Build applications using Wi-Fi library

- Wifi.h provides Wi-Fi-based functionality in IoT applications.
  - Basic Wi-Fi connectivity features like connecting to networks.
  - Client station or access point.
  - Data exchange over Wi-Fi
- Wifi.h focuses on synchronous operations for Wi-Fi connection management.

© Copyright National University of Singapore. All Rights Reserved.

Build applications using Wi-Fi library

- Applications
  - Wi-Fi Station Mode/Wi-Fi Access Point Mode/Wi-Fi Client + Access Point Mode
  - Basic HTTP Client/Server Applications
  - Web Server Hosting
  - Wi-Fi Scanning and Network Diagnostics
  - OTA (Over-the-Air) Firmware Updates
  - Wi-Fi Mesh Networking
  - Smart Config (Easy Wi-Fi Configuration)
  - MQTT Client (Message Queuing Telemetry Transport)

© Copyright National University of Singapore. All Rights Reserved.

Hypertext Transfer Protocol (HTTP)

- An application-layer protocol for transmitting hypermedia documents.
- Define the client and server model between a client (a web browser) and a server (a webserver) and the message format between them.
- Used in many other applications -- REST API, IoT, multimedia streaming, remote device control, distributed systems and webhooks.

© Copyright National University of Singapore. All Rights Reserved.

Hypertext Transfer Protocol (HTTP)

- Request-response cycle
  - Request: The client sends an HTTP request to the server, asking for data or action. The request contains a method (like GET, POST, PUT, etc.), headers, and sometimes a body.
  - Response: The server processes the request and sends back a response, which includes a status code (like 200 OK, 404 Not Found), headers, and possibly data (like an HTML page or JSON data).

© Copyright National University of Singapore. All Rights Reserved.

Hypertext Transfer Protocol (HTTP)

- HTTP Status
  - 200 OK: The request was successful.
  - 404 Not Found: The requested resource was not found.
  - 500 Internal Server Error: There was a problem on the server side.
  - 301 Moved Permanently: The resource has been moved to a different URL.
- Headers:
  - HTTP headers are additional information sent along with the request or response.
  - They can contain details like content type (HTML, JSON, etc.), authorization, cache settings, user agent, and more.

© Copyright National University of Singapore. All Rights Reserved.

Hypertext Transfer Protocol (HTTP)

- HTTP is stateless
  - Each request from a client to a server is independent.
  - The server doesn't retain any information from previous requests unless cookies or session management techniques are used.
- Asynchronous handling of HTTP requests.
- Support WebSockets, making real-time data communication possible.

© Copyright National University of Singapore. All Rights Reserved.

Hypertext Transfer Protocol (HTTP)

- HTTP methods
  - GET: Requests data from the server. Used to retrieve a webpage or data.
  - POST: Sends data to the server to create or update a resource. Often used in forms or API calls.
  - PUT: Updates a resource on the server.
  - DELETE: Deletes a resource from the server.
  - PATCH : Partially update an existing resource at certain field.
  - OPTIONS: Requests the communication options available on the target resource.

© Copyright National University of Singapore. All Rights Reserved.

Applications of HTTP in IoT

- Web Interfaces:
  - IoT devices host web servers where users can control devices, view sensor data, or configure settings through a web interface.
- API Communication:
  - IoT devices use HTTP to communicate with cloud-based APIs for data logging, remote control, or system monitoring.
- RESTful Web Services:
  - IoT devices use HTTP to interact with REST APIs, which allow clients to perform operations (such as reading or writing data) on a web service.

© Copyright National University of Singapore. All Rights Reserved.

Example of an HTTP Request and Response

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

```
HTTP/1.1 200 OK
Date: Mon, 18 Sep 2024 12:28:53 GMT
Server: Apache/2.4.1
Content-Type: text/html
Content-Length: 1234

<html>
<body>
<h1>Hello, World!</h1>
<p>This is a sample web page.</p>
</body>
</html>
```


© Copyright National University of Singapore. All Rights Reserved.

GET Request

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Specify the URL (example: weather API)
    String url = "http://jsonplaceholder.typicode.com/posts/1";
    http.begin(url);

    // Send the GET request
    int httpResponseCode = http.GET();

    // Check if the request was successful
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Response:");
      Serial.println(response);
    } else {
      Serial.printf("Error: %d\n", httpResponseCode);
    }

    // End the connection
    http.end();
  }
}
```

```
void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

POST Request

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Specify the URL (example: API endpoint)
    String url = "http://jsonplaceholder.typicode.com/posts";
    http.begin(url);

    // Specify content type header
    http.addHeader("Content-Type", "application/json");

    // JSON payload
    String jsonPayload = "{\"title\":\"foo\",\"body\":\"bar\",\"userId\":1}";

