Module 7: WiFi, HTTP(S), & MQTT(S)

Basics of WiFi Networking

WiFi Standards and Protocols

Wi-Fi is a technology that allows devices to connect to a wireless network and exchange data. Wi-Fi is based on the IEEE 802.11 standard, which defines the physical and data link layers of a network. There are several versions of the IEEE 802.11 standard, such as 802.11a, 802.11b, 802.11g, 802.11n, 802.11ac, and 802.11ax. Each version has different characteristics, such as frequency bands, data rates, modulation schemes, and range.

The ESP32 supports the following Wi-Fi standards:

- **802.11b**: Operates on the 2.4 GHz band with a maximum data rate of 11 Mbps.
- **802.11g**: Operates on the 2.4 GHz band with a maximum data rate of 54 Mbps.
- **802.11n**: Operates on either the 2.4 GHz or 5 GHz band, with a maximum data rate of 150 Mbps (single stream) or 300 Mbps (dual stream).

The ESP32 also supports the following Wi-Fi protocols:

- **WEP**: Wired Equivalent Privacy, a legacy encryption protocol that is insecure and not recommended.
- **WPA**: Wi-Fi Protected Access, an enhanced encryption protocol using TKIP (Temporal Key Integrity Protocol) or AES (Advanced Encryption Standard).
- **WPA2**: Wi-Fi Protected Access 2, a more secure encryption protocol using AES, providing stronger security than WPA.
- **WPA3**: Wi-Fi Protected Access 3, a new encryption protocol offering more robust security and privacy features than WPA2.

Network Types and Their Relevance

There are three main types of Wi-Fi networks that the ESP32 can operate in:

1. **Infrastructure**: The most common type of Wi-Fi network, where the ESP32 connects to an access point (AP) like a router or hotspot. The AP acts as a bridge between the ESP32 and the internet or other devices on the same network. The AP also assigns an IP address to the ESP32 and manages network traffic.
2. **Ad-hoc**: In this Wi-Fi network type, the ESP32 connects directly to another device without needing an AP. The ESP32 and the other device form a peer-to-peer network and assign IP addresses to each other. This network type is useful for temporary or local communication, such as file sharing or gaming.
3. **Wi-Fi Direct**: In this Wi-Fi network type, the ESP32 connects to another device that supports Wi-Fi Direct, such as a smartphone or printer. The ESP32 and the other device form a one-to-one network and communicate using established protocols like P2P (Peer-to-Peer) or Miracast. This network type is useful for specific applications like streaming or printing.

The ESP32 can operate in any of these network types, depending on the configured mode:

1. **Station mode (STA)**: The ESP32 acts as a station and connects to an AP. This is the default mode for the ESP32.
2. **Access Point mode (AP)**: The ESP32 acts as an AP, allowing other stations to connect to it. The ESP32 can also provide internet access to connected stations using NAT (Network Address Translation) or routing.
3. **Station/AP coexistence mode (STA+AP)**: The ESP32 operates as both a station and an AP simultaneously. The ESP32 can connect to another AP as a station and provide a separate network as an AP. This mode is useful for extending the range of an existing network or creating a bridge between two networks.
4. **NAN mode (NAN)**: The ESP32 acts as a node in a NAN (Neighbor Awareness Networking) network. NAN is a new Wi-Fi standard that allows devices to discover and communicate with each other without an AP. NAN is designed for low-power and proximity-based applications, such as social networking or location-based services.

IP Addressing, DHCP, DNS, and Other Network Fundamentals

An IP address is a unique identifier assigned to a device on a network. IP addresses consist of four numbers separated by dots, like `192.168.1.100`. Each number can range from 0 to 255. IP addresses can be static or dynamic. A static IP address remains constant, while a dynamic IP address is assigned by a DHCP (Dynamic Host Configuration Protocol) server and may change over time.

A **DHCP server** is a device that manages IP address allocation on a network. A DHCP server can be an AP, router, or dedicated server. The DHCP server assigns an IP address to a device upon request and releases it when the device disconnects or its lease expires. The DHCP server also provides other network information, such as subnet mask, gateway, and DNS server.

A **subnet mask** is a number that defines the size and structure of a network. It consists of four numbers separated by dots, such as `255.255.255.0`. Each number can be 255 or 0. The subnet mask specifies which part of an IP address belongs to the network and which part belongs to the host. For example, if the subnet mask is `255.255.255.0`, the first three numbers of the IP address represent the network, while the last number represents the host.

A **gateway** is a device that connects two or more networks and routes traffic between them. A gateway can be an AP, router, or dedicated device. The gateway has an IP address on each network it connects. For example, if a gateway connects a local network (`192.168.1.0/24`) to the internet (`0.0.0.0/0`), it has the IP address `192.168.1.1` on the local network and `1.2.3.4` on the internet.

A **DNS server** is a device that translates domain names into IP addresses. A domain name is a human-readable identifier for a website or service on the internet, such as `www.bing.com`. The DNS server maintains a database of domain names and their corresponding IP addresses. A DNS server can be an AP, router, or dedicated server. A DNS server can also cache previous query results to speed up resolution.

The ESP32 can obtain an IP address and other network information from a DHCP server when connected to an AP as a station. The ESP32 can also act as a DHCP server when operating as an AP and assign IP addresses to connected stations. The ESP32 can use a DNS server to resolve domain names to IP addresses when accessing the internet or other services.

Wi-Fi Capabilities of ESP32

Hardware Capabilities

The ESP32 supports the following Wi-Fi features:

- **WMM**: Wi-Fi Multimedia, a quality of service (QoS) feature that prioritizes traffic based on four access categories: voice, video, best effort, and background.
- **TX/RX A-MPDU**: Aggregated MAC Protocol Data Unit, a technique that combines multiple frames into a single transmission unit, reducing overhead and increasing throughput.
- **RX A-MSDU**: Aggregated MAC Service Data Unit, a technique that combines multiple frames from the same sender into one frame, reducing overhead and increasing throughput.
- **Immediate Block ACK**: A mechanism that allows the receiver to acknowledge multiple frames in a single response, reducing latency and improving efficiency.
- **Defragmentation**: A mechanism that reassembles fragmented frames at the receiver, enhancing reliability and performance.
- **Automatic Beacon Monitoring**: A hardware feature that tracks the timestamp and beacon interval of the connected AP, allowing for power savings and fast reconnection.

- **4 × Virtual Wi-Fi Interfaces**: Allows the ESP32 to create up to four logical Wi-Fi interfaces, such as station, softAP, or promiscuous mode, and operate them simultaneously.
- **Antenna Diversity**: Allows the ESP32 to switch between two antennas dynamically, depending on signal quality and power consumption.

Dual-Mode Functionality and Use Cases

The ESP32 can operate in two Wi-Fi modes: station mode and access point mode. In station mode, the ESP32 connects to an existing Wi-Fi network as a client. In access point mode, the ESP32 creates its own Wi-Fi network and allows other devices to join as clients. The ESP32 can also operate in both modes simultaneously, creating a Wi-Fi repeater or bridge between two networks.

