Chapter 7 Integrating Real-Time Cloud Database

This chapter will look at integrating a real-time cloud-based database to enable remote monitoring of the IoT devices using Google Firebase. Using the Firebase HTTP request protocol, ESP32 can securely store sensor data in the cloud. This setting allows for scalable, real-time data management and exploration. We will also use the Thunkable platform to develop a mobile application that will interface with Firebase through the Application Programming Interface (API). This approach provides remote access to sensor data from the cloud database and displays it in a mobile application. Through this comprehensive approach, you will learn how to implement real-time data visualization and update in a mobile application, configure a cloud-based database and ensure that your IoT system is accessible and up to date at any location.

In this chapter, you will be able to

- Gain a foundational knowledge of Google Firebase and its role in IoT applications, focusing on its real-time database capabilities.
- Learn how to set up and configure the ESP32 to communicate with Google Firebase using HTTPS API for storing and retrieving sensor data.
- Understand and apply HTTP and HTTPS protocols to ensure secure data transfer between the ESP32 and Firebase.
- Understand the JavaScript Object Notation (JSON) data encoding format, which is commonly used in IoT applications for data interchange.
- Develop a responsive mobile application using Thunkable that can access, and display sensor data stored in Firebase in real-time.
- Implement real-time data visualization techniques within the mobile application to monitor sensor data dynamically.
- Explore advanced features of Firebase and Thunkable to extend the functionality of your IoT system, such as push notifications and alerts.

7.1 What is HTTP?

Hypertext Transfer Protocol (HTTP) facilitates data transfer over the Internet and allows users to access websites and other online resources through hypertext links [1], [2]. HTTP runs at higher levels of the network protocol stack and is an application layer protocol designed to transport data between networked devices. A server receives a request from a client computer over HTTP, and the server responds with a message as shown in **Figure *1***.

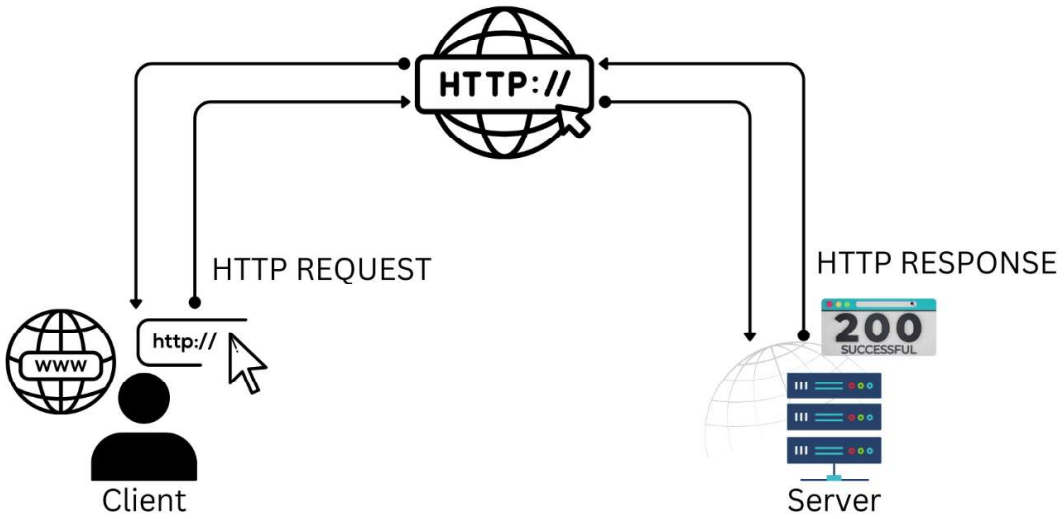


Figure 1 *HTTP Request and Response Model Structure.*

An HTTP request message containing the requested resource and any additional parameters is sent from the client to the server to start a request. After receiving the request message, the server uses its resources to process it and create a response message. The requested resource and any additional data are usually included in the response message that the server sends back to the client. After the client receives the response message, it processes it, typically by displaying the content in an application or rendering it in a web browser. If more requests are needed, the client can start the cycle over by sending other requests to the server.

An HTTP request is a basic mechanism used by Internet communication platforms such as web browsers to request information from servers to load websites. Every HTTP request made over the Internet carries a set of encrypted data containing various types of information. A typical HTTP request contains the following components:

- **HTTP Version Type**: Specifies the version of the HTTP protocol being used, such as HTTP/1.1 and above [3]. This ensures compatibility and proper communication between the client and the server.
- **Uniform Resource Locator (URL)**: Specifies the address location of the resources on the Internet that the client requests from the server [3], [4]. It contains the domain name and the path to the resource and helps the server find and serve the corresponding content. A URL for an HTTP/HTTPS request shown in **Figure 2** is normally made up of three or four components as highlighted below.

Figure 2 *Anatomy of Uniform Resource Locator (URL).*

The protocol to be used to access the resources on the Internet is identified by the *scheme*. It can be HTTP or HTTPS, a secure variant of HTTP. The *host* specifies which web server is requested. This is usually a domain name, but it can also be an IP address. Next, the *path* to the resource is displayed on the web server, this indicates the location of the physical file on the server. When a *query string* is used to retrieve specific content, it follows the path component that begins after the question mark (?) and consists of key-value pairs of the parameter [4].

- **HTTP Method**: Defines the action the client wants to perform on the resource. Common methods include GET to retrieve data, POST to send data, PUT to update data, and DELETE to remove data [5].
- **HTTP Request Headers**: Contain information stored in the set of key-value pairs, that help the server understand the context and requirements of the request such as the URL, content type, request method, and other data fields being requested [2].
- **HTTP Request Body**: Used in methods like POST and PUT, the body contains the actual data being sent to the server. This could be data in the form of JSON payloads, or other types of content that the server needs to process.

7.2 Getting Started with HTTP Request

Testing HTTP requests is a crucial step in ensuring that your applications communicate effectively with servers. One of the most powerful tools for testing HTTP requests is the Postman desktop client. Postman provides a user-friendly interface for sending, and analyzing HTTP requests, making it a valuable tool for developers. In this section, we'll explore how to get started with HTTP requests using Postman by leveraging the JSONPlaceholder API, which provides fake data for testing purposes.

7.2.1 Setting Up Postman Client

1. **Download and Install Postman**: Visit the Postman website and download the Postman desktop client for your operating system. Install the application by following the on-screen instructions [6].
2. **Launch Postman and Create Workspace**: Open the Postman application after installation. You will be greeted with a user-friendly interface where you can create and manage your HTTP requests. To start with, go to "**Workspaces**" > "**Create Workspace**", use the default Blank workspace then named your workspace "**JSON Placeholder HTTP Request**" or anything you want.

7.2.2 Creating HTTP Request with Postman Client

For testing and prototyping, fake data can be accessed for free via the JSONPlaceholder API, an online Representational State Transfer (REST) API. It has endpoints for retrieving and working with various resources including users, comments, posts, and more [7]. This API is used to test HTTP requests in Postman.

1. Sending an HTTP Request: From your Postman Workspace created earlier, click New at the upper right side of your Workspace tab as shown in **Figure 3**, then choose HTTP to create an HTTP request workspace.

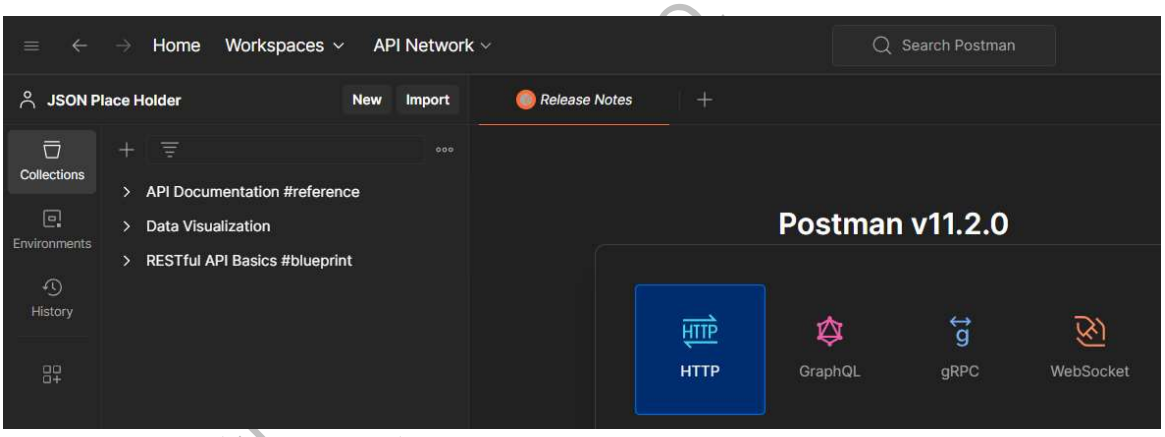


Figure 3 *Creating a Postman HTTP request workspace.*

Enter the URL https://jsonplaceholder.typicode.com/posts in the field from which we can access the content of the fake post using the HTTP GET method . Enter the key-value pair such as "**userID = 2**" or "**id = 1**" in the Query Params to filter the content we want to access, and then click Send to complete the request. **Figure 4** shows the interface of this process for reference.

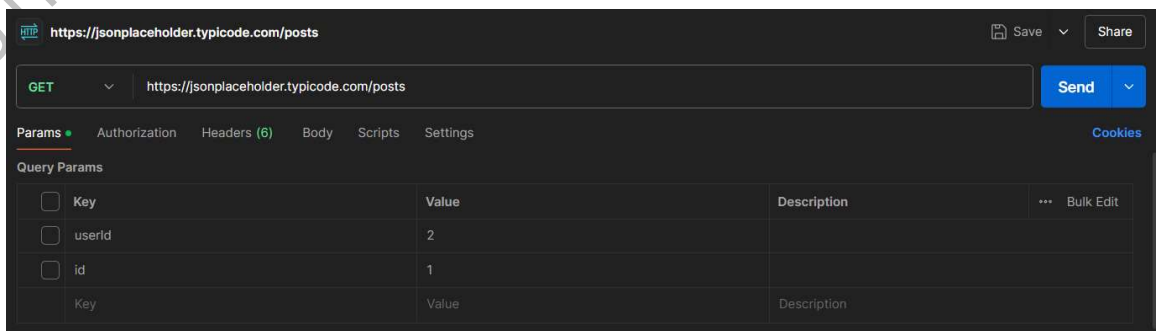


Figure 4 *Sending a HTTP GET request via Postman.*

2. Analyzing the HTTP Response: Postman displays the server's response at the bottom under HTTP Body. You should see a JSON array containing multiple posts with fields such as "**userId**", "**id**", "**title**", and "**body**", with content written in Latin as shown in **Figure 5** Viewing HTTP response from JSONPlaceholder server.. The HTTP Request Headers and Network status can also be viewed.