    // Send POST request
    int httpResponseCode = http.POST(jsonPayload);

    // Check if the request was successful
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Response:");
      Serial.println(response);
    } else {
      Serial.printf("Error: %d\n", httpResponseCode);
    }

    // End the connection
    http.end();
  }
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

PUT Request

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Specify the URL to update the resource
    String url = "http://jsonplaceholder.typicode.com/posts/1";
    http.begin(url);

    // Specify content type header
    http.addHeader("Content-Type", "application/json");

    // JSON payload
    String jsonPayload = "{\"title\":\"updated title\",\"body\":\"updated body\",\"userId\":1}";

    // Send PUT request
    int httpResponseCode = http.PUT(jsonPayload);

    // Check if the request was successful
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Response:");
      Serial.println(response);
    } else {
      Serial.printf("Error: %d\n", httpResponseCode);
    }

    // End the connection
    http.end();
  }
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

PATCH Request

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Specify the URL
    String url = "http://jsonplaceholder.typicode.com/posts/1";
    http.begin(url);

    // Specify content type header
    http.addHeader("Content-Type", "application/json");

    // JSON payload (only update the title)
    String jsonPayload = "{\"title\":\"partially updated title\"}";

// Send PATCH request
    int httpResponseCode = http.PATCH(jsonPayload);

    // Check if the request was successful
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Response:");
      Serial.println(response);
    } else {
      Serial.printf("Error: %d\n", httpResponseCode);
    }

    // End the connection
    http.end();
  }
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

DELETE Request

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;

    // Specify the URL to delete the resource
    String url = "http://jsonplaceholder.typicode.com/posts/1";
    http.begin(url);

    // Send DELETE request
    int httpResponseCode = http.sendRequest("DELETE");

    // Check if the request was successful
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println("Response:");
      Serial.println(response);
    } else {
      Serial.printf("Error: %d\n", httpResponseCode);
    }

    // End the connection
    http.end();
  }
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

Considerations for HTTP Operations

- Headers: Depending on the API, authentication tokens or other headers may need.

```
http.addHeader("header-name", "header-value")
```

- Timeouts: By default, the HTTPClient class has a timeout of around 5 seconds. Adjust it using http.setTimeout() if needed.
- HTTPS Requests: If the server uses HTTPS, use the WiFiClientSecure class instead of WiFiClient, and handle SSL certificates properly.

© Copyright National University of Singapore. All Rights Reserved.

Considerations for HTTP Operations

- Error Handling: Always check the HTTP response codes and handle potential errors like 404 (Not Found), 500 (Server Error), or connection timeouts.
- Memory Management: Be cautious with memory on the board, especially with large JSON responses or payloads. Use StaticJsonDocument for fixed-size JSON parsing and avoid dynamic memory allocation when possible.

© Copyright National University of Singapore. All Rights Reserved.

Synchronous connections

- Synchronous connection: Persistently listen to the port for a response before continuing with other tasks after sending a request to a server or device.
  - *HTTPClient.h* and *WebServer.h*
  - Simpler to implement.
  - Useful when the result of a task is immediately needed for the next step.
  - If the response takes time (e.g., slow server), the ESP32 will remain idle and unresponsive.
  - Not suitable for real-time applications, where timing is critical.

© Copyright National University of Singapore. All Rights Reserved.

Synchronous Client

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";
const char* serverName = "http://example.com/api/data";
// Replace with your server

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Make the synchronous HTTP request
  makeHttpRequest();
}

void loop() {
  // request was handled synchronously in setup
}
```

```
void makeHttpRequest() {
  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin(serverName);

    // This is a synchronous call - ESP32 waits for a response
    int httpResponseCode = http.GET();

    if (httpResponseCode > 0) {
      String payload = http.getString();
      Serial.println("HTTP Response code: " +
                                          String(httpResponseCode));
      Serial.println("Payload: " + payload);
    } else {
      Serial.println("Error on HTTP request");
    }

    http.end(); // Close connection
  } else {
    Serial.println("WiFi Disconnected");
  }
}
```


© Copyright National University of Singapore. All Rights Reserved.

Synchronous Server

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

// Replace with your network credentials
const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

// Create a WebServer object on port 80
WebServer server(80);

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

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

  // Wait until the ESP32 connects to Wi-Fi
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }

  Serial.println("Connected to WiFi");
  Serial.println(WiFi.localIP());

  // Define routes for the server
  server.on("/", handleRoot); // Root URL
  server.on("/status", handleStatus); // Status URL

  // Start the server
  server.begin();
  Serial.println("HTTP server started");
}

void loop() {
  // Handle incoming client requests synchronously
  server.handleClient();
}

// Function to handle the root URL "/"
void handleRoot() {
  Serial.println("Root request received");
  // Send a response to the client
  server.send(200, "text/plain", "Hello, this is a synchronous ESP32
server!");
}

// Function to handle the URL "/status"
void handleStatus() {
  Serial.println("Status request received");
  String message = "ESP32 Status: OK\n";
  message += "Free Heap: " + String(ESP.getFreeHeap()) + " bytes\n";
  message += "WiFi Signal Strength: " + String(WiFi.RSSI()) + " dBm\n";
  // Send the response
  server.send(200, "text/plain", message);
}
```


© Copyright National University of Singapore. All Rights Reserved.