The dual-mode functionality of the ESP32 enables various use cases, such as:

- **Wi-Fi Repeater**: The ESP32 can extend the range of an existing Wi-Fi network by connecting to it as a station and creating a new network as an access point. The ESP32 can also provide internet access to connected devices using NAT (Network Address Translation) or routing.
- **Wi-Fi Bridge**: The ESP32 can connect two different Wi-Fi networks by operating as a station in one network and as an access point in another network. The ESP32 can also transfer data between the two networks using TCP/IP or UDP protocols.
- **Wi-Fi Scanner**: The ESP32 can scan nearby Wi-Fi networks using promiscuous mode, allowing it to receive all packets on a particular channel. The ESP32 can also analyze packets and extract information such as SSID, MAC address, RSSI, channel, encryption type, etc.
- **Wi-Fi Sniffer**: The ESP32 can capture Wi-Fi traffic on a specific channel using promiscuous mode. The ESP32 can also send captured packets to a PC or smartphone for further analysis using tools like Wireshark.

ESP32 Security Protocols: Support for WPA, WPA2, WPA3

The ESP32 supports various security protocols to protect Wi-Fi communication from unauthorized access and attacks. The security protocols include:

- **WEP**: Wired Equivalent Privacy, a legacy encryption protocol that is insecure and not recommended.
- **WPA**: Wi-Fi Protected Access, an enhanced encryption protocol that uses TKIP (Temporal Key Integrity Protocol) or AES (Advanced Encryption Standard) algorithms.
- **WPA2**: Wi-Fi Protected Access 2, an improved encryption protocol that uses AES and provides stronger security than WPA.

- **WPA3**: Wi-Fi Protected Access 3, a new encryption protocol offering stronger security and privacy features than WPA2.

The ESP32 supports WPA, WPA2, and WPA3 in both station and access point modes. It can also operate in mixed mode, allowing the ESP32 to accept connections from devices that support different security protocols. For example, the ESP32 can create a Wi-Fi network supporting both WPA2 and WPA3, enabling devices that support either protocol to join the network.

The ESP32 also supports Protected Management Frames (PMF), a feature that encrypts and authenticates management frames, such as deauthentication, disassociation, and robust management frames. PMF prevents attacks using forged or spoofed management frames to disrupt Wi-Fi connections or perform man-in-the-middle attacks. PMF can be configured as optional, required, or disabled in both station and access point modes.

The ESP32 supports Wi-Fi Enterprise, a secure authentication mechanism for enterprise wireless networks. It uses a RADIUS server for user authentication before connecting to an access point. The authentication process is based on 802.1X policies and comes with various Extended Authentication Protocol (EAP) methods, such as TLS, TTLS, PEAP, and EAP-FAST. The ESP32 only supports Wi-Fi Enterprise in station mode.

Configuring ESP32 as a Wi-Fi Station (STA)

The ESP32 can operate as a Wi-Fi station, meaning it can connect to an existing Wi-Fi network as a client device. This mode is useful for accessing the internet or other network services. In this section, we'll look at how to configure the ESP32 as a Wi-Fi station and manage Wi-Fi connection events and system responses.

Connecting to an Existing Wi-Fi Network

```
#include <WiFi.h>
#define WIFI_SSID "my_wifi"
#define WIFI_PASS "my_password"

void setup() {
  // Initialize serial monitor
  Serial.begin(115200);

  // Connect to Wi-Fi network
  WiFi.begin(WIFI_SSID, WIFI_PASS);

  // Wait until connected or timeout
  uint8_t timeout = 30;
```

```
  while (WiFi.status() != WL_CONNECTED && timeout--) {
    delay(1000);
    Serial.print(".");
  }

  // Check connection status
  if (WiFi.status() == WL_CONNECTED) {
    Serial.println("Connected to Wi-Fi");
    Serial.println("IP Address: " + WiFi.localIP().toString());
  } else {
    Serial.println("Failed to connect to Wi-Fi");
  }
}

void loop() {
  // Do nothing
}
```

Managing Wi-Fi Connection Events and System Responses

The ESP32 can handle various Wi-Fi connection events and system responses using the `WiFiEvent` class. This class allows the ESP32 to register callback functions for different types of events, such as:

- **WiFiEventStationModeConnected**: This event occurs when the ESP32 successfully connects to an Access Point (AP). The callback function receives a `WiFiEventStationModeConnected` object containing information about the AP, such as SSID, BSSID, and channel.
- **WiFiEventStationModeDisconnected**: This event occurs when the ESP32 disconnects from the AP or fails to connect. The callback function receives a `WiFiEventStationModeDisconnected` object containing information about the AP and the reason for disconnection, such as `AUTH_EXPIRE`, `NO_AP_FOUND`, or `HANDSHAKE_TIMEOUT`.
- **WiFiEventStationModeGotIP**: This event occurs when the ESP32 obtains an IP address from the DHCP server. The callback function receives a `WiFiEventStationModeGotIP` object containing information about the IP address, subnet mask, and gateway.
- **WiFiEventStationModeAuthModeChanged**: This event occurs when the AP's authentication mode changes. The callback function receives a `WiFiEventStationModeAuthModeChanged` object containing information about the old and new authentication modes, such as `WIFI_AUTH_OPEN`, `WIFI_AUTH_WPA_PSK`, or `WIFI_AUTH_WPA3_PSK`.
- **WiFiEventStationModeDHCPTimeout**: This event occurs when the ESP32 fails to obtain an IP address from the DHCP server within the specified time. The callback function does not receive any parameters.

To use the `WiFiEvent` class, we need to follow this code:

```
#include <WiFi.h>
#define WIFI_SSID "my_wifi"
#define WIFI_PASS "my_password"

// Callback function for WiFiEventStationModeConnected event
void onStationModeConnected(WiFiEventStationModeConnected info) {
  Serial.println("Connected to AP");
  Serial.println("SSID: " + info.ssid);
  Serial.println("BSSID: " + info.bssid);
  Serial.println("Channel: " + String(info.channel));
}

// Callback function for WiFiEventStationModeDisconnected event
void onStationModeDisconnected(WiFiEventStationModeDisconnected info) {
  Serial.println("Disconnected from AP");
  Serial.println("SSID: " + info.ssid);
  Serial.println("BSSID: " + info.bssid);
  Serial.println("Reason: " + String(info.reason));
}

// Callback function for WiFiEventStationModeGotIP event
void onStationModeGotIP(WiFiEventStationModeGotIP info) {
  Serial.println("Obtained IP Address");
  Serial.println("IP: " + info.ip.toString());
  Serial.println("Mask: " + info.mask.toString());
  Serial.println("Gateway: " + info.gw.toString());
}

// Callback function for WiFiEventStationModeAuthModeChanged event
void onStationModeAuthModeChanged(WiFiEventStationModeAuthModeChanged info) {
  Serial.println("Authentication mode changed");
  Serial.println("Old mode: " + String(info.oldMode));
  Serial.println("New mode: " + String(info.newMode));
}

// Callback function for WiFiEventStationModeDHCPTimeout event
void onStationModeDHCPTimeout() {
  Serial.println("DHCP Timeout");
```

```
}

void setup() {
  // Initialize serial monitor
  Serial.begin(115200);

  // Register event handlers
  WiFi.onEvent(onStationModeConnected, SYSTEM_EVENT_STA_CONNECTED);
  WiFi.onEvent(onStationModeDisconnected, SYSTEM_EVENT_STA_DISCONNECTED);
  WiFi.onEvent(onStationModeGotIP, SYSTEM_EVENT_STA_GOT_IP);
  WiFi.onEvent(onStationModeAuthModeChanged, SYSTEM_EVENT_STA_AUTHMODE_CHANGE);
  WiFi.onEvent(onStationModeDHCPTimeout, SYSTEM_EVENT_STA_DHCP_TIMEOUT);

  // Connect to Wi-Fi network
  WiFi.begin(WIFI_SSID, WIFI_PASS);
}

void loop() {
  // Do nothing
}
```