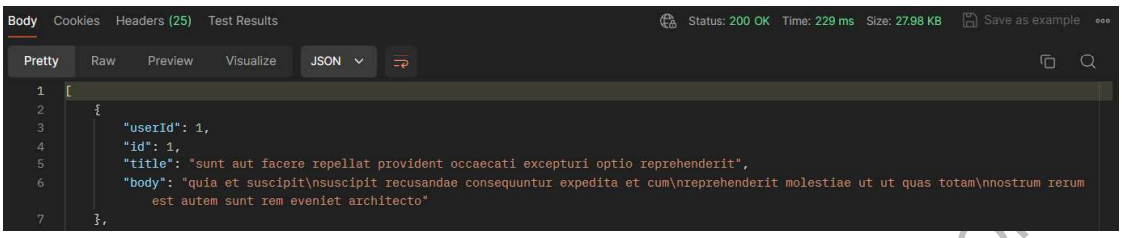


Figure 5 *Viewing HTTP response from JSONPlaceholder server.*

An HTTP response is what web clients receive from an Internet server in answer to an HTTP request. These responses communicate valuable information based on what was asked for in the HTTP request. A typical HTTP response contains:

- an HTTP status code
- HTTP response headers
- HTTP body (optional)

Let's break these down. HTTP status codes are 3-digit codes most often used to indicate whether an HTTP request has been completed [8]. Status codes are broken into the following 5 blocks:

- 1xx Informational (e.g., 100 Continue, 102 Processing, etc.)
- 2xx Success (e.g., 200 OK, 201 Created, etc.)
- 3xx Redirection (e.g., 302 Found, 305 Use Proxy, etc.)
- 4xx Client Error (e.g., 400 Bad Request, 404 Not Found, etc.)
- 5xx Server Error (e.g., 502 Bad Gateway, 504 Gateway Timeout, etc.)

Status codes starting with the number "2" indicate a success. For example, after a Postman client requests a content to the JSON Placeholder, we seen responses have a status code of "200 OK", indicating that the request was properly completed. If the answer begins with a "4" or a "5," there is an error and the content will not be displayed. A status code that begins with "4" indicates a client-side error, it is very common to see a "404 NOT FOUND" status code when there is a typo in a URL. A status code that starts with "5" means that an error occurred on the server side.

7.3 Getting Started with HTTP Request on ESP32

In the previous section, we explored how to make HTTP requests to the JSONPlaceholder API using the Postman client. Now we replicate the same process using ESP32 microcontroller with WiFi.h library to call WiFiClient class to create HTTP requests [9]. The ESP32's built-in Wi-Fi capabilities make it a powerful tool for IoT applications, allowing data to be sent and received over the Internet. In this section, we show how to use HTTP requests on the ESP32 to request data from the JSONPlaceholder API. **Listing 1** is the example of Arduino sketch for ESP32 sending HTTP GET requests to JSONPlaceholder API is shown below.

Listing 1 *ESP32 Implementation of HTTP GET Request with the JSONPlaceholder server*

```
#include <WiFi.h>

// Create Wi-Fi client object
WiFiClient client;
```

The WiFi.h library is included to enable Wi-Fi functionality on the ESP32. The WiFiClient class is part of the WiFi.h library that enables attempts to establish a Transmission Control Protocol (TCP) connections [9]. It is used to create client objects that can connect to remote servers over the network, facilitating communication over protocols such as HTTP, and HTTPS. The WiFiClient class creates an object called client that handles network communication.

```
// Defines the SSID and password of the Wi-Fi network
const char* SSID = "PipoyT11";
const char* PASSWORD = "data032722";

const int HTTP_PORT = 80;                              // Server port
const char* HTTP_METHOD = "GET ";                      // HTTP request method (e.g.,
POST)
const char* HOST = "jsonplaceholder.typicode.com";     // Domain of the server
const String URL_PATH = "/posts";                      // Contains content path
location from server
const String URL_QUERY = "?userId=1";                  // Contains parameters of the
content to be request
```

Define the SSID and password for your Wi-Fi network. Also, define the HTTP method, server host, URL path, and the URL string query of the JSONPlaceholder API [7].

```
void setup() {
  Serial.begin(115200);

  // Set ESP32 WiFi to station mode
  WiFi.mode(WIFI_STA);

  // Initiate the Wi-Fi connection
  WiFi.begin(SSID, PASSWORD);

  Serial.print("Connecting...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(100);
  }
  // Print some information of the Wi-Fi network
  Serial.print("\nConnected to the ");
  Serial.println((String)WiFi.SSID() + " network");
  Serial.print("[*] RSSI: ");
  Serial.println((String)WiFi.RSSI() + " dB");
  Serial.print("[*] ESP32 IP: ");
  Serial.println(WiFi.localIP());
  Serial.println();
}
```

The setup function starts serial communication for debugging purposes at a baud rate of 115200. It then puts the ESP32 into station mode (WIFI_STA) so that it can connect to an existing Wi-Fi network using the WiFi.begin() function . It continuously checks the connection status until the connection is established. Once connected, the ESP32's SSID, RSSI and local IP address will be printed to indicate successful connection to the WiFi network.

```
void loop() {
  Serial.print("Connecting to ");
  Serial.println(HOST);

  // Create TCP (Transmission Control Protocol) connections
  if (!client.connect(HOST, HTTP_PORT)) {
    Serial.println("Connection failed");
```

```
    return;
  }
```

The loop function begins by establishing a TCP connection to the specified HOST (i.e. jsonplaceholder.typicode.com) at the specified HTTP_PORT (usually port 80 for HTTP). The connection function returns a Boolean value of true if the connection is successful and false if the connection fails. If the connection fails, "Connection Failed" is print on the Serial monitor to inform the user that the connection attempt failed and to retry the TCP connection.

```
  // Send HTTP request header
  client.print(HTTP_METHOD + URL_PATH + URL_QUERY + " HTTP/1.1\r\n" + "Host: " +
HOST + "\r\n" + "Connection: close\r\n\r\n");
```

This code constructs and sends an HTTP GET request to the server. It specifies the resource path, query parameters, HTTP version, host header, and connection header. The request is formatted according to the HTTP/1.1 standard and sent using the client.print() method. This request is sent to the server to request data from the specified path with the given query parameters.

```
  // Check connection if reach timeout (five seconds)
  unsigned long timeout = millis();
  while (client.available() == 0) {
    if (millis() - timeout > 5000) {
      Serial.println(">>> Client Timeout !");
      client.stop();
      return;
    }
  }
```

This segment implements a timeout mechanism to avoid waiting indefinitely for a server response. It uses millis() and returns the number of milliseconds since the ESP32 was powered on, which marks the start of the timeout period. The client.available() checks whether bytes are available to read from the server. Within the while loop, this line checks whether the current time minus the start time exceeds 5000 milliseconds (5 seconds). If more than 5 seconds have passed without data being received, the condition returns ">>> Client Timeout!" to the Serial monitor to indicate that the timeout period has been exceeded. Then it calls client.stop() to terminate the connection to the server and retry the availability check from the server.

```
  // Read response from server and print them to Serial monitor
  while (client.available()) {
    char response = client.read();
    Serial.print(response);
  }

  // Wait for 10-seconds for another HTTP GET request
  delay(10000);
}
```

The last part of the code is crucial for processing and displaying the server's response. It continuously checks whether data is available, reads each byte from the server using the client.read() function, and outputs it to the serial monitor. This allows you to observe the server's complete response, which is essential for debugging and verifying that your HTTP request was successful and the server responded as expected. If the server response, the HTTP response might look something like this:

```
HTTP/1.1 200 OK
Date: Wed, 03 Jul 2024 10:26:49 GMT
Content-Type: application/json; charset=utf-8
```

```
Transfer-Encoding: chunked
Connection: close
...
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio
reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et
cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt
rem eveniet architecto"
  },...
]
```

7.4 Migrating HTTP Request on ESP32 to HTTPS

In this section, you will learn how to migrate from HTTP to HTTPS requests on the ESP32. Transitioning to Hypertext Transfer Protocol Secure (HTTPS) is critical to improving the security of your IoT applications. It is an extension of HTTP and is often used to secure data transmitted over the Internet by encrypting it between the client and server. The protocol used to secure HTTP is called Transport Layer Security (TLS), although it was previously known as Secure Sockets Layer (SSL) [10]. TLS is an updated, more secure version of SSL. It encrypts data sent over the Internet, making it difficult for unauthorized parties to access or manipulate the transmitted information.

In the previous example, you are familiar with HTTP communication on ESP32, HTTPS is not very difficult to implement [11]. Three changes in the code and you're done.

1. **Change WiFiClient to WiFiClientSecure**
   `WiFiClient client;`
   changes to

   ```
   WiFiClientSecure client;
   ```

   The Arduino framework WiFi class WiFiClientSecure extends the basic WiFiClient class to support secure connections over HTTPS using TLS. After declaring client, you can use it to connect to HTTPS servers, send HTTP requests securely, and receive responses over a secure connection.

2. **Change Port from 80 to 443**
   `client.connect(host, 80);`
   changes to

   ```
   client.connect(host, 443);
   ```

   By convention, HTTP traffic uses port 80. This is the standard port for unencrypted Internet communication. HTTPS traffic uses the standard port 443. Data sent over HTTPS is encrypted using TLS/SSL, providing a secure channel.

3. **Change the host URL from http:// to https://**
   `client.println("GET http://www.site.com/ HTTP/1.1");`
   changes to

   ```
   client.println("GET https://www.site.com/ HTTP/1.1");
   ```

   When you connect to a web server over HTTP (e.g., http://www.site.com), your browser or HTTP client assumes that the server is listening on port 80. Similarly, when you connect to a web server over HTTPS (e.g., https://www.site.com), your browser or HTTPS client assumes that the server is listening on port 443 unless it specified.

4. **Set the Certificate Authority (CA) of the Server (Optional)**

Adding CA certificate of the server is optional because it doesn't affect the security of the communication. It just assures you that you are communicating with the correct server. If you don't provide the CA certificate, your communication will still be secure. To set the CA certificate on the ESP32, you need to get the CA certificate of the server you are connecting to. This certificate is usually available from the server connection details. We will use the JSONPlaceholder as example, using Google Chrome web browser, click the view site information icon next to the URL in the address bar as shown in **Figure 6**.

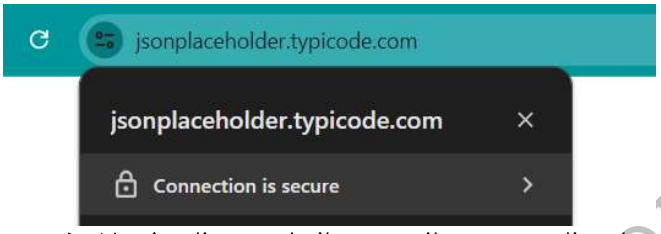


Figure 6 *Navigating website security connection type.*

A dropdown menu will appear with security information about the website, then click the "**Connection is secure**" to show the connection details. **Figure 7** show the type of security connection in the web server, at the bottom bar, click the "**Certificate is valid**" to show the SSL/TLS certificate viewer.