HTTPClient library

- Provide interface to make HTTP requests from the ESP32 to a web server.
  - Send GET, POST, PUT, and DELETE requests over HTTP or HTTPS.
    - GET: Retrieve data from a web service or API.
    - POST: Send data to an API (e.g., sensor readings, form submissions).
    - PUT: Update existing data on a server.
    - DELETE: Remove data from a web service.
    - HTTPS: Securely interact with services using SSL/TLS.
  - Enable the ESP32 to interact with REST APIs, fetch web pages, upload data to cloud services, and more.

© Copyright National University of Singapore. All Rights Reserved.

HTTPClient library key APIs

- begin(url): Initializes the connection to a specified URL.

```
http.begin("http://example.com/api");
```

- begin(url, root_ca): Initializes a secure connection (HTTPS) to a specified URL, using a root certificate for SSL verification.

```
http.begin("https://example.com", root_ca);
```

- addHeader(header_name, header_value): Adds custom headers to the HTTP request.

```
http.addHeader("Content-Type", "application/json");
```

- GET(): Sends an HTTP GET request.

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

- POST(payload): Sends an HTTP POST request with a body payload.

```
int httpCode = http.POST("{\"key\":\"value\"}");
```

- PUT(payload): Sends an HTTP PUT request with a body payload.

```
int httpCode = http.PUT("{\"key\":\"value\"}");
```


© Copyright National University of Singapore. All Rights Reserved.

HTTPClient library key APIs

- DELETE(): Sends an HTTP DELETE request.

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

- getString(): Returns the response body as a string.

```
String payload = http.getString();
```

- getStream(): Returns the response body as a stream, useful for large files.

```
Stream& response = http.getStream();
```

- getSize(): Returns the size of the response payload.

```
int len = http.getSize();
```

- errorToString(httpCode): Converts an HTTP error code into a human-readable string.

```
Serial.println(http.errorToString(httpCode));
```

- end(): Ends the HTTP connection, freeing resources.

```
http.end();
```


© Copyright National University of Singapore. All Rights Reserved.

HTTPClient use cases

- Fetching data from a web API.
  - Fetch data from a REST API using HTTP GET to retrieve sensor data, weather information, or other external data sources.
- Sending sensor data to a cloud service.
  - Use an HTTP POST request to send data from device to a cloud service (like Google Cloud, AWS, or Thingspeak) for storage, monitoring, or further processing.

© Copyright National University of Singapore. All Rights Reserved.

HTTPClient use cases

- Uploading data to a web server.
  - Use HTTP POST or PUT to upload data to a web server, such as a file upload or submitting form data.
- Interacting with IoT platforms.
  - Many IoT platforms provide RESTful APIs. You can use HTTPClient to send and receive data from IoT services like Adafruit IO, Blynk, or Home Assistant.

© Copyright National University of Singapore. All Rights Reserved.

HTTPClient use cases

- Controlling web-connected devices.
  - You can send HTTP requests to control or interact with other devices over the web, such as triggering relays or actuating motors.
- Downloading files or images.
  - Use HTTP GET to download a file or image from a remote server, which can then be saved to SPIFFS or SD card on the ESP32.

© Copyright National University of Singapore. All Rights Reserved.

HTTP GET Request Example

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

const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";
const char* serverName = "http://jsonplaceholder.typicode.com/posts/1"; // Example API

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

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  HTTPClient http;

  // Specify the URL
  http.begin(serverName);

  // Send the HTTP GET request
  int httpCode = http.GET();

  // Check if the request was successful
  if (httpCode > 0) {
    // Get the response payload
    String payload = http.getString();
    Serial.println("Response code: " + String(httpCode));
    Serial.println("Response: " + payload);
  } else {
    Serial.println("Error on HTTP request");
  }

  http.end();  // Close connection
}
```

```
void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

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

const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";
const char* serverName = "http://jsonplaceholder.typicode.com/posts";  // Example API

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

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  HTTPClient http;

  // Specify the URL
  http.begin(serverName);

  // Specify the content type header
  http.addHeader("Content-Type", "application/json");

  // Prepare the payload (JSON)
  String jsonPayload = "{\"title\":\"foo\",\"body\":\"bar\",\"userId\":1}";