Example Output:

```
Connected to AP
SSID: my_wifi
BSSID: 00:11:22:33:44:55
Channel: 6

Obtaining IP address
IP: 192.168.1.100
Mask: 255.255.255.0
Gateway: 192.168.1.1

Disconnected from AP
SSID: my_wifi
BSSID: 00:11:22:33:44:55
Reason: 200
```

Secure Storage and Management of Wi-Fi Credentials

The ESP32 can securely store and manage Wi-Fi credentials using the Preferences library. The Preferences library allows the ESP32 to save and retrieve key-value pairs in the non-volatile storage (NVS) partition. The Preferences library also supports encryption of stored data using the AES-256 algorithm.

To use the Preferences library, follow the code below:

```
#include <WiFi.h>
#include <Preferences.h>

#define WIFI_SSID "my_wifi"
#define WIFI_PASS "my_password"

// Create a Preferences object
Preferences preferences;

// Define a 32-byte encryption key
uint8_t enc_key[32] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                        0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10,
                        0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
                        0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F, 0x20 };

void setup()
{
   // Initialize serial monitor
   Serial.begin(115200);

   // Open a namespace called "wifi"
   preferences.begin("wifi", false); // false = read/write

   // Enable encryption with the encryption key
   preferences.setEncryptionKey(enc_key);

   // Store Wi-Fi SSID and password
   preferences.putString("ssid", WIFI_SSID);
   preferences.putString("pass", WIFI_PASS);

   // Close the namespace
   preferences.end();
}
```

To retrieve the SSID and Wi-Fi password from the NVS partition using the getString method, you can optionally enable encryption using the setEncryptionKey method. Here’s an example:

```
// Define a 32-byte encryption key
uint8_t enc_key[32] = { /* ... */ };

void setup()
{
  // Initialize serial monitor
  Serial.begin(115200);

  // Open a namespace called "wifi"
  preferences.begin("wifi", true);

  // Enable encryption with the encryption key
  preferences.setEncryptionKey(enc_key);

  // Retrieve Wi-Fi SSID and password
  String ssid = preferences.getString("ssid", "");
  String pass = preferences.getString("pass", "");

  // Close the namespace
  preferences.end();

  // Connect to the Wi-Fi network
  WiFi.begin(ssid.c_str(), pass.c_str());

  // Wait until connected or timeout
  uint8_t timeout = 30;
  while (WiFi.status() != WL_CONNECTED && timeout--)
  {
    delay(1000);
    Serial.print(".");
  }

  // Check connection status
  if (WiFi.status() == WL_CONNECTED)
  {
    Serial.println("Connected to Wi-Fi");
    Serial.println("IP Address: " + WiFi.localIP().toString());
```

```
    }
    else
    {
        Serial.println("Failed to connect to Wi-Fi");
    }
}

void loop()
{
    // Do nothing
}
```

Configuring ESP32 as a Wi-Fi Access Point (AP)

The ESP32 can also operate as a Wi-Fi access point, meaning it can create its own Wi-Fi network and allow other devices to join as clients. This mode is useful for creating a local network without an internet connection or router. In this section, we will look at how to set up the ESP32 as a Wi-Fi access point and how to manage connected client devices and their data flow.

Procedure to Initialize a Local Wi-Fi Network

To initialize a local Wi-Fi network with the ESP32 as an access point, follow these steps:

```
#include <WiFi.h>

#define WIFI_SSID "ESP32-Access-Point"
#define WIFI_PASS "123456789"

// Set the web server port to 80
WiFiServer server(80);

void setup()
{
    // Initialize serial monitor
    Serial.begin(115200);

    // Set Wi-Fi mode to AP
    WiFi.mode(WIFI_AP);
```

```
    // Create Wi-Fi network
    // WiFi.softAP(ssid, password, channel, hidden, max_connection)
    WiFi.softAP(WIFI_SSID, WIFI_PASS, 1, false, 4);

    // Print the IP address of the AP
    Serial.print("AP IP Address: ");
    Serial.println(WiFi.softAPIP());

    // Start web server
    server.begin();
}

void loop()
{
    // Listen for incoming clients
    WiFiClient client = server.available();
    if (client)
    {
        // If a new client connects, print a message
        Serial.println("New client.");

        // Create a String to hold incoming data
        String request = "";

        // Loop while the client is connected
        while (client.connected())
        {
            // If there are bytes to read from the client
            if (client.available())
            {
                // Read a byte and add it to the request
                char c = client.read();
                request += c;

                // If the byte is a newline character
                if (c == '\n')
                {
                    // If the request is empty, the client has sent an empty line
                    // This is the end of the HTTP request, so send a response
```

```
                if (request.length() == 0)
                {
                    // HTTP header always starts with response code and content type
                    // Then a blank line
                    client.println("HTTP/1.1 200 OK");
                    client.println("Content-type:text/plain");
                    client.println();

                    // Send the response body
                    client.println("Hello from ESP32 AP!");

                    // Break out of the loop
                    break;
                }
                else
                {
                    // Clear the request
                    request = "";
                }
            }
        }
    }

    // Close the connection
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

The ESP32 access point can be configured with various parameters, such as SSID, password, channel, SSID visibility, and maximum number of connections. These parameters can be passed as arguments to the `WiFi.softAP` method, as shown below:

```
// WiFi.softAP(ssid, password, channel, hidden, max_connection)

// Create an open Wi-Fi network with the default channel (1), SSID visibility (true), and maximum connections (4)
WiFi.softAP("ESP32-AP", NULL);

// Create a secure Wi-Fi network with the default channel (1), SSID visibility (true), and maximum connections (4)
WiFi.softAP("ESP32-AP", "123456789");
```

```
// Create a secure Wi-Fi network with a custom channel (6), SSID visibility (true), and maximum connections (4)
WiFi.softAP("ESP32-AP", "123456789", 6);

// Create a secure Wi-Fi network with a custom channel (6), SSID visibility (false), and maximum connections (4)
WiFi.softAP("ESP32-AP", "123456789", 6, false);