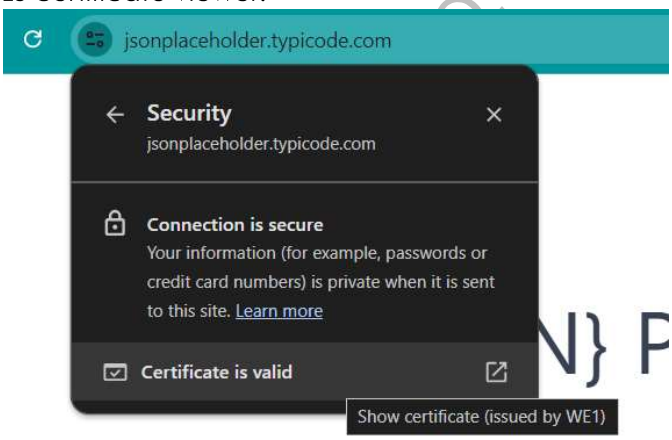


Figure 7 *Navigating SSL certificate security status.*

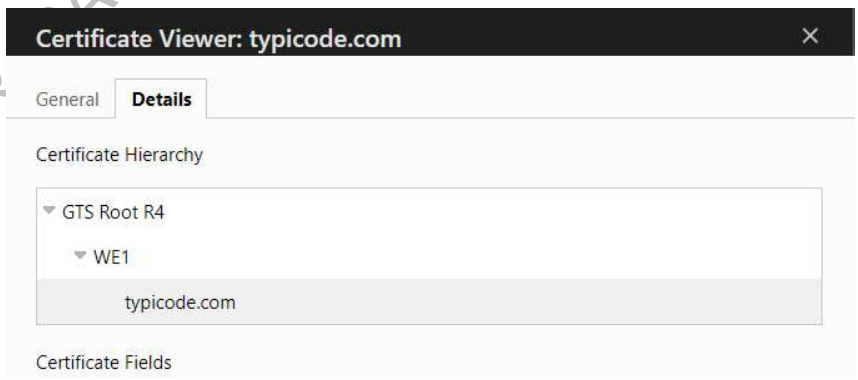


Figure 8 *Exporting SSL certificate from a Certificate Viewer.*

In the certificate dialog box, go to the "**Details**" tab shown in **Figure 8**, then click "**Export…**" button to save CA certificate on your device as .pem or .crt file format extension. Open the saved CA certificate file with a text editor such as note pad. Copy the entire contents of the certificate, including the -----BEGIN CERTIFICATE----- and -----

END CERTIFICATE----- lines. Paste the certificate content into your ESP32 code, as shown in the example below.

```
const char SERVER_CERT_CA[] PROGMEM = R"=====(
-----BEGIN CERTIFICATE-----
MIIDQTCCAimgAwIBAgITBmyfz5m/jAo54vB4ikPmljZbyjANBgkqhkiG9w0BAQsF
ADA5MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6
b24gUm9vdCBDQSAxMB4XDTE1MDUyNjAwMDAwMFoXDTM4MDExNzAwMDAwMFowOTEL
MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+rcdqsq08p8kDi1L
93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+OyQm
jgSubJrIqg0CAwEAAaNCMEAwDwYDVR0TAQH/BAUwAwEB/zAOBgNVHQ8BAf8EBAMC
AYYwHQYDVR0OBBYEFIQYzIU07LwMlJQuCFmcx7IQTgoIMA0GCSqGSIb3DQEBCwUA
A4IBAQCY8jdaQZChGsV2USggNiMOruYou6r4lK5IpDB/G/wkjUu0yKGX9rbxenDI
U5PMCCjjmCXPI6T53iHTfIUJrU6adTrCC2qJeHZERxhlbI1Bjjt/msv0tadQ1wUs
N+gDS63pYaACbvXy8MWy7Vu33PqUXHeeE6V/Uq2V8viTO96LXFvKWlJbYK8U90vv
o/ufQJVtMVT8QtPHRh8jrdkPSHCa2XV4cdFyQzR1bldZwgJcJmApzyMZFo6IQ6XU
5MsI+yMRQ+hDKXJioaldXgjUkK642M4UwtBV8ob2xJNDd2ZhwLnoQdeXeGADbkpy
rqXRfboQnoZsG4q5WTP468SQvvG5
-----END CERTIFICATE-----
)=====";
...
client.setCACert(SERVER_CERT_CA);
```

The above code snippet is used to secure an HTTPS connection on an ESP32 by setting the CA certificate. This defines a constant character array called SERVER_CERT_CA to store the CA certificate and instructs the compiler to store the certificate in the ESP32's program memory (flash memory) rather than dynamic memory (RAM). By using this syntax R"=====(...)=====", the certificate can be inserted as a raw string without having to escape special characters in the certificate. The client.setCACert() function call sets the CA certificate for the WiFiClientSecure client. The WiFiClientSecure library uses this certificate to verify the identity of the server during the SSL/TLS handshake process.

Note that CA certificates generally have an expiration date of no more than one year. If you plan to use ESP32 devices in the field for years, they may stop working when the server's CA certificate expires. Use the client.setInsecure() function to configure the client to ignore SSL/TLS certificate validation.

```
client.setInsecure();
```

When this method is called, the client no longer checks the authenticity of the server certificate. This allows the ESP32 to connect to a server without having to validate the server's certificate chain.

7.5 Getting Started with Google Firebase

In this section, you'll learn how to integrate Google Firebase with the ESP32 to build IoT applications. Google Firebase is a back-as-a-service that enables developers to build iOS, Android, and web applications. Real-time database is one of its services where developers store and synchronize data as JSON between users' devices in real-time [12]. Using the Firebase real-time database, we can remotely store, update and retrieve data from our IoT devices in real time, enabling integration and communication between devices and applications.

7.5.1 Setting Up Firebase Project

1. **Create a Firebase Project**
   Create your project on the Firebase Console as shown in **Figure 9**. You can log in using your existing Gmail account. Click on "**Create a project**", you can named your first Firebase project as "**ESP32 IOT Firebase DB**", and follow the prompts to create a new project.

Figure 9 *Google Firebase console page.*

2. **Add Firebase Realtime Database**
   Navigate to the "**Build**" section of the Firebase console and then click "**Realtime Database**" as shown in **Figure *10***. You will be prompted for the Firebase Real-Time Database console. Then click "**Create Database**" and follow the instructions to set up your database. You will be asked to select a server location and security rule of your Firebase database. Select the database server location closest to you, e.g., "**Singapore (asia-southeast1)**", and "**locked mode data**" as the default database security rule.

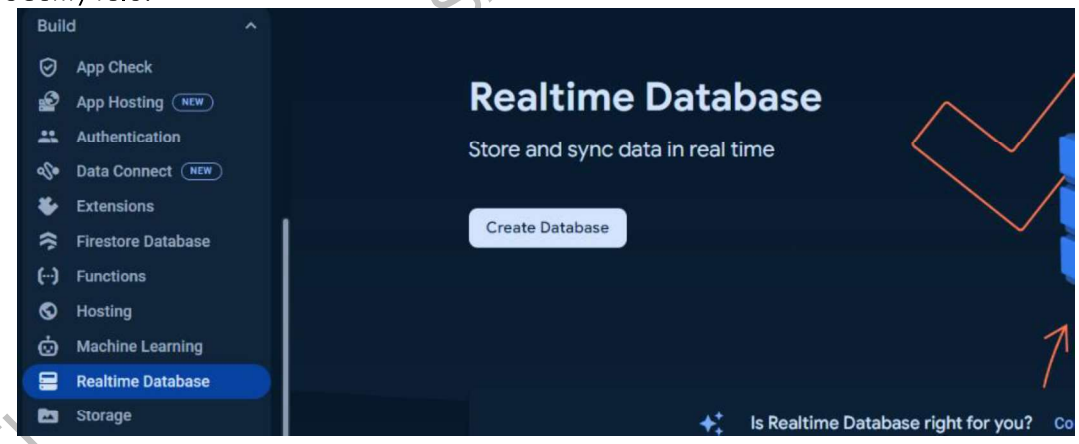


Figure 10 *Creating Firebase real-time database.*

3. **Set Database Rules**
   For development purposes, go to the "**Rules**" section of the real-time database and modify the default security rules to allow the client read and write access to the database as shown below. Click "**Publish**" to apply the rules. You can simulate the get, update, create, and delete request by clicking "**Rules Playground**" to check whether Firebase allows you to read and write data. Note that these rules are insecure for production environments and should be modified to suit your security needs.

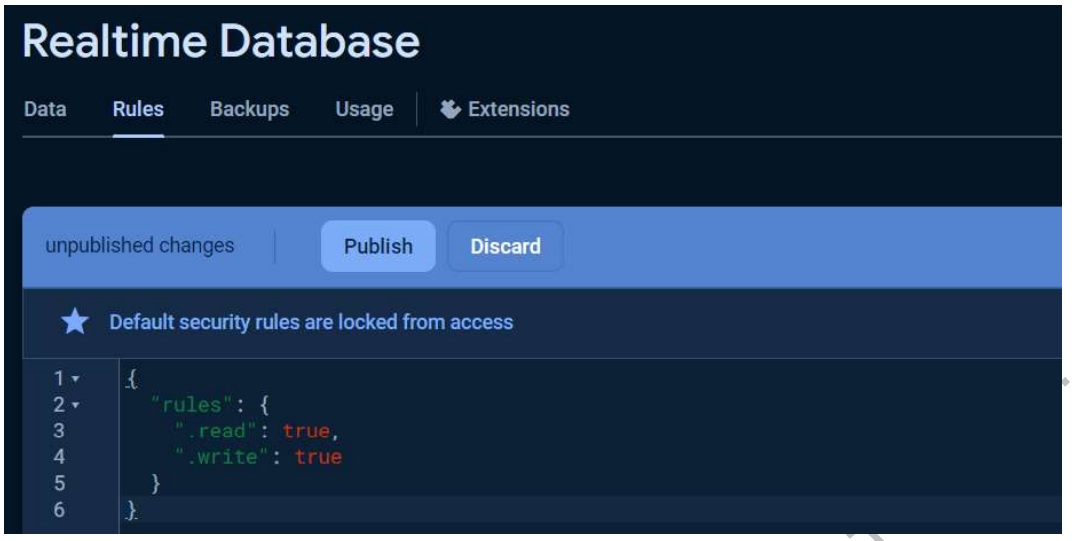


Figure 11 *Configurating Firebase database security rules.*

4. **Adding Data to Firebase Realtime Database**
   To add a data container to the Firebase real-time database, navigate to the Data tab of your database. Click the "**+**" icon to add a new data container. The "**Key**" and "**Value**" fields will appear. Enter a key name for your data node named "**Sensors**", as data node.

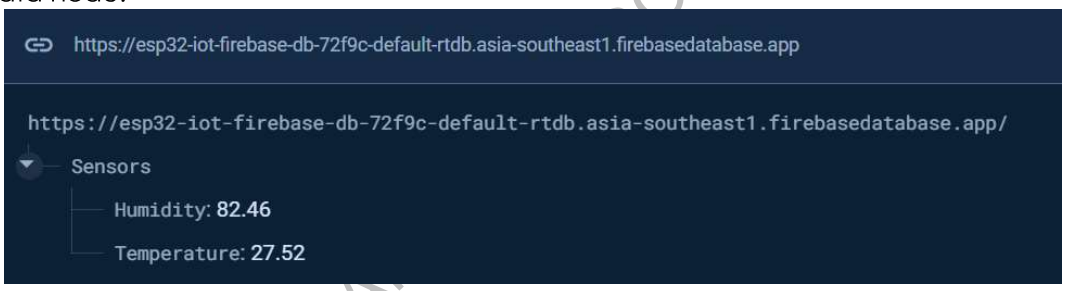


Figure 12 *Adding data container to the Firebased database.*

Click "**Add**" to create a hierarchical tree called "**Sensors**". Below "**Sensors**" data node, data variable will be added. Let's say "**Humidity**" and "**Temperature**" and then enter any initial value as shown in **Figure *12***. You can leave it as it is for now. Later it will be used to demonstrate accessing the database via an HTTP URL.

