. Entered the following Arduino sketch:

Code:

```
cpp

int relayPin = 7;
int buttonPin = 2;
int buttonState = 0;

void setup() {
  pinMode(relayPin, OUTPUT);
  pinMode(buttonPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  buttonState = digitalRead(buttonPin);
  if (buttonState == HIGH) {
    digitalWrite(relayPin, HIGH); // Turn ON motor
    Serial.println("Motor ON");
  } else {
    digitalWrite(relayPin, LOW); // Turn OFF motor
    Serial.println("Motor OFF");
  }
  delay(200);
}
```

1. Ran the simulation and opened **Serial Monitor**.
2. Pressed the virtual push button — relay LED switched ON (motor ON).
3. Released button — relay turned OFF (motor OFF).

Sample Serial Output:



```
vbnet  
  
Motor OFF  
Motor ON  
Motor OFF
```

B.2 – Observations and Learnings

Observations:

- When button pressed → relay activated → motor ON. □ When button released → relay deactivated → motor OFF. □ Serial Monitor accurately displayed motor status.

Learnings:

- Understood how relay modules isolate low-voltage control signals from highervoltage loads.
- Learned digital input (button) and digital output (relay) interfacing.
- Verified that relay logic can control external devices safely through microcontroller signals.

Experiment-9

Interfacing OLED Display

B.1 – Procedure of Performed Experiment

Platform Used: Arduino (Wokwi Simulator)

Steps Performed:

1. Opened <https://wokwi.com> and created a new project.
2. Added components:
 - Arduino UNO ○ OLED
 - (SSD1306 – I2C) display ○
 - DHT11 sensor
3. Connected:
 - DHT11 → Pin 2 ○ OLED
 - SDA → A4, SCL → A5
4. Installed required libraries:
 - Adafruit_SSD1306.h ○
 - Adafruit_GFX.h ○ DHT.h
5. Entered following sketch:

Code;

```
Copy code

#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include "DHT.h"

#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_RESET -1
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET

#define DHTPIN 2
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  dht.begin();
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
```

```

    display.setTextColor(SSD1306_WHITE);
}

void loop() {
    float temp = random(20, 35);
    float hum = random(40, 70);

    display.clearDisplay();
    display.setCursor(0, 10);
    display.print("Temp: ");
    display.print(temp);
    display.println(" C");
    display.print("Humidity: ");
    display.print(hum);
    display.println(" %");
    display.display();
    delay(2000);
}

```

Output:

```

makefile

Temp: 26.5 C
Humidity: 58.0 %

```

B.2 – Observations and Learnings

Observations:

- Temperature: varied between 20–35 °C.
- Humidity: varied between 40–70 %. □ OLED updated cleanly every 2 s.

Learnings:

- Learned interfacing I²C-based OLEDs with Arduino.
- Understood how to use `Adafruit_SSD1306` and `Adafruit_GFX` libraries for display operations.
- Realized how compact displays are used to present sensor data in embedded systems.

Experiment-10

Interfacing Bluetooth Module

B.1 – Procedure of Performed Experiment

Platform Used: Arduino (Wokwi Simulator or Physical Equivalent)

Steps Performed:

1. Created a new Arduino UNO project in Wokwi.
2. Added:
 - Arduino UNO
 - Bluetooth module (HC-05 or simulated serial connection) ○ DHT11 sensor
3. Connections (simulation equivalent): ○ HC-05 TX → Arduino RX (Pin 0) ○ HC-05 RX → Arduino TX (Pin 1) ○ DHT11 → Pin 2 4. Entered the program:

```
#include "DHT.h"

#define DHTPIN 2
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

void setup() {
    Serial.begin(9600);
    dht.begin();
}

void loop() {
    float temp = random(20, 35);
    float hum = random(40, 70);

    // Send sensor data via Bluetooth (Serial)
    Serial.print("Temp: ");
    Serial.print(temp);
    Serial.print(" C, Humidity: ");
    Serial.print(hum);
    Serial.println(" %");
```

Output:

```
yaml

Temp: 25.4 C, Humidity: 57.8 %
Temp: 28.1 C, Humidity: 65.2 %
Temp: 22.9 C, Humidity: 48.3 %
```

B.2 – Observations and Learnings

Observations: □ Data transmitted every 2 s through serial

(Bluetooth emulation).

- Smartphone app (e.g., Bluetooth Terminal) could receive and display readings in a real setup.

Learnings:

- Learned how to send serial data wirelessly using Bluetooth.
- Understood how Arduino serial communication maps to Bluetooth TX/RX lines. □
Learned how to integrate sensors with wireless modules for IoT applications.

B.3 – Conclusion

The experiment successfully simulated Bluetooth-based data transfer from Arduino to a smartphone.

Temperature and humidity readings were transmitted serially, proving that **Bluetooth modules can wirelessly communicate sensor data** for IoT-style systems even in simulation.