// Create a secure Wi-Fi network with a custom channel (6), SSID visibility (false), and maximum connections (2)
WiFi.softAP("ESP32-AP", "123456789", 6, false, 2);
```

- **SSID** is the name of the Wi-Fi network and can be up to 31 characters long.
- **Password** is the security key to join the network and must be at least 8 characters long. If the password is set to `NULL`, the network will be open, and anyone can join without a password.
- **Channel** is the frequency band used by the Wi-Fi network and can range from 1 to 13.
- The **hidden** parameter is a boolean value that determines whether the SSID is broadcasted. If set to `true`, the network will not appear in Wi-Fi scans, and clients will need to know the SSID to join.
- The **max_connection** parameter is the maximum number of clients that can connect to the network simultaneously and can range from 1 to 4.

HTTP/HTTPS Introduction

Overview of HTTP and HTTPS

- **HTTP** (Hypertext Transfer Protocol) is a set of rules for transferring data over the internet. HTTP is the most commonly used protocol for web browsing, allowing clients (such as web browsers) to request and receive web pages, images, videos, and other resources from a server. HTTP is based on a **request-response model**, where the client sends a request to the server, and the server responds with the requested resource or an error code. HTTP is a **stateless protocol**, meaning that each request and response is independent and does not remember previous interactions.
- **HTTPS** (Hypertext Transfer Protocol Secure) is an extension of HTTP that adds an additional layer of security. HTTPS uses **Transport Layer Security (TLS)** to encrypt and authenticate data exchanged between the client and server. HTTPS prevents eavesdropping, tampering, or modification of data, ensuring the privacy and integrity of communication. HTTPS also provides **identity verification** using digital certificates issued by trusted authorities. HTTPS is a **stateful protocol**, meaning that it maintains a secure connection between the client and server throughout the session.

Importance in IoT Communication

Internet of Things (IoT) refers to a network of physical devices, such as sensors, actuators, cameras, and microcontrollers, that are connected to the internet. HTTP/HTTPS is important for IoT communication because it allows IoT devices to interact with web servers and cloud platforms (e.g., Google Cloud, AWS, Microsoft Azure). HTTP/HTTPS enables IoT devices to send data to the cloud, receive commands or updates, or access web APIs.

Differences and Transition from HTTP to HTTPS

HTTP and HTTPS have several key differences:

- **Port**: HTTP uses port 80, while HTTPS uses port 443. This allows HTTP and HTTPS to coexist on the same network without interference.
- **Data Security**: HTTP transmits data in plaintext, while HTTPS encrypts data using TLS, protecting it from unauthorized access.
- **Authentication**: HTTP does not verify server identity, while HTTPS uses digital certificates for authentication.
- **Attack Protection**: HTTPS protects against attacks such as man-in-the-middle, phishing, and spoofing.
- **Performance**: HTTP is faster because it does not require encryption or authentication, while HTTPS has additional overhead.

Many websites have switched to HTTPS for better security and privacy. Some web browsers also mark HTTP sites as not secure to warn users.

Basics of Web Communication

Request-Response Model

The request-response model describes how web communication works. It is based on the idea that a client (e.g., web browser) sends a request to a server (e.g., web server), and the server responds.

1. **Request**: Consists of:
   - **Request Line**: Specifies the method, URL, and HTTP version.
   - **Request Headers**: Additional information about the request (e.g., host, content type).

- **Request Body**: Data that the client wants to send to the server (e.g., form inputs, files).

2. **Response**: Consists of:
   - **Status Line**: Specifies the HTTP version, status code, and status message.
   - **Response Headers**: Additional information about the response (e.g., content type, content length).
   - **Response Body**: Data that the server sends back to the client (e.g., web pages, images).

HTTP Methods: GET, POST, PUT, DELETE, and Use Cases

- **GET**: Requests a resource from the server. It is safe and idempotent. Example: `GET /index.html`.
- **POST**: Sends data to the server to create or update a resource. It is not safe or idempotent. Example: `POST /login`.
- **PUT**: Replaces a resource on the server. It is not safe, but it is idempotent. Example: `PUT /profile`.
- **DELETE**: Deletes a resource from the server. It is not safe or idempotent. Example: `DELETE /post/123`.

Status Codes and Their Meaning

Status codes are numbers that indicate the result of a request. They are grouped as follows:

- **1xx**: Informational. Indicates the request has been received and is being processed.
- **2xx**: Success. Indicates the request was successfully completed (e.g., `200 OK`).
- **3xx**: Redirection. Indicates the request needs to be redirected to another URL (e.g., `301 Moved Permanently`).
- **4xx**: Client Error. Indicates the request is invalid or cannot be fulfilled (e.g., `404 Not Found`).
- **5xx**: Server Error. Indicates the server encountered an error (e.g., `500 Internal Server Error`).

HTTP Communication with ESP32

Creating a Basic HTTP Client

To use the ESP32 as an HTTP client, we need to include the following libraries:

```
#include <WiFi.h>
#include <HTTPClient.h>
```

The WiFi.h library allows us to connect to a Wi-Fi network, and the HTTPClient.h library allows us to make HTTP requests.

We also need to declare the Wi-Fi SSID and password, as well as the hostname and pathname of the server we want to communicate with. For example:

```
const char* WIFI_SSID = "YOUR_WIFI_SSID"; // change this
const char* WIFI_PASSWORD = "YOUR_WIFI_PASSWORD"; // change this
String HOST_NAME = "http://YOUR_DOMAIN.com"; // change this
String PATH_NAME = "/products/arduino"; // change this
```

In the setup() function, we need to initialize the serial monitor and connect to the Wi-Fi network. We can use the WiFi.begin() and WiFi.status() functions to do this. For example:

```
void setup() {
  Serial.begin(115200);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println();
  Serial.println("Connected to Wi-Fi");
  Serial.println("IP Address: ");
  Serial.println(WiFi.localIP());
}
```

In the loop() function, we need to create an HTTP client object and use the begin() method to specify the URL we want to request. For example:

```
void loop() {
  HTTPClient http;
  http.begin(HOST_NAME + PATH_NAME);
  // further code will follow
}
```

Sending an HTTP GET Request to an API

To send an HTTP GET request to an API, we use the GET() method of the HTTP client object. This method returns an integer representing the status code of the response. For example:

```
int httpCode = http.GET();
```

We can use the httpCode variable to check if the request was successful. If httpCode is positive, it means the server responded with a valid status code. If httpCode is negative, it means there was an error in the request. For example:

```
if (httpCode > 0) {
  Serial.print("HTTP GET request completed, code: ");
  Serial.println(httpCode);
} else {
  Serial.print("HTTP GET request failed, error: ");
  Serial.println(http.errorToString(httpCode));
}
```

If the request is successful, we can use the getString() method of the HTTP client object to get the response content as a string. This method returns a string containing the data sent by the server. For example:

```
if (httpCode == HTTP_CODE_OK) {
  String payload = http.getString();
  Serial.println("HTTP Response: ");
  Serial.println(payload);
}
```

The response content may contain various types of data, depending on the API. For example, it could contain JSON, XML, HTML, plain text, or binary data. We need to parse the response content according to the data type we expect. For instance, if the response contains JSON data, we need to use a JSON parsing library to extract the values we need.

Sending an HTTP POST Request

To send an HTTP POST request to the server, we use the POST() method of the HTTP client object. This method takes a string as an argument, representing the data we want to send in the request body. For example:

```
String data = "temperature=26&humidity=70"; // example data
int httpCode = http.POST(data);
```

We can use the same httpCode variable to check if the request was successful, as we did with the GET request. If the request is successful, we can also use the same getString() method to get the response content as a string, just like we did with the GET request.

The data we send in the request body can be in various formats, depending on the server. For example, it could be in query string format, JSON format, XML format, or binary format. We need to format the data according to the server's expectations. We may also need to set the appropriate content type header for the request, using the addHeader() method of the HTTP client object. For example:

```
http.addHeader("Content-Type", "application/json"); // example header
String data = "{\"temperature\":26,\"humidity\":70}"; // example data in JSON format
int httpCode = http.POST(data);
```

Parsing JSON Responses and Error Handling

To parse JSON responses, we need to include the following library:

```
#include <ArduinoJson.h>
```

The ArduinoJson.h library allows us to parse and generate JSON data. We can use the deserializeJson() function to parse a JSON string into a JSON object. For example:

```
String payload = http.getString(); // get the response content as a string
StaticJsonDocument<200> doc; // create a JSON document object with a capacity of 200 bytes
deserializeJson(doc, payload); // parse the JSON string into the JSON document object
```

We can use the doc variable to access values within the JSON object, using the [] operator. For example, if the JSON object contains a key called "temperature" and a key called "humidity," we can retrieve their values as follows:

```
int temperature = doc["temperature"]; // get the value of "temperature" as an integer
int humidity = doc["humidity"]; // get the value of "humidity" as an integer
Serial.print("Temperature: ");
Serial.println(temperature);
Serial.print("Humidity: ");
```

```
Serial.println(humidity);
```

We can also use the doc variable to check if the JSON object contains a specific key, using the containsKey() method. For example, if we want to check if the JSON object contains a key called "error," we can do as follows:

```
if (doc.containsKey("error")) {
  Serial.println("Error in JSON response");
}
```

We can also use the doc variable to check if JSON parsing was successful, using the isNull() method. For example, if JSON parsing fails, the doc variable will be null, and we can handle it as follows:

```
if (doc.isNull()) {
  Serial.println("Invalid JSON response");
}
```

Code Snippets for GET and POST Requests

The following code snippets show how to make GET and POST requests using the ESP32 as an HTTP client. This code assumes that the server is running on the same network as the ESP32, with a server IP address of `192.168.1.100`. It also assumes that the server responds with JSON data containing keys "temperature" and "humidity".

GET Request

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>

const char* WIFI_SSID = "YOUR_WIFI_SSID"; // change this
const char* WIFI_PASSWORD = "YOUR_WIFI_PASSWORD"; // change this
String HOST_NAME = "http://192.168.1.100"; // change this
String PATH_NAME = "/get_data";

void setup() {
  Serial.begin(115200);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
```

```
    Serial.print(".");
    delay(1000);
  }
  Serial.println();
  Serial.println("Connected to Wi-Fi");
  Serial.println("IP Address: ");
  Serial.println(WiFi.localIP());
}

void loop() {
  HTTPClient http;
  http.begin(HOST_NAME + PATH_NAME);

  int httpCode = http.GET();
  if (httpCode > 0) {
    Serial.print("HTTP GET request completed, code: ");
    Serial.println(httpCode);

    if (httpCode == HTTP_CODE_OK) {
      String payload = http.getString();
      Serial.println("HTTP Response: ");
      Serial.println(payload);

      StaticJsonDocument<200> doc;
      deserializeJson(doc, payload);

      if (doc.isNull()) {
        Serial.println("Invalid JSON response");
      } else {
        if (doc.containsKey("error")) {
          Serial.println("Error in JSON response");
        } else {
          int temperature = doc["temperature"];
          int humidity = doc["humidity"];
          Serial.print("Temperature: ");
          Serial.println(temperature);
          Serial.print("Humidity: ");
          Serial.println(humidity);
        }
      }
```

```
    }
  } else {
    Serial.print("HTTP GET request failed, error: ");
    Serial.println(http.errorToString(httpCode));
  }

  http.end();
  delay(5000);
}
```

POST Request

The following code demonstrates how to make a POST request using the ESP32 as an HTTP client. This code assumes that the server is running on the same network as the ESP32, with the server IP address `192.168.1.100`. The server is expected to respond with JSON data containing the keys "temperature" and "humidity".

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>

const char* WIFI_SSID = "YOUR_WIFI_SSID"; // change this
const char* WIFI_PASSWORD = "YOUR_WIFI_PASSWORD"; // change this
String HOST_NAME = "http://192.168.1.100"; // change this
String PATH_NAME = "/post_data";

void setup() {
  Serial.begin(115200);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println();
  Serial.println("Connected to Wi-Fi");
  Serial.println("IP Address: ");
  Serial.println(WiFi.localIP());
}
```

```
void loop() {
  HTTPClient http;
  http.begin(HOST_NAME + PATH_NAME);
  http.addHeader("Content-Type", "application/json"); // set content-type to JSON

  StaticJsonDocument<100> doc; // create a JSON document with a 100-byte capacity
  doc["temperature"] = 26; // set the "temperature" key value to 26
  doc["humidity"] = 70; // set the "humidity" key value to 70

  String data; // create a string to store JSON data
  serializeJson(doc, data); // serialize the JSON document into the string

  int httpCode = http.POST(data); // send POST request with JSON data
  if (httpCode > 0) {
    Serial.print("HTTP POST request done, code: ");
    Serial.println(httpCode);

    if (httpCode == HTTP_CODE_OK) {
      String payload = http.getString();
      Serial.println("HTTP Response: ");
      Serial.println(payload);

      StaticJsonDocument<200> doc;
      deserializeJson(doc, payload);

      if (doc.isNull()) {
        Serial.println("Invalid JSON response");
      } else {
        if (doc.containsKey("error")) {
          Serial.println("Error in JSON response");
        } else {
          int temperature = doc["temperature"];
          int humidity = doc["humidity"];
          Serial.print("Temperature: ");
          Serial.println(temperature);
          Serial.print("Humidity: ");
          Serial.println(humidity);
        }
      }
    }
```

```
  } else {
    Serial.print("HTTP POST request failed, error: ");
    Serial.println(http.errorToString(httpCode));
  }

  http.end();
  delay(5000);
}
```

Creating an HTTP Server with ESP32

Configuring ESP32 as an HTTP Server

To use ESP32 as an HTTP server, we need to include the following libraries:

```
#include <WiFi.h>
#include <WebServer.h>
```

The WiFi.h library allows us to connect to a Wi-Fi network, and the WebServer.h library enables us to create and handle HTTP requests.

We also need to declare the Wi-Fi SSID, Wi-Fi password, and the port number for the web server. For example:

```
const char* WIFI_SSID = "YOUR_WIFI_SSID"; // change this
const char* WIFI_PASSWORD = "YOUR_WIFI_PASSWORD"; // change this
const int SERVER_PORT = 80; // default port for HTTP
```

In the setup() function, we need to initialize the serial monitor and connect to the Wi-Fi network. We can use the WiFi.begin() and WiFi.status() functions to do this. For example:

```
void setup() {
  Serial.begin(115200);
  Serial.print("Connecting to ");
  Serial.println(WIFI_SSID);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
```

```
  }

  Serial.println();
  Serial.println("Connected to Wi-Fi");
  Serial.println("IP Address: ");
  Serial.println(WiFi.localIP());
}
```

Next, we need to create a web server object and pass the port number as an argument. For example:

```
WebServer server(SERVER_PORT); // create a web server on port 80
```

Then, we define routes and handlers for the web server. A route is a path or URL that the client requests from the server. A handler is a function that executes when a specific route is requested. We can use the on() method from the web server object to define routes and handlers. For example:

```
server.on("/", handleRoot); // calls the 'handleRoot' function when the root route is requested
```

The handleRoot function is a custom function that we need to define later. It will handle requests for the root route, which is the default route when we access the web server.

Finally, we start the web server using the begin() method from the web server object. For example:

```
server.begin(); // start the web server
Serial.println("Web server started");
```

To define routes and handlers for the web server, we need to write custom functions for each route. These functions should have the following structure:

```
void handleRoute() {
  // code to handle the request for the route
}
```

The handleRoute function must have the same name as the one we pass to the on() method. For example, if we define a route like this:

```
server.on("/about", handleAbout);
```

We need to write a function like this:

```
void handleAbout() {
  // code to handle the request for the /about route
}
```

Within functions, we can use various methods and properties of the web server object to handle requests. Some of the most useful methods and properties are:

- **`server.method()`**: This method returns the HTTP method of the request, such as GET, POST, PUT, or DELETE. We can use this method to check the type of request made by the client and respond accordingly.
- **`server.arg(name)`**: This method returns the value of a query parameter or form field in the request with the given name. For example, if the request URL is `/login?username=alice&password=1234`, we can retrieve the values of the `username` and `password` parameters like this:

  ```
  String username = server.arg("username"); // username = "alice"
  String password = server.arg("password"); // password = "1234"
  ```

- **`server.send(code, type, content)`**: This method sends a response to the client with a status code, content type, and content. For example, if we want to send a plain text response with a 200 (OK) code, we can do this:

  ```
  server.send(200, "text/plain", "Hello, world!");
  ```

- **`server.sendHeader(name, value)`**: This method sends a custom header in the response with a name and value. For example, if we want to set a cookie for the client, we can do this:

  ```
  server.sendHeader("Set-Cookie", "user=alice");
  ```

- **`server.client()`**: This property returns a reference to the client object connected to the server. We can use this object to access low-level information about the client, such as IP address, MAC address, or port number. For example, if we want to get the client's IP address, we can do this:

  ```
  IPAddress clientIP = server.client().remoteIP(); // get the IP address of the client
  ```

To serve HTML and CSS content from the ESP32, we need to write HTML and CSS code as a string in our sketch. For example, we can write a simple HTML code like this:

```
String HTML = "<!DOCTYPE html>"
        "<html>"
        "<head>"
        "<style>"
        "h1 {color: blue;}"
        "</style>"
```

```
        "</head>"
        "<body>"
        "<h1>Hello, world!</h1>"
        "</body>"
        "</html>";
```

Then, we can send the HTML code as a response to the client using the server.send() method. We need to specify the content type as "text/html" to inform the browser how to interpret the content. For example, we can write a function to handle the root route like this:

```
void handleRoot() {
  server.send(200, "text/html", HTML); // send the HTML code as a response
}
```

We can also use variables or expressions within our HTML code to make it dynamic. For example, we can use the millis() function to display the current time on the web page. We can do this by combining the HTML code with the millis() function using the + operator. For example:

```
String HTML = "<!DOCTYPE html>"
        "<html>"
        "<head>"
        "<style>"
        "h1 {color: blue;}"
        "</style>"
        "</head>"
        "<body>"
        "<h1>Hello, world!</h1>"
        "<p>The current time is " + String(millis()) + " milliseconds.</p>"
        "</body>"
        "</html>";
```

Implementing HTTPS with SSL/TLS

Explanation of SSL/TLS Encryption

SSL stands for **Secure Sockets Layer**, and TLS stands for **Transport Layer Security**. Both are cryptographic protocols that provide encryption and authentication for data transmitted over the internet. They are commonly used to secure web communications, like HTTPS, as well as other protocols, such as SMTP, FTP, and MQTT.

SSL/TLS encryption works by establishing a secure connection between a client (such as a web browser) and a server (such as a web server) through a process called an SSL/TLS handshake. The handshake involves the following steps:

1. The client sends a **ClientHello** message to the server, indicating the SSL/TLS version, supported cipher suites, and a random number.
2. The server responds with a **ServerHello** message, selecting the SSL/TLS version, cipher suite, and another random number. The server also sends its certificate, which contains its public key and identity information, signed by a trusted authority.
3. The client verifies the server's certificate and, optionally, sends its own certificate if the server requests it. The client also generates a **premaster secret**, which is a random number, and encrypts it with the server's public key. The client then sends the encrypted premaster secret to the server.
4. The server decrypts the premaster secret using its private key. Both the client and server use the premaster secret and random numbers to generate a **master secret**, which is a shared secret key. They also use the master secret to generate a **session key**, which is a symmetric key used to encrypt and decrypt data.
5. The client sends a **Finished** message to the server, containing a hash of previous messages, encrypted with the session key. The server does the same, sending a **Finished** message to the client.
6. The client and server can now exchange data securely, using the session key to encrypt and decrypt data.

Using `WiFiClientSecure.h` for Secure Server Communication

To simplify this explanation, using `WiFiClientSecure` is essentially the same as using a regular `WiFiClient`. However, there is one key difference to note. Before you initiate a connection to the target server, you need to define the server's certificate in a constant variable, for example, named `server_cert`. After that, you can run `client.setCACert(server_cert)` on the client. Once this step is completed, you will use `WiFiClientSecure` in a similar way to how you would use a regular `WiFiClient`. This means all operations you perform afterward—such as sending and receiving data—will be done over a secure connection, as it is authenticated with the correct certificate.

Introduction to MQTT and MQTTS

Overview of MQTT in IoT

MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe messaging protocol designed for low-bandwidth, high-latency, and unreliable networks. It is widely used in the

Internet of Things (IoT) to enable communication between devices and applications.

MQTT operates on a **broker** and **client** principle. The **broker** is a central server that manages message distribution among clients. **Clients** are devices or applications connected to the broker and exchange messages using topics. A **topic** is a hierarchical string that identifies the content and scope of a message. For example, `home/temperature` is a topic representing temperature data for a home.

Clients can publish messages to a topic or subscribe to a topic to receive messages from the broker. The broker ensures that messages are delivered to subscribed clients according to the **Quality of Service (QoS)** level specified by the publisher. The QoS level indicates message delivery guarantees. MQTT has three QoS levels:

1. **QoS 0**: At most once delivery. Messages are delivered at most once or not at all. This QoS level is the fastest and simplest but does not provide reliability. Suitable for scenarios where occasional message loss is acceptable, such as sensor data or telemetry.
2. **QoS 1**: At least once delivery. Messages are delivered at least once, but they may be delivered more than once. This level ensures messages are received by the broker and subscribed clients but may introduce duplicate messages. Suitable for scenarios where message loss is unacceptable, but duplicates can be tolerated or filtered, such as alerts or notifications.
3. **QoS 2**: Exactly once delivery. Messages are delivered exactly once. This is the most reliable and complex level but also introduces more overhead and latency. Suitable for scenarios where both message loss and duplication are unacceptable, such as financial transactions or commands.

MQTT is a simple and flexible protocol that can be implemented on various platforms and devices. It has many advantages for IoT applications, such as:

1. **Low overhead**: MQTT packet headers are only 2 bytes, which minimizes network bandwidth and resource consumption.
2. **Scalability**: Brokers can handle millions of simultaneous connections and messages from clients, allowing large-scale IoT deployments.
3. **Security**: MQTT supports **Transport Layer Security (TLS)** encryption and authentication, which protects transmitted data from eavesdropping and tampering.
4. **Interoperability**: MQTT is based on the standard **TCP/IP** stack, making it compatible with any network infrastructure and device. It also supports various data formats, such as JSON, XML, or binary, making integration with different applications and services easy.

Advantages of MQTT for ESP32 Devices

Using MQTT with ESP32 devices has several advantages, such as:

- **Easy integration**: ESP32 natively supports the MQTT protocol, as it includes a built-in MQTT client library (ESP-MQTT) that can be used with the Arduino IDE or ESP-IDF

framework. The ESP-MQTT library provides a simple and convenient way to connect, publish, and subscribe to MQTT brokers and topics, without requiring additional libraries or dependencies.
- **Low power consumption**: The ESP32 has various power modes, such as active, light sleep, deep sleep, and hibernation, which allow for power consumption adjustments based on application needs. Using MQTT with ESP32 devices can further reduce power consumption, as MQTT is a lightweight protocol that minimizes network traffic and CPU usage. Moreover, MQTT supports a **keep-alive** mechanism, allowing devices to maintain a connection with the broker without sending or receiving data until a message is available or a timeout occurs. This allows the device to save power by entering low-power mode when inactive and only waking when needed.
- **Reliable communication**: Using MQTT with ESP32 devices can improve communication reliability, as MQTT supports various QoS levels to ensure message delivery according to application requirements. Additionally, MQTT supports **Last Will and Testament (LWT)** functionality, allowing a device to send a predefined message to the broker in case of an unexpected disconnection, such as power failure or network disruption. This allows the broker and other clients to be informed of the device's status and take appropriate action.

Comparing MQTT and MQTTS (MQTT over SSL/TLS)

MQTT is a protocol that operates on the TCP/IP stack, providing reliable and ordered data delivery. However, TCP/IP does not offer security or encryption for data, making it vulnerable to various attacks, such as:

- **Eavesdropping**: An attacker can intercept and read transmitted data, potentially exposing sensitive or confidential information, such as passwords, personal data, or commands.
- **Tampering**: An attacker can modify or inject data being transmitted, potentially altering the intended communication's behavior or outcomes, such as changing sensor readings, triggering false alarms, or executing malicious commands.
- **Spoofing**: An attacker can impersonate another device or application involved in the communication, potentially deceiving or misleading the recipient, such as sending fake messages, requesting unauthorized access, or stealing data.

To protect data from these attacks, MQTT can be used with **SSL/TLS**, a protocol that provides security and encryption for data. SSL/TLS stands for **Secure Sockets Layer/Transport Layer Security** and is widely used to secure internet communications. SSL/TLS works by establishing a secure channel between the sender and receiver, which involves the following steps:

- **Handshake**: The sender and receiver exchange information and agree on parameters for the secure channel, such as protocol version, cipher suite, and key exchange method. The

sender and receiver also verify each other's identity using certificates, which are digital documents containing the owner's public key and identity information and signed by a trusted authority. This step ensures the authenticity and integrity of the communication parties.
- **Encryption**: The sender and receiver generate a shared secret key using the key exchange method agreed upon during the handshake, such as **Diffie-Hellman** or **RSA**. The sender and receiver use this key to encrypt and decrypt transmitted data over the secure channel, using the cipher suite agreed upon during the handshake, such as **AES** or **ChaCha20**. This step ensures the confidentiality and integrity of the data.
- **Termination**: The sender and receiver close the secure channel and release the resources, such as keys and certificates, used for communication. This step ensures the security and efficiency of the communication.

MQTT over SSL/TLS, also known as **MQTTS**, is a variant of MQTT that uses SSL/TLS to secure data. MQTTS has the same features and functionality as MQTT, except it uses a different port number (8883 instead of 1883) and a different URI scheme (`mqtts` instead of `mqtt`) to indicate SSL/TLS usage. MQTTS offers several advantages over MQTT, such as:

- **Data protection**: MQTTS protects data from eavesdropping, tampering, and spoofing by using SSL/TLS encryption and authentication. This ensures that only the intended recipient can read, modify, or impersonate the communication.
- **Compliance**: MQTTS helps meet legal, regulatory, or industry requirements for data protection, privacy, or cybersecurity, such as **GDPR** in Europe or **HIPAA** in the United States.
- **Trust**: MQTTS increases trust and confidence between the communication parties by verifying each other's identity using certificates, which proves that they are who they claim to be and that they are authorized to participate in the communication.

Basics of MQTT Protocol

Topics

A topic is a hierarchical string that identifies the content and scope of a message. Topics are used by the broker to filter and distribute messages among clients. A topic consists of one or more topic levels, separated by forward slashes (`/`). For example, `home/temperature` is a topic with two levels: `home` and `temperature`.

Topics are case-sensitive and can contain UTF-8 encoded characters, except:

- Wildcard characters: `+` and `#`
- Control characters: `U+0000` to `U+001F` and `U+007F`
- Space character: `U+0020`

Wildcard characters are used to create topic filters, allowing clients to subscribe to multiple topics with a single subscription. There are two types of wildcards:

- **Single-level wildcard (`+`)**: Matches any single topic level. For example, `home/+` matches `home/temperature`, `home/humidity`, `home/light`, etc.
- **Multi-level wildcard (`#`)**: Matches the specified level and all following levels. For example, `home/#` matches `home/temperature`, `home/humidity`, `home/light`, `home/temperature/average`, etc.

Wildcards can only be used in topic filters, not in topic names. Topic filters must follow these rules:

- The multi-level wildcard (`#`) must be the last character in the topic filter and be preceded by a forward slash (`/`) or be the only character in the filter. For example, `home/#` and `#` are valid, but `home/#/light` and `home#` are not.
- The single-level wildcard (`+`) can be used at any level in the filter and must be surrounded by forward slashes (`/`) or be the first or last character in the filter. For example, `+/temperature`, `home/+`, and `+/+/light` are valid, but `home+` and `+home` are not.

Broker

The broker is a central server that manages MQTT communication among clients. It is responsible for:

- Establishing and maintaining connections with clients
- Receiving and storing messages from publishers
- Filtering and sending messages to subscribers
- Handling QoS levels, session persistence, and authentication

Brokers can be hosted on cloud platforms, such as AWS IoT Core, Azure IoT Hub, or Google Cloud IoT Core, or on local machines, such as a Raspberry Pi, using MQTT broker software like Mosquitto, EMQ X, or HiveMQ.

Clients

A client is a device or application that connects to a broker and exchanges messages using topics. Clients can be publishers, subscribers, or both. Publishers are clients that publish messages to topics, while subscribers are clients that subscribe to topics to receive messages from the broker. Clients can publish or subscribe to multiple topics simultaneously.

Clients can use various MQTT client libraries or tools to connect to the broker and perform MQTT operations, such as:

- **Paho**: An open-source set of MQTT client libraries for various programming languages, such as Python, Java, C, C++, JavaScript, etc.

- **MQTT.fx**: A graphical MQTT client tool that allows users to connect, publish, and subscribe to MQTT brokers and topics and monitor message exchanges.
- **Mosquitto**: A command-line MQTT client tool that allows users to connect, publish, and subscribe to MQTT brokers and topics and perform various MQTT operations.

Publish/Subscribe Mechanism

MQTT uses a publish/subscribe mechanism to enable communication between clients and the broker. The publish/subscribe mechanism works as follows:

- The publisher client publishes a message to a topic, specifying the QoS level and retain flag. A message consists of a fixed header, a variable header, and a payload. The fixed header contains the message type, QoS level, retain flag, and remaining length. The variable header contains the topic name and packet identifier. The payload contains the application data.
- The broker receives the message and stores it according to the QoS level and retain flag. The broker also checks other clients' subscriptions to determine which clients are interested in the message based on the topic name and topic filter.
- The broker sends the message to subscribing clients based on the QoS level and retain flag. The broker and clients exchange acknowledgments to ensure message delivery.

Quality of Service (QoS) Levels: 0, 1, and 2

- **QoS 0**: The lowest and fastest QoS level, but provides no reliability. The publisher sends the message only once to the broker without waiting for any acknowledgment. The broker sends the message only once to the subscriber, without waiting for acknowledgment. Messages may be lost or duplicated due to network failures or congestion.
- **QoS 1**: The intermediate QoS level, ensuring the message is delivered at least once, but may be delivered more than once. The publisher sends the message to the broker with a packet identifier and waits for a PUBACK (publish acknowledgment) from the broker. If the publisher doesn’t receive PUBACK within a certain time, it resends the message with the same packet identifier. The broker sends the message to the subscriber with the same packet identifier and waits for PUBACK from the subscriber. If the broker doesn’t receive PUBACK within a certain time, it resends the message with the same packet identifier.
- **QoS 2**: The highest and most reliable QoS level, ensuring the message is delivered exactly once. The publisher and subscriber exchange four messages to complete the delivery. The publisher sends the message to the broker with a packet identifier and waits for PUBREC (publish received) from the broker. The publisher then sends PUBREL (publish release) to the broker with the same packet identifier and waits for PUBCOMP (publish complete) from the broker. The broker sends the message to the subscriber with the same packet identifier and waits for PUBREC from the subscriber. The broker then sends

PUBREL to the subscriber with the same packet identifier and waits for PUBCOMP from the subscriber.

Retained Messages

A retained message is a message stored by the broker for a topic and delivered to new subscribers when they subscribe to that topic. A publisher can mark a message as retained by setting the retain flag to 1 in the fixed header. The broker will store the last retained message for each topic and replace it with a new one if the publisher publishes another retained message to the same topic. A publisher can also delete the retained message by publishing a message with a null payload and retain flag set to 1 to the same topic.

Retained messages are useful for providing the latest status or information about a topic to new subscribers without waiting for the publisher to publish a new message.

Last Will

The last will is a message sent by the broker on behalf of a client if the client disconnects unexpectedly. A client can specify a last will message when connecting to the broker by providing the topic, payload, QoS level, and retain flag of the message. The broker will store the last will message until the client disconnects normally or the keep-alive interval expires. If the client disconnects abnormally, the broker publishes the last will message to the topic and delivers it to subscribers according to the QoS level and retain flag.

The last will message is useful for notifying other clients about the status or reason for the client’s disconnection and for taking appropriate action.

MQTT Communication with ESP32

Connecting to the MQTT Broker

To communicate with an MQTT broker, ESP32 needs to use an MQTT client library compatible with the Arduino IDE. One of the most popular and easy-to-use libraries is the **PubSubClient** library by Nick O'Leary. This library provides a simple and convenient way to connect, publish, and subscribe to MQTT brokers and topics without additional libraries or dependencies.

To install the PubSubClient library, follow these steps:

1. Open Arduino IDE and go to **Sketch > Include Library > Manage Libraries**
2. Search for "PubSubClient" and select the latest version
3. Click "Install" and wait for the installation to complete
4. Close the Library Manager window

Code Example