5. **Get the Database URL and Authentication Key**
   Navigate to "**Project Settings**" in the Firebase console as shown in **Figure *13***. Click "**Service Accounts**" > "**Database Secrets**" where you will find the authentication token used to access the data.

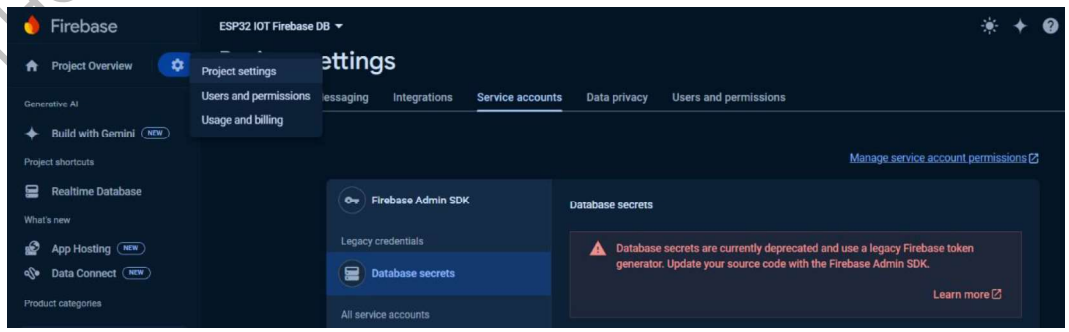


Figure 13 *Locating Firebase database secret key or authentication token.*

Under "**Database secrets**", click "**Show**" to view the database secret or authentication key as shown in **Figure *14***. Copy and paste the Firebase database URL and secret as follows.
[Database_URL].json?auth=[Database_Secret]
Note that in "**Data**" tab, where you can access the database URL, which is something like https://your-database-project.firebasedatabase.app/.

Figure 14 *Firebase database secret or authentication key manager window.*

If we have a reference URL for the Firebase database, such as https://esp32-iot-firebase-db-72f9c-default-rtdb.asia-southeast1.firebasedatabase.app/, and an authentication key, hIvRKyDJ9ypDdafJHFXXXXXXXv9Dbk0mzjaN3Z6M, the URL for Firebase database be present like this:
https://esp32-iot-firebase-db-72f9c-default-rtdb.asia-southeast1.firebasedatabase.app/.json?auth=hIvRKyDJ9ypDdafJHFXXXXXXXv9Dbk0mzjaN3Z6M
Copy the URL to your Web browser. You should access the database content shown below.

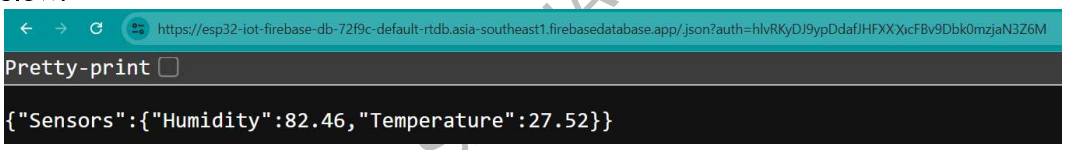


Figure 15 *Locating Firebase database secret key or authentication token.*

Using the Firebase URL scheme, the database content you created earlier should appear in your web browser. The sensor data, temperature and humidity are formatted in JavaScript Object Notation (JSON) format. In the next section, you'll learn how to create and parse JSON document.

7.5.2 Creating and Parsing JSON Document

All Firebase real-time database data is stored as JSON objects. JSON is a readable standard format for structuring data. It is primarily used to transfer data between a server and a web application as an alternative to XML [13], [14]. The two main parts of JSON are keys and values.

The data types that can be passed in the JSON data structure are array, boolean, number, object and string. Unlike a SQL database, there are no tables or records. When you add data to the JSON tree, it becomes a node in the existing JSON tree with an associated key. For example, imagine an environmental monitoring application that allows the user to save a device profile, timestamp, and environmental parameters. The data structure should look like as follows.

```
{
  "Device ID": "1089664092",
  "Device Name": "ESP-EnviSense_001",
  "Location": "Greenhouse",
  "Description": "Monitoring temperature and humidity",
  "Sensors": {
```

```
    "Timestamp": "2024-07-9 18:04:07",
    "Temperature": 27.81,
    "Humidity": 83
  }
}
```

The easiest way to create and parse JSON document with the Arduino is to use the ArduinoJson library. It is a simple and efficient JSON library for Arduino, and any embedded C++ project. First, we need to install Benoit Blanchon's ArduinoJson library as shown in **Figure 16** via the Arduino IDE library manager.

Figure 16 *Arduino library manager - ArduinoJson library by Benoit Blanchon.*

Using the ArduinoJson library, we can easily create and parse JSON document on ESP32. **Listing 2**, we will create a simple object JSON document that contains device meta data, timestamp, temperature, and humidity sensor values.

Listing 2 *Creating JSON documents using ArduinoJson library*

```
#include <ArduinoJson.h>

// Create JSON object named doc
JsonDocument doc;
```

First, we need to include the ArduinoJson library, which provides functions for creating, parsing and serializing JSON data. The JsonDocument is an object provided by the ArduinoJson library for storing JSON data, regardless of the underlying memory management strategy. In practice you should specify the size based on your JSON structure. You can use DynamicJsonDocument, which allocates memory dynamically on the heap, or StaticJsonDocument, which allocates memory statically on the stack [15]. For simplicity, this example does not include a capacity definition, but it is critical for practical use cases.

```
void setup() {
  Serial.begin(115200);
}

void loop() {
  // Create Key-Value pairs inside the JSON object
  doc["Device ID"] = "1089664092";
  doc["Device Name"] = "ESP-EnviSense_001";
  doc["Location"] = "Greenhouse";
  doc["Description"] = "Monitoring temperature and humidity";
  doc["Sensors"]["Timestamp"] = "2024-07-9 18:04:07";
  doc["Sensors"]["Temperature"] = 27.81;
  doc["Sensors"]["Humidity"] = 83;
```

This part of the code adds key-value pairs to the JsonDocument. The doc["Device ID"] = "1089664092"; and others directly add a key called Device ID with the value "1089664092" to the JSON document. The Sensor object contains nested key-value pairs for Timestamp, Temperature, and Humidity.

```
  // Generate simple JSON data structure
  String simpleJSON;
  serializeJson(doc, simpleJSON);
  Serial.println("Simple JSON Format: \n" + simpleJSON);
```

This part serializes a JSON document using the serializeJson() function. It converts the JSON document into a compact JSON string, then saves it to the simpleJSON string and prints it to the Serial Monitor. On the other hand, you can prettify a JSON document as follows.

```
  // Generate prettified JSON data structure
  String prettyJSON;
  serializeJsonPretty(doc, prettyJSON);
  Serial.println("Pretty JSON Format: \n" + prettyJSON);

  delay(5000);
}
```

The serializeJsonPretty() function converts the JSON document into a prettified JSON string with indentation and saves it in the prettyJSON string variable to print the prettified JSON document on the Serial monitor. The last line of code introduces a 5 second delay before the loop function repeats.

We print out the resulting JSON document in the Arduino Serial monitor. The result of the above code is shown below.

```
Simple JSON Format:
{"Device ID":"1089664092","Device Name":"ESP-
EnviSense_001","Location":"Greenhouse","Description":"Monitoring
temperature and humidity","Sensor":{"Timestamp":"2024-07-9
18:04:07","Temperature":27.81,"Humidity":83}}
Pretty JSON Format:
{
  "Device ID": "1089664092",
  "Device Name": "ESP-EnviSense_001",
  "Location": "Greenhouse",
  "Description": "Monitoring temperature and humidity",
  "Sensor": {
    "Timestamp": "2024-07-9 18:04:07",
    "Temperature": 27.81,
    "Humidity": 83
  }
}
```

Now you have learn the basics of how to create JSON document and using ArduinoJson library with ESP32. At this time, we will be going to parse JSON data. **Listing 3** focuses on extracting and printing specific values from a JSON string that represents the same JSON document we created.

Listing 3 *Parsing JSON documents using ArduinoJson library*

```
#include <ArduinoJson.h>

// Create JSON object named doc
JsonDocument doc;

void setup() {
  Serial.begin(115200);
}

void loop() {
```

```
  // Input JSON string
  const char* jsondoc = "{\"Device ID\":\"1089664092\",\"Device Name\":\"ESP-
EnviSense_001\",\"Location\":\"Greenhouse\",\"Description\":\"Monitoring
temperature and humidity\",\"Sensors\":{\"Timestamp\":\"2024-07-9
18:04:07\",\"Temperature\":27.81,\"Humidity\":83}}";
```

This defines a constant character pointer jsondoc that contains a JSON string. The JSON string represents the data of an environmental monitoring device that we created earlier.

```
  // Deserialize the JSON document
  DeserializationError error = deserializeJson(doc, jsondoc);
```

The deserializeJson() function parses the JSON string and stores the result in the doc object [16]. It returns a DeserializationError object to indicate whether the parsing was successful or not. It can have one of the following values:

- Ok: the deserialization was successful.
- EmptyInput: the input was empty or contained only spaces.
- IncompleteInput: the input was valid but ended prematurely.
- InvalidInput: the input was not a valid JSON document.
- NoMemory: the JsonDocument was too small.
- TooDeep: the input was valid, but it contained too many nesting levels.

```
  // Test if parsing succeeds
  if (error) {
    Serial.print(F("deserializeJson() failed: "));
    Serial.println(error.f_str());
    return;
  }
```

This part checks for deserialization errors. If deserializeJson() fails, the error message is printed on the Serial monitor and the function returns early to avoid further processing.

```
  // Extracting values
  unsigned long int deviceId = doc["Device ID"].as<unsigned long int>();
  auto deviceName = doc["Device Name"].as<const char*>();
  auto location = doc["Location"].as<const char*>();
  auto description = doc["Description"].as<const char*>();
  auto timeStamp = doc["Sensors"]["Timestamp"].as<const char*>();
  float temperature = doc["Sensors"]["Temperature"].as<float>();
  int humidity = doc["Sensors"]["Humidity"].as<int>();
```

These lines extract values from the parsed JSON document. The as<dataType>() method is used to cast the extracted values to their appropriate data types. We use the variable declaration with the keyword "auto", which makes it possible to infer the type of the variable from the type of the expression on the right.

```
  // Print values.
  Serial.printf("Device ID: %lu, Device Name: %s, Location: %s, Description:
%s, Timestamp: %s, Temperature: %.2f, Humidity: %d \n", deviceId, deviceName,
location, description, timeStamp, temperature, humidity);

  delay(5000);
}
```

The final part of the code is to print the extracted values using Serial.printf() for formatted output on the serial monitor. A 5 second delay is added before the loop repeats. This prevents the loop from running too fast and gives time to read the serial monitor's output.