 // Send HTTP POST request
  int httpCode = http.POST(jsonPayload);

  // Check if the request was successful
  if (httpCode > 0) {
    String response = http.getString();
    Serial.println("Response code: " + String(httpCode));
    Serial.println("Response: " + response);
  } else {
    Serial.println("Error on HTTP request");
  }
    http.end();  // Close connection
}

void loop() {
  // Nothing here
}
```

HTTP POST Request Example (Sending JSON Data)

© Copyright National University of Singapore. All Rights Reserved.

HTTP PUT Request Example (Updating Data)

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

const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";
const char* serverName = "http://jsonplaceholder.typicode.com/posts/1";  // Example API

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

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  HTTPClient http;

  // Specify the URL
  http.begin(serverName);

  // Specify the content type header
  http.addHeader("Content-Type", "application/json");

  // Prepare the payload (JSON)
  String jsonPayload = "{\"id\":1,\"title\":\"foo\",\"body\":\"updated content\",\"userId\":1}";

  // Send HTTP PUT request
  int httpCode = http.PUT(jsonPayload);

  // Check if the request was successful
  if (httpCode > 0) {
    String response = http.getString();
    Serial.println("Response code: " + String(httpCode));
    Serial.println("Response: " + response);
  } else {
    Serial.println("Error on HTTP request");
  }

  http.end();  // Close connection
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

HTTPS Request Example (Secure Connection)

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

const char* ssid = "Your_SSID";
const char* password = "Your_PASSWORD";

// Root CA for the server's SSL certificate (example for jsonplaceholder.typicode.com)
const char* root_ca = \
"-----BEGIN CERTIFICATE-----\n" \
"MIIDdzCCAl+gAwIBAgIEbdsIrzANBgkqhkiG9w0BAQsFADBoMQswCQYDVQQGEwJV\n" \
...
"-----END CERTIFICATE-----";

const char* serverName = "https://jsonplaceholder.typicode.com/posts/1";  // Example HTTPS API

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

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  WiFiClientSecure client;

  // Use root certificate to verify SSL connection
  client.setCACert(root_ca);

  HTTPClient http;

  // Specify the URL
  http.begin(client, serverName);

  // Send HTTP GET request
  int httpCode = http.GET();

// Check if the request was successful
  if (httpCode > 0) {
    String payload = http.getString();
    Serial.println("Response code: " + String(httpCode));
    Serial.println("Response: " + payload);
  } else {
    Serial.println("Error on HTTPS request");
  }

  http.end();  // Close connection
}

void loop() {
  // Nothing here
}
```


© Copyright National University of Singapore. All Rights Reserved.

WebServer library

- Creating a webserver accessible from a local network (or even the internet) is one of the most common and useful applications for IoT hardware.
- IoT hardware programmed as a webserver.
  - Serve webpages, respond to API requests, and even allow remote control of devices or sensor data monitoring.
- Webserver APIs significantly enhance IoT projects
  - Remote control, monitoring, and interaction with sensors and actuators.
  - Serve static content, handle dynamic API requests, and even control hardware in real time via HTTP.

© Copyright National University of Singapore. All Rights Reserved.

WebServer library

- Key Concepts for Webserver APIs
  - Web Server: The ESP32 runs a lightweight web server that listens for HTTP requests.
  - APIs: These allow external clients (e.g., mobile apps, web browsers, or other IoT devices) to interact with the ESP32 by sending HTTP requests.
  - RESTful API: A common API architecture style, which typically uses HTTP methods (GET, POST, PUT, DELETE) to manage data.
  - Endpoints: Specific URLs that perform certain actions, such as */temperature* to get sensor data or */control* to activate a relay.

© Copyright National University of Singapore. All Rights Reserved.

WebServer library

- Common Use Cases for ESP32 Webserver APIs
  - Home Automation: Controls relays, lights, fans, or smart outlets through API endpoints accessible via web browsers or mobile apps.
  - Remote Sensor Monitoring: Expose sensor data (e.g., temperature, humidity, air quality) through API endpoints, allowing remote access to sensor readings.
  - Data Logging: Webserver collects data from various sensors and exposes it via an API for remote retrieval, or even sends data directly to cloud services.

© Copyright National University of Singapore. All Rights Reserved.

WebServer library

- Common Use Cases for ESP32 Webserver APIs
  - User Interfaces: Serve webpages that allow users to configure the device, adjust settings, or monitor the current status through a simple dashboard.
  - OTA Updates: Expose an API endpoint that allows for remote firmware updates to the ESP32 over the network.

© Copyright National University of Singapore. All Rights Reserved.

Basic ESP32 Web Server with a Simple API

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <DHT.h>

#define DHTPIN 4       // Pin where the DHT sensor is connected
#define DHTTYPE DHT11  // Change to DHT22 if you're using that sensor

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