```
// Include Wi-Fi and MQTT libraries
#include <WiFi.h>
#include <PubSubClient.h>

// Define Wi-Fi and MQTT credentials
const char* ssid = "your_wifi_ssid";
const char* password = "your_wifi_password";
const char* mqtt_server = "your_mqtt_broker_address";

// Create Wi-Fi client object
WiFiClient wifiClient;

// Create PubSubClient object
PubSubClient mqttClient(wifiClient);

// Connect to Wi-Fi network
void setup_wifi() {
    Serial.print("Connecting to ");
    Serial.println(ssid);

    // Connect to Wi-Fi network
    WiFi.begin(ssid, password);

    // Wait for connection to establish
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(".");
        delay(1000);
    }

    Serial.println("");
    Serial.println("WiFi connected");
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
}

// Connect to MQTT broker
void reconnect() {
    while (!mqttClient.connected()) {
        Serial.print("Attempting MQTT connection...");
```

```
        String clientId = "ESP32Client-";
        clientId += String(random(0xffff), HEX);

        // Try connecting to the broker
        if (mqttClient.connect(clientId.c_str())) {
            Serial.println("connected");
            mqttClient.subscribe("esp32/output");
        } else {
            Serial.print("failed, rc=");
            Serial.print(mqttClient.state());
            Serial.println(" try again in 5 seconds");
            delay(5000);
        }
    }
}

// Setup function runs once when ESP32 starts
void setup() {
    Serial.begin(115200);
    setup_wifi();
    mqttClient.setServer(mqtt_server, 1883);
    mqttClient.setCallback(callback);
}

// Loop function runs repeatedly after setup
void loop() {
    if (!mqttClient.connected()) {
        reconnect();
    }
    mqttClient.loop();
}
```

Revision #2
Created 4 November 2024 00:32:26 by AM
Updated 7 November 2024 03:08:06 by AM