7.5.3 ESP32 HTTP Client Communicate with Google Firebase

In this section, you'll learn how to store and retrieve data on Google Firebase using ESP32. With a real-time database in the cloud, various applications such as mobile and web applications can be integrated with the IoT devices. We will use Firebase HTTP API to connect hardware devices like ESP32 to Firebase real-time database. It allows the different systems to communicate via an HTTP request.

Using the following HTTP request, we can store and retrieve data in Firebase real-time database as shown below [17], [18].

TABLE 1
Example table

HTTP Request Method	Description
PUT	Write or replace data with a defined path.
PATCH	Update some of the keys for a defined path without replacing all of the data.
POST	Add to a list of data in the Firebase database. Every time you send a POST request, the Firebase client generates a unique key.
DELETE	Remove data from the specified Firebase database reference.
GET	Read data from the Firebase database.

In the next example in **Listing 4**, we will show how to save new data from ESP32 to Firebase real-time database and retrieve data from Firebase real-time database to ESP32 using HTTP request methods. In the previous sections, we created our first database and sensor data in a JSON structure. We will use them as a reference for storing and retrieving the data on ESP32. The example Arduino sketch is shown below.

Listing 4 *Updating and Retrieving data from Firebase Real-Time Database on ESP32*

```
#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <ArduinoJson.h>
#include <time.h>
#include <esp_sntp.h>
```

These libraries are necessary for Wi-Fi connectivity, secure HTTPS communication, JSON handling, and time synchronization.

```
// Create Wi-Fi client secure object for Firebase GET/PATCH HTTP request
WiFiClientSecure FirebaseGetRequest;
WiFiClientSecure FirebasePatchRequest;
```

These lines create two secure Wi-Fi client objects for making GET and PATCH HTTP requests to Firebase database. WiFiClientSecure ensures encrypted communication over HTTPS.

```
// Create JSON object named doc
JsonDocument doc;
```

This line creates a JSON document object named doc which will be used to handle JSON data as Firebase payload during PATCH operations.

```
// Wi-Fi credentials
const char* SSID = "REPLACE_WITH_WIFI_SSID";
```

```
const char* PASSWORD = "REPLACE_WITH_WIFI_PASSWORD";

// Firebase Database Credentials
const int SSL_PORT = 443;
const char* FIREBASE_HOST = "<your-database-project-name>.<database-
server>.firebasedatabase.app";
const String FIREBASE_AUTH = "<your-firebase-database-secret-key>";
const String FIREBASE_PATH = "/";
```

The following lines are the Firebase database credentials need to be define. The SSL_PORT is set to 443, which is the standard port for HTTPS communication. The FIREBASE_HOST is the Firebase database host URL. Replace <your-database-project-name> and <database-server> with your Firebase project's specific details that can be found the the database data tab. The FIREBASE_AUTH is the Firebase database secret key. This key is required to authenticate requests to the Firebase database. The FIREBASE_PATH specifies the root path of your Firebase database, for now a single slash is used for demonstration, you may also play around to specify your database path to be access.

```
// Define the NTP servers and parameters settings
#define NTP_SERVER_1 "pool.ntp.org"
#define NTP_SERVER_2 "asia.pool.ntp.org"
#define TIME_ZONE "PHT"
#define GMT_OFFSET_S 28800ULL
#define DAYLIGHT_OFFSET_S 0ULL

// Define variable that store local time info
struct tm timeinfo;
```

To ensure that the ESP32 has accurate time for timestamping, it's essential to synchronize it with Network Time Protocol (NTP) servers. The following line of codes sets up the NTP server addresses, time zone, and offsets. It also defines a structure to store the local time information. The TIME_ZONE is defined as "PHT", which stands for Philippine Time. The GMT_OFFSET_S is the offset from Greenwich Mean Time (GMT) in seconds. Philippine Time is GMT+8, which translates to 28800 seconds (8 hours * 3600 seconds/hour). The DAYLIGHT_OFFSET_S is set to 0ULL as there is no daylight saving time adjustment in the Philippines.

```
void setup() {
  Serial.begin(115200);

  // Initialize network time protocol (NTP) configurations
  sntp_set_time_sync_notification_cb(timeavailable);
  sntp_servermode_dhcp(1);
  configTime(GMT_OFFSET_S, DAYLIGHT_OFFSET_S, NTP_SERVER_1, NTP_SERVER_2);
```

The following line of code snippet is the initialization of NTP configurations in the setup() function. The sntp_set_time_sync_notification_cb(timeavailable) registers a callback function timeavailable that will be called whenever the time is successfully synchronized with the NTP server. This is useful for executing specific actions such as printing debug message when the time synchronization occurs. The sntp_servermode_dhcp(1) configures the Simple Network Time Protocol (SNTP) client to use the DHCP-provided NTP servers. When set to 1, the NTP servers specified by the DHCP server will be used for synchronization. In this setup, the ESP32 connects to Wi-Fi and synchronizes its clock with NTP servers using the configTime function.

```
  // Set ESP32 WiFi to station mode
  WiFi.mode(WIFI_STA);

  // Initiate the Wi-Fi connection
```

```
  WiFi.begin(SSID, PASSWORD);

  Serial.print("Connecting...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(100);
  }

  // Print some information of the Wi-Fi network
  Serial.print("\nConnected to the ");
  Serial.println((String)WiFi.SSID() + " network");
  Serial.print("[*] RSSI: ");
  Serial.println((String)WiFi.RSSI() + " dB");
  Serial.print("[*] ESP32 IP: ");
  Serial.println(WiFi.localIP());
  Serial.println();

  // Allows for 'insecure' connections. It turns off CA certificate checking
  FirebaseGetRequest.setInsecure();
  FirebasePatchRequest.setInsecure();

  // Initialize HTTP GET request once
  FirebaseGetRequestBegin(FIREBASE_PATH);
```

The following line of code is responsible for connecting the ESP32 to a Wi-Fi network and preparing the device to communicate with the Firebase database over secure HTTPS connections. It configures the FirebaseGetRequest and FirebasePatchRequest objects to allow insecure connections by turning off CA (Certificate Authority) certificate checking. This means the ESP32 will not validate the server's SSL/TLS certificate. The FirebaseGetRequest function prepares for an HTTP GET request to the specified Firebase database path (FIREBASE_PATH).

```
  // Loop until the current local time is obtained
  Serial.print("[NTP] Waiting for time synchronization...");
  while (!getLocalTime(&timeinfo)) {
    Serial.print(".");
  }
}
```

The last part of the setup function is responsible for ensuring that the ESP32 obtains the current local time from an NTP server before proceeding with loop function. The getLocalTime(&timeinfo) is a function that attempts to get the current time and store it in the timeinfo structure, where struct tm is a standard C library structure that holds date and time information. The code waits for the time to be set before proceeding with further operations.

```
void loop() {
  // Get the payload form Firebase Database
  FirebaseGetPayload();
```

This part of the code involves the loop() function, which continuously runs and handles the fetching of data from the Firebase database. The primary focus here is on obtaining the payload from the Firebase database using the FirebaseGetPayload() function. It keeps the ESP32 updated with the most recent information stored in Firebase.

```
  // Create Key-Value pairs inside the JSON object
  doc["Sensors"]["Timestamp"] = getLocalTime();
  doc["Sensors"]["Temperature"] = random(2000, 3000) / 100.00;
  doc["Sensors"]["Humidity"] = random(70, 100);

  // Generate simple JSON data structure
```

```
  String firebasePayload;
  serializeJson(doc, firebasePayload);
```

This code create a JSON object, creating key-value pairs in a JSON object representing sensor data, serialize it to a JSON string as payload to the Firebase database.

```
  // Updata data to Firebase Database
  FirebaseUpdateDB(firebasePayload);
}
```

The last part of the loop function, where it updates data to the Firebase Database using a custom function FirebaseUpdateDB. The firebasePayload variable contains the serialized JSON data that created in the previous lines.

```
void FirebaseGetRequestBegin(String url_path) {
  // Create TLS (Transport Layer Security) connections
  if (!FirebaseGetRequest.connect(FIREBASE_HOST, SSL_PORT)) {
    return;
  }

  // Create HTTPS URL
  String HTTP_PARAM_URL = url_path + ".json?auth=" + FIREBASE_AUTH;

  // Send HTTP request header
  FirebaseGetRequest.println("GET " + HTTP_PARAM_URL + " HTTP/1.1");
  FirebaseGetRequest.println("Host: " + String(FIREBASE_HOST));
  FirebaseGetRequest.println("Accept: text/event-stream");
  FirebaseGetRequest.println("Connection: close");
  FirebaseGetRequest.println();
}
```

The FirebaseGetRequestBegin function initiates a secure HTTPS GET request to a specified path in the Firebase Realtime Database. It establishes a TLS connection using the WiFiClientSecure object, constructs the full URL with the required authentication token, and sends the appropriate HTTP headers. This function is essential for retrieving data securely from Firebase, ensuring that the ESP32 can communicate with the Firebase Realtime Database over an encrypted connection.

```
String FirebaseGetPayload() {
  String response;
  // Read response from server and print them to Serial monitor
  while (FirebaseGetRequest.available()) {
    response = FirebaseGetRequest.readStringUntil('\r');
    Serial.print(response);
  }
  return response;
}
```

The FirebaseGetPayload function reads the response from the Firebase server after an HTTP GET request. It uses a while loop to read the response line by line until no more data is available. The response is stored in a String object and printed to the Serial Monitor for debugging purposes. Finally, the function returns the response string. This function is essential for extracting and utilizing the data retrieved from the Firebase Realtime Database, allowing the ESP32 to process and display the information received from the server.

```
void FirebaseUpdateDB(String body_content) {
  // Create TLS (Transport Layer Security) connections
  if (!FirebasePatchRequest.connect(FIREBASE_HOST, SSL_PORT)) {
    return;
```

```
  }

  // Create HTTPS URL
  String HTTP_PARAM_URL = FIREBASE_PATH + ".json?auth=" + FIREBASE_AUTH;

  // Create HTTP body in JSON (JavaScript Object Notation) format
  String HTTP_BODY = body_content;

  // Send HTTP request header
  FirebasePatchRequest.println("PATCH " + HTTP_PARAM_URL + " HTTP/1.1");
  FirebasePatchRequest.println("Host: " + String(FIREBASE_HOST));
  FirebasePatchRequest.println("Content-Type: application/json");
  FirebasePatchRequest.println("Content-Length: " + String(HTTP_BODY.length()));
  FirebasePatchRequest.println("Connection: close");
  FirebasePatchRequest.println();
  FirebasePatchRequest.println(HTTP_BODY);

  // Disconnect to Firebase before to reconnect it for new request
  FirebasePatchRequest.stop();
}
```

The FirebaseUpdateDB function updates the Firebase Realtime Database by sending an HTTP PATCH request with JSON data. It establishes a secure TLS connection, constructs the HTTPS URL, creates the HTTP body with the JSON data, and sends the HTTP request header and body. After sending the request, it disconnects from the Firebase server. This function is essential for ensuring that the ESP32 can update the Firebase database with new sensor data or other information.

```
String getLocalTime() {
  // Format timestamp as yyyy-MM-dd HH:mm:ss
  String timestamp;
  if (getLocalTime(&timeinfo)) {
    timestamp = String(timeinfo.tm_year + 1900) + "-" + String(timeinfo.tm_mon +
1) + "-" + String(timeinfo.tm_mday) + "  " + String(timeinfo.tm_hour) + ":" +
String(timeinfo.tm_min) + ":" + String(timeinfo.tm_sec);
  }
  return timestamp;
}

void timeavailable(struct timeval* t) {
  // Print debug time synchronization message to Serial Monitor
  Serial.println("got time adjustment from NTP!");
  getLocalTime();
}
```

The last part of the code provides functionality for obtaining and formatting the current local time and handling notifications when time synchronization from an NTP server occurs. The getLocalTime function provides a formatted timestamp of the current local time, useful for logging and data synchronization. The timeavailable function handles notifications from the NTP server, ensuring the system's time is accurate and up-to-date. These functions are essential for applications that require precise timing, such as logging sensor data or synchronizing with remote servers.

Open your Arduino Serial monitor, to see the printed Firebase database server response as shown below.

```
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 17 Jul 2024 09:07:27 GMT
```

```
Content-Type: text/event-stream; charset=utf-8
Connection: close
Cache-Control: no-cache
Access-Control-Allow-Origin: *
Strict-Transport-Security: max-age=31556926; includeSubDomains; preload

event: put
data:
{"path":"/","data":{"Sensors":{"Humidity":95,"Temperature":29.77,"Timestamp
":"2024-7-17  17:0:17"}}}

event: patch
data:
{"path":"/","data":{"Sensors":{"Humidity":95,"Temperature":24.69,"Timestamp
":"2024-7-17  17:7:47"}}}
```

7.6 Build a Mobile App with Thunkable

Thunkable is a block-based coding and drag and drop User Interface (UI) development platform that allows you to create mobile applications for iOS and Android without needing to write code [19]. It provides a visual interface where you can drag and drop components to design app's UI and use block-based programming to add functionality. This method is like platforms like Scratch and MIT App Inventor, aiming to simplify the development process often used for education, prototyping, and personal projects.

7.6.1 Getting Started with Thunkable

In the rapidly evolving world of the IoT, the ability to quickly develop and deploy applications that can control and monitor devices is crucial. Thunkable, offers an intuitive way for both beginners and experienced developers to create powerful mobile applications. This section will guide you through the basics of getting started with Thunkable, with a focus on creating applications for IoT projects. Just follow the following steps.

1. **Setting Up Your Thunkable Account**
   You can sign up with a Google account or an Apple ID and sign in immediately, or you can enter your email address as shown in **Figure 17**. It recommends using a Google or Apple account, which will make testing your app easier. To get started with Thunkable, visit the sign up (https://x.thunkable.com/signup) page [20].

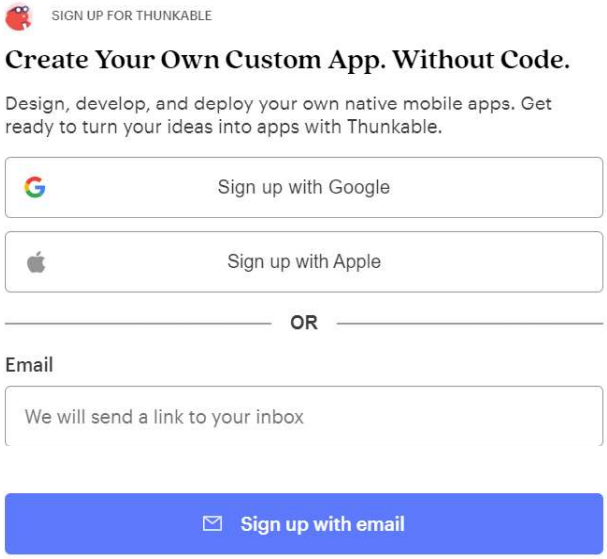


Figure 17 *Creating a Thunkable account via email.*

2. **Creating a New Project**

Once logged in, click on "**Create New Project**" to start a new project. You can choose to start from scratch or use one of the pre-built templates as shown in **Figure *18***. In this case, you will be able to create a mobile application from scratch to start with [21].

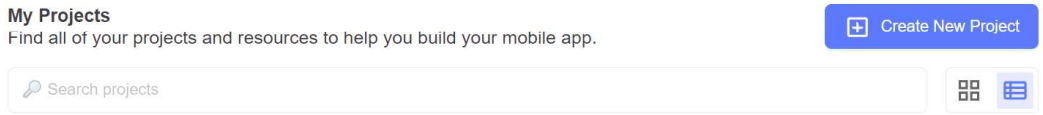


Figure 18 *Thunkable project list.*

Name your first project, anything you want, and you will see the of your Thunkable project as depicted in **Figure *19***. In this example, you will see a personal project named "Firebase Test" from the workspace where it is stored.

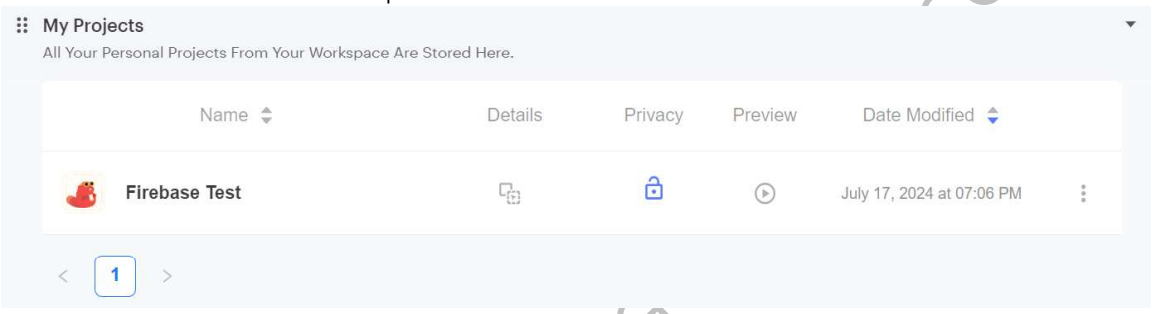


Figure 19 *Thunkable project list view stored in the workspace.*

3. **Understanding the Thunkable Interface**

Open your first Thunkable project, the workspace should be as shown **Figure *20***. The Thunkable interface consists of three main areas.

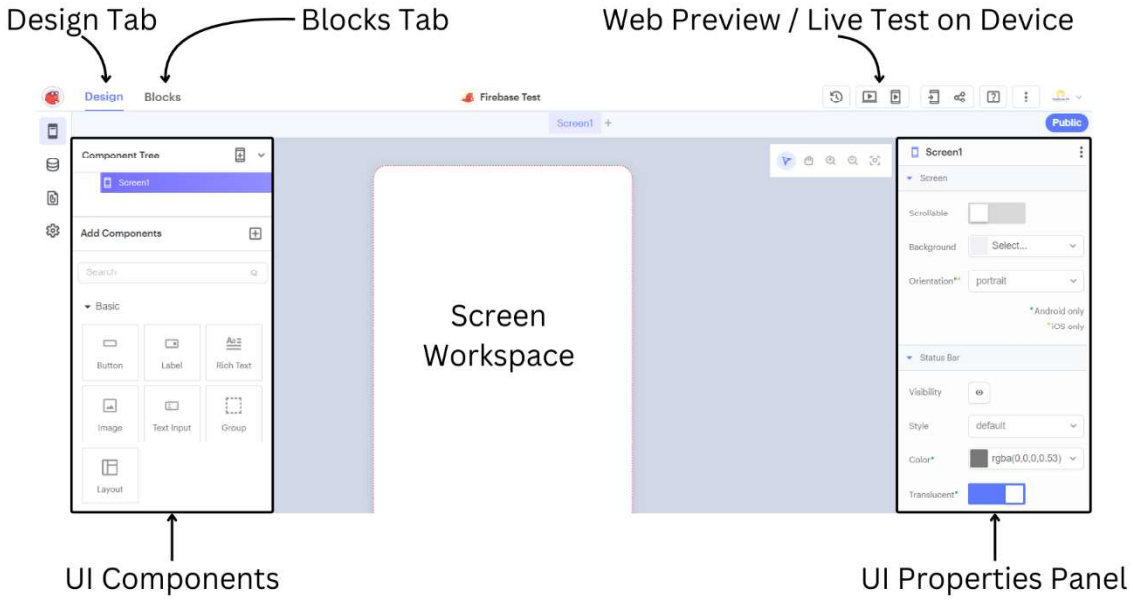


Figure 20 *Thungkable workspace interface.*

- **Design Tab**: This is where you design the user interface (UI) of your app. You can drag and drop UI components such as buttons, labels, text inputs, and more onto the screen workspace. Each UI component you add to your app has an associated properties panel unique to that component. Using the various settings in the properties panel, you can style and customize your Thunkable UI components [22].

- Blocks Tab: This is where you add functionality to your app using block-based programming. Each block represents a piece of code that performs a specific action. Every UI component has its own set of blocks to start or trigger an event and set and change properties [22].
- Web Preview / Live Test: The Thunkable Live app on your mobile device allows you to test your Thunkable application on your device in real-time or test your app directly on your web browser. These tools aren't designed to check for errors but to allow you to preview your app's design and test its functionality in real time [22].

7.6.2 Creating Your First IoT App

Thunkable provides built-in support for integrating Firebase Real-time Database, you can quickly build and deploy IoT apps that monitor and control devices, process sensor data, and provide real-time feedback, all with minimal coding effort.

1. **Connect Your Firebase Account to Thunkable**
   To connect your Firebase and Thunkable accounts, you need to locate two Firebase properties, the database secret key / API key, and database URL. We already have this in the previous section, copy the API key and database URL [23]. In your Thunkable tab, click the Settings gear icon in the sidebar as shown in **Figure 21**. Scroll to the Firebase Settings section and paste the API key and Database URL into the fields respectively as shown below.

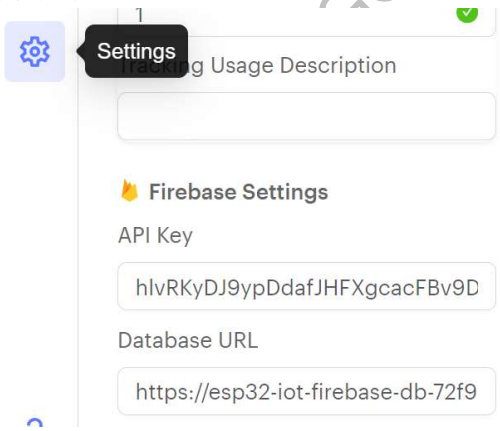


Figure 21 *Firebase database integration via Thunkable API settings.*

2. **Add UI Components**
   Drag and drop the following components onto your screen as shown below.
   - Labels x4: For add description about the value and display the generated value and retrieved temperature value from database.
   - Buttons x3: To trigger generate temperature value, update value to Firebase database, and read database temperature value.

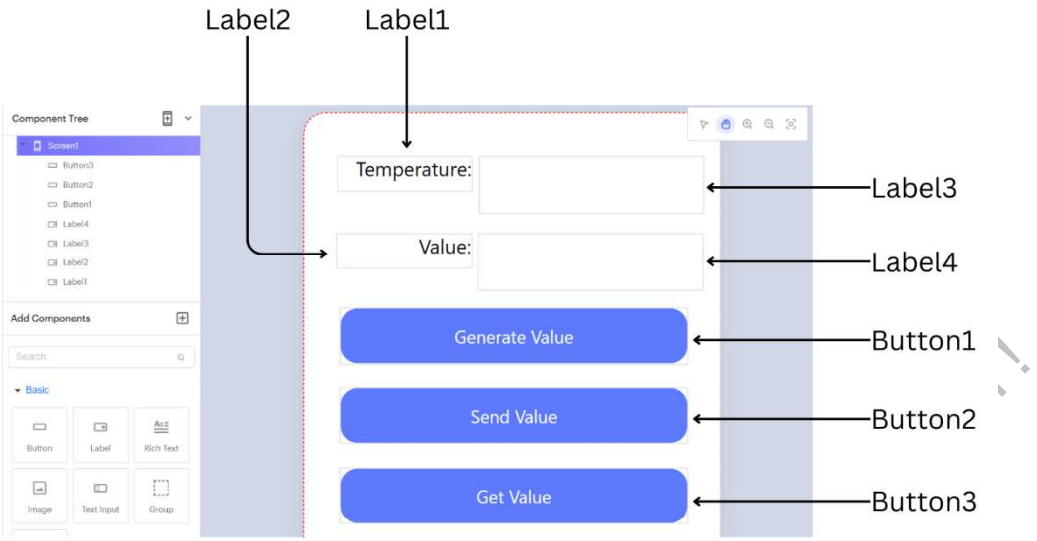
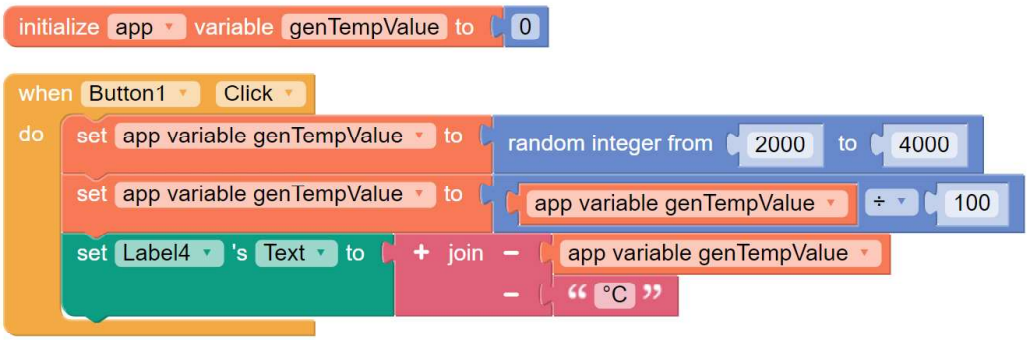


Figure 22 *Designed Thunkable mobile application user interface.*

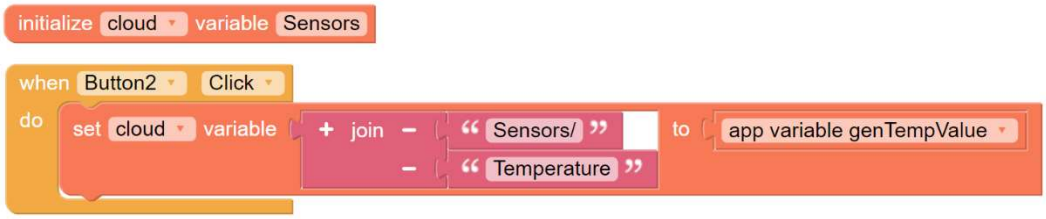
The app aims to update and read temperature data in real-time to demonstrate the basics of how to read and update data in a Firebase real-time database as shown in **Figure 22**. After placing the labels and buttons UI components in the screen, you may change its UI properties such as text field and dimensions [22]. Let's rename the Label1 as "Temperature:" and Label2 as "Value:" then set their dimensions to 120x30 (WxH). Delete the text field of Label3 and Label4, then set their dimensions to 200x50 (WxH). Rename the Button1, Button2, and Button3 text field to "Generate Value", "Send Value", and "Get Value" respectively to define the distinctive function of each button. Arrange the components as shown in the image above, after that you are ready to proceed with Blocks tab.

3. **Adding Functions with Blocks**

Go to the Blocks tab to create a function of the app. Let's begin with generating random temperature value function [24]. When the Button1 is clicked, it will generate a random value range from 20 to less than 40 degrees Celsius and set the Label4 text field into it to display the generated value. The Blocks code is shown below. The Blocks functions used can be found from Variables, Objects, Math, Text, and UI components.



The connection between Thunkable app and Firebase database takes place with cloud variable. We need to define a cloud variable "Sensors" a data node of the database that contains timestamp, temperature and humidity data. Then create a function to update temperature data in Firebase database when a Button2 is clicked. The Blocks code is shown below.



Create another function that reads data from Firebase database when the button3 is clicked. The retrieved data will be displayed in the Label3 text field. The Blocks code is shown below.

4. Testing the App

Use Thunkable's Web Preview feature to test your app directly on your web browser. Click the "Generate Value" button to generate a random temperature value displayed in Label4. Then click the "Send Value" button to update the temperature data to the Firebase real-time database [25]. Open your database tab, you should see that the temperature field value is updated with the generated one. Then click the "Get Value" button to read the data from the temperature field value and display it to Label3. The app test preview is shown below.

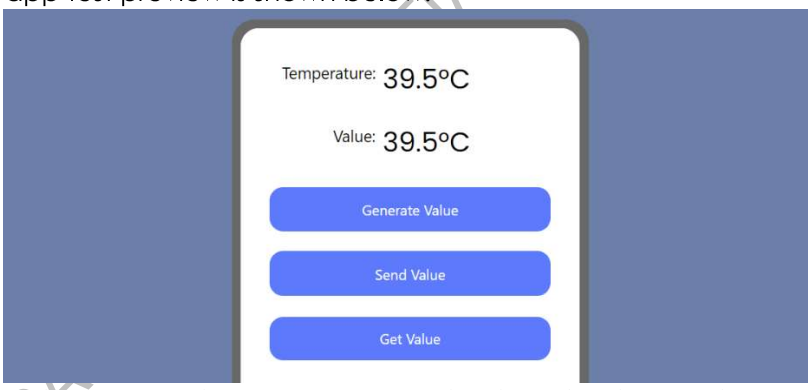


Figure 23 *Preview and testing Thunkable application with Firebase integration.*

By following these steps, you have successfully created an IoT application using Thunkable, which is able to communicate with the Firebase real-time database. This application can read and update data in real time, making it an excellent tool for monitoring and controlling devices connected to the Internet. The intuitive interface of Thunkable and the backends integration of Firebase make it an excellent combination for developing IoT applications without extensive programming knowledge.

7.7 Mini-Project 6: Real-Time Remote Monitoring and Control System with ESP32, Firebase Database, and Thunkable

7.7.1 Introduction

Real-time remote monitoring and control systems are key developments in IoT applications, enabling wireless interaction between physical devices and digital interfaces for increased efficiency and responsiveness in environmental management, industrial automation and smart home solutions. This mini project uses the ESP32 microcontroller as a wireless sensor node, equipped with environmental sensors such as temperature, humidity, soil moisture and light to capture real-time data from the environment. Complementing these inputs are physical buttons to facilitate manual control of the actuators state and LED lights acting as actuators for output control. The sensor node connected to Wi-Fi for network communication. ESP32 periodically pushes sensor data to the real-time Google Firebase database. The custom mobile application developed on the Thunkable platform interconnects to this database via its API and allows real-time access to sensor data and visualization, as well as remote updating of actuator status, such as switching LED's, which provides a bidirectional control that overrides hardware inputs.

Through implementation of this project, you will gain practical experience in integrating cloud-based database, secure HTTP or HTTPS protocol, JSON data handling, and drag and drop code mobile app development. It brings further comprehensive understanding of end-to-end IoT ecosystems that are scalable and innovative.

Upon completing this project, you will be able to:

- Program the ESP32 to establish secure Wi-Fi connectivity and perform periodic data updates to a Google Firebase real-time database, utilizing HTTP/HTTPS protocols and JSON data handling.
- Configure and manage a real-time Firebase database to store, update, and retrieve sensor data and actuator status, including the implementation of database authentication and database rules.
- Develop responsive mobile applications using the drag and drop programming platform with Thunkable to interface with Firebase database, to facilitate real-time visualization of sensor data and remote control of actuators.
- Design a user-interface, build event-driven programming and two-way data flow in the mobile IoT ecosystem.
- Integrate hardware and software components for bidirectional control, where mobile app commands can override or complement physical button inputs to actuators.

7.7.2 Components Needed

Hardware Components:

1. LILYGO TTGO LoRa32 OLED ESP32 Development Board
2. Digital Temp & Humidity Sensor (AM2302/DHT22)
3. Capacitive Soil Moisture Sensor
4. Light Intensity Sensor (Photoresistor (LDR) GL5528)
5. OLED Display (SSD1306 128x64 pixels)
6. LED Indicator
7. Jumper Wires
8. Power Source (Battery or USB power)

Software Tools:

1. Arduino IDE
2. Google Firebase

3. Thunkable

7.7.3 System Architecture

This mini project demonstrates a 4-layer IoT architecture as depicted in **Figure 24**. The sensors collect real-time environmental data, which is processed by the ESP32 microcontroller. The sensor node also incorporates LEDs as actuators to represent controlled outputs such as fan ventilation, irrigation pump, or lighting systems and physical push buttons for manual control of actuator states. The hardware layer is responsible for data acquisition, local processing, and actuator control, forming the foundation of the IoT system.

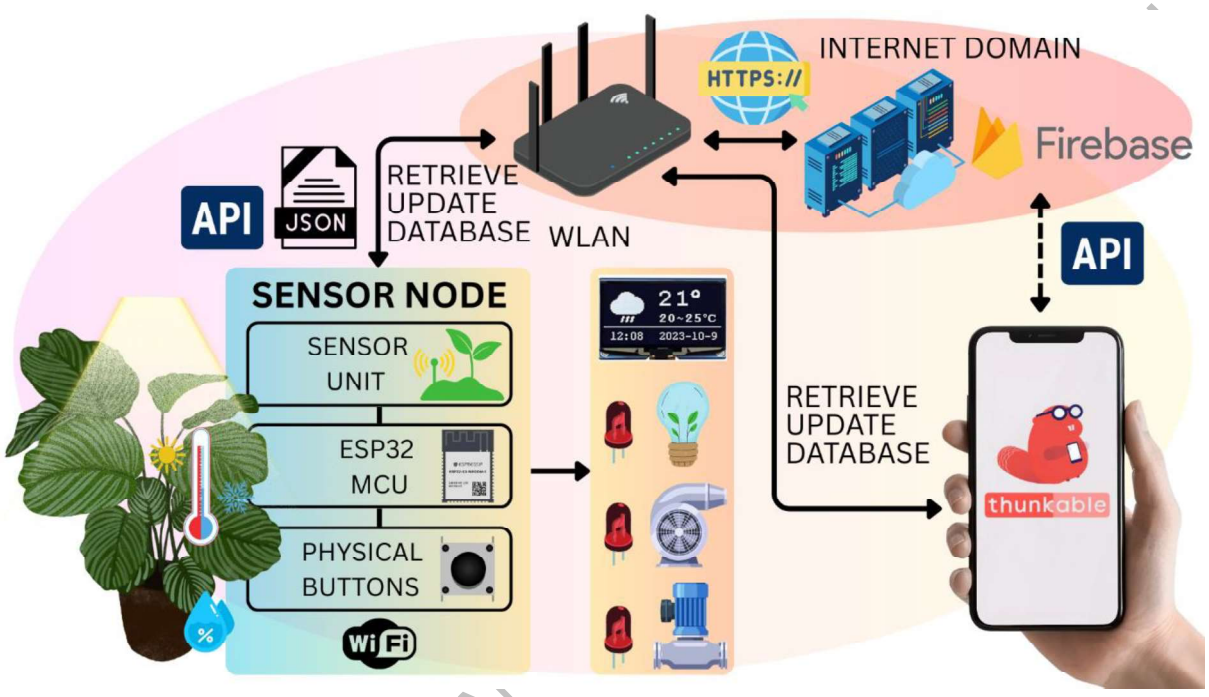


Figure 24 *Preview and testing Thunkable application with Firebase integration.*

In the network layer, ESP32 uses its built-in Wi-Fi module to establish connectivity with the internet, using the WiFi.h and WiFiClientSecure libraries to enable secure HTTPS communication. It periodically collects sensor data, formats it as JSON, and sends it to Firebase using secure HTTPS requests. Simultaneously, it polls the Firebase database for actuator state updates, adjusting LED states accordingly. The Thunkable app retrieves sensor data from Firebase for display and sends actuator commands to the database when user inputs are triggered. This bidirectional communication ensures that both hardware buttons and remote via the app control are synchronized, with Firebase acting as the central hub for data exchange.

The cloud layer utilizes Google Firebase's real-time database as the central data management. Data formatted as JSON objects transmitted to Firebase via HTTPS PATCH requests. The Firebase database is configured with a hierarchical structure, where a "Sensors" node stores environmental data and an "Actuators" node manages LED states. Authentication is handled using a database secret key, and security rules are set to allow read/write access during development, with recommendations for stricter rules in production. The Firebase HTTP API facilitates low-latency, bidirectional communication, enabling real-time updates between the ESP32 and the mobile application.

The application layer consists of a mobile application developed using the Thunkable no-code platform, designed to provide users with a responsive interface for monitoring and controlling the IoT device. The Thunkable app connects to the Firebase real-time database via its API, retrieving sensor data for real-time visualization on UI components such as labels. You can view environmental parameters displayed in an app widget. The app also includes buttons to send actuator commands to toggle LEDs, which are transmitted as JSON payloads

to update the actuators state in Firebase. These updates are then fetched by the ESP32 through periodic HTTPS GET requests, enabling remote control that can override or complement local button inputs. The Thunkable interface supports real-time testing via web preview or mobile device, ensuring immediate feedback on system functionality. The application layer thus bridges the user with the IoT system, providing an intuitive means to monitor and manage remote devices.

7.7.4 Project Plan

You must complete the following tasks to meet the project objectives.

- Develop an Arduino sketch to allow ESP32 to connect to the Wi-Fi network and synchronize time with the NTP servers for accurate timestamps. Use WiFiClientSecure to implement HTTPS communication to securely send sensor data such as temperature, humidity, soil moisture, light intensity, timestamp and actuator state as JSON payloads to Firebase real-time database via PATCH requests. Program ESP32 to periodically retrieve the actuator status (on and off) from Firebase using a GET request to ensure that remote commands are synchronized.
- Create a Firebase project in the Firebase console and create a real-time database with a hierarchical structure, including a Sensor node for environmental data and an Activator node for LED status. Get the database URL and secret key from the project settings in the Firebase Console and set the security rules to allow read and write access for development purposes. Test data storage and retrieval by manually adding sensor sample data and actuator status to the Firebase Data tab and then accessing the database URL via a web browser using the validated database URL.
- Design a mobile application for real-time monitoring and control of the project. Add the UI components, including labels for sensor descriptions and values, and buttons for sending commands to actuators, to the Design tab. In the Blocks tab, implement the functionality to connect to a Firebase database using an API key and database URL, to retrieve sensor data for display, and to send updates about the status of the actuators to Firebase. To ensure real-time visualization and control of your data, test your application using the Thunkable Web Preview or Live Test features.
- To process JSON data from an HTTP response in Firebase, the HTTP response must be broken up into substrings that contain only the JSON document. In this way, you can now deserialize JSON documents for data retrieval.
- Combine hardware, cloud, and application layers to create an IoT system. Flash the Arduino sketch to ESP32 to enable the capture of sensor data and communication with Firebase. Use the Thunkable application to monitor sensor data and control the LED's and verify that remote commands override or supplement the hardware input of the buttons.

7.7.5 Guidelines and Tips

- Consider using Figma (https://www.figma.com/), a collaborative design tool, for prototyping the user interface. Create a wireframe or mockup to plan the layout of the label, buttons, and display to ensure a pure and intuitive user experience. For example, design a user interface in the dashboard style with clearly identified sections for temperature, humidity, soil moisture and light intensity, together with separate buttons for the actuators control. You can export Figma designs and import them into the Thunkable Design tab to reduce the time to iteration and to align with user-centered design principles [26].
- To avoid ESP32 not responding due to a blocking delay() function, use non-blocking timing function using millis() to manage the sampling of sensors and the Firebase communication interval. For example, instead of using delay(10000) for waiting

between HTTP requests, implement a timer based on millis() to check the time elapsed, allowing ESP32 to perform other tasks at the same time [27]. This approach ensures smooth operation, particularly when processing multiple sensors or dealing with updates of actuators and is in line with the real-time requirements of the embedded system.

- Familiarize yourself with the Firebase HTTP response format, which includes HTTP headers, for example, "HTTP/1.1 200 OK, Content-Type: text/event-stream", followed by event data such as "event: put, data: {...}". The JSON payload is inserted in the data field and contains key-value pairs with nested objects, such as path and data. Recognize that JSON data may differ according to the data structure.
- After sending an HTTP GET request to Firebase using WiFiClientSecure, read the response streamline by line using client.readStringUntil('\n') or client.read() until the "data:" line is encountered. For example, you can break the header and event data lines and then extract the content of the JSON payload to a substring such as {path":"/","data":{"key":value}}. This way you can now deserialize the JSON payload and parse nested key-value pairs.
- You can add additional functionality to the mobile application by including alerts and notifications when specific conditions are met, such as temperature above the threshold or water level indicating the need for irrigation [28]. This feature increases user involvement and allows for proactive monitoring, making the system suitable for critical applications in the Internet of Things such as smart agriculture and environmental control.

References

[1] R. Mustapha, "What is HTTP? Protocol Overview for Beginners," freeCodeCamp. Accessed: Jul. 02, 2024. [Online]. Available: https://www.freecodecamp.org/news/what-is-http/

[2] Cloudflare, "What is HTTP?," Cloudflare, Inc. . Accessed: Jul. 02, 2024. [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

[3] Mozilla, "What is a URL? - Learn Web Development | MDN," mdn web docs. Accessed: Jul. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_URL

[4] IBM, "The components of a URL - IBM Documentation," IBM Corporation. Accessed: Jul. 02, 2024. [Online]. Available: https://www.ibm.com/docs/en/cics-ts/6.x?topic=concepts-components-url

[5] Mozilla, "HTTP request methods - HTTP | MDN." Accessed: Jul. 02, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[6] Postman, "Download Postman | Get Started for Free." Accessed: Aug. 15, 2025. [Online]. Available: https://www.postman.com/downloads/

[7] JSONPlaceholder, "JSONPlaceholder - Free Fake REST API." Accessed: Aug. 15, 2025. [Online]. Available: https://jsonplaceholder.typicode.com/

[8] Mozilla, "HTTP response status codes - HTTP | MDN." Accessed: Jul. 03, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[9] Espressif, "Wi-Fi API - Arduino ESP32 Latest Documentation," Espressif Systems (Shanghai) Co., Ltd. Accessed: Aug. 15, 2025. [Online]. Available: https://docs.espressif.com/projects/arduino-esp32/en/latest/api/wifi.html

[10] Cloudflare, "What is Transport Layer Security (TLS)? | Cloudflare," Cloudflare, Inc. Accessed: Jul. 03, 2024. [Online]. Available: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

[11] Y. Sanghvi, "Migrating any HTTP request to HTTPS on ESP32 | by Yash Sanghvi | Medium," Medium. Accessed: Jul. 03, 2024. [Online]. Available: https://medium.com/@sanghviyash6/migrating-any-http-request-to-https-on-esp32-5545a6de7845

[12] Firebase, "Firebase Realtime Database." Accessed: Jul. 04, 2024. [Online]. Available: https://firebase.google.com/docs/database
[13] Mozilla, "Working with JSON - Learn web development | MDN." Accessed: Jul. 09, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON
[14] Firebase, "Structure Your Database | Firebase Realtime Database." Accessed: Jul. 09, 2024. [Online]. Available: https://firebase.google.com/docs/database/web/structure-data
[15] ArduinoJson, "Serialization tutorial | ArduinoJson 7." Accessed: Aug. 16, 2025. [Online]. Available: https://arduinojson.org/v7/tutorial/serialization/
[16] ArduinoJson, "Deserialization tutorial | ArduinoJson 7." Accessed: Aug. 16, 2025. [Online]. Available: https://arduinojson.org/v7/tutorial/deserialization/
[17] Firebase, "Saving Data | Firebase Realtime Database." Accessed: Jul. 07, 2024. [Online]. Available: https://firebase.google.com/docs/database/rest/save-data
[18] Firebase, "Retrieving Data | Firebase Realtime Database." Accessed: Jul. 07, 2024. [Online]. Available: https://firebase.google.com/docs/database/rest/retrieve-data
[19] Thunkable, "What is Thunkable? | Thunkable Docs." Accessed: Jul. 17, 2024. [Online]. Available: https://docs.thunkable.com/
[20] Thunkable, "Thunkable Account | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/getting-started/get-started/signing-in
[21] Thunkable, "Getting Started Guide | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/getting-started/get-started
[22] Thunkable, "User Interface (UI) Components | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/app-design/ui-components
[23] Thunkable, "Connect a Firebase Realtime Database | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/blocks/blocks/variables-overview/connect-a-firebase-realtime-database
[24] Thunkable, "Thunkable Blocks Overview | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/blocks/thunkable-blocks-overview
[25] Thunkable, "Preview and Test your App | Thunkable Docs." Accessed: Jul. 18, 2024. [Online]. Available: https://docs.thunkable.com/getting-started/get-started/live-test
[26] Thunkable, "Figma Integration | Thunkable Docs." Accessed: Aug. 17, 2025. [Online]. Available: https://docs.thunkable.com/app-design/figma
[27] Arduino Docs, "millis() | Arduino Documentation," Arduino. Accessed: Jul. 26, 2025. [Online]. Available: https://docs.arduino.cc/language-reference/en/functions/time/millis/
[28] Thunkable, "Alerts & Notifications | Thunkable Docs." Accessed: Aug. 17, 2025. [Online]. Available: https://docs.thunkable.com/getting-started/sample-apps-and-tutorials/video-tutorials/alerts-and-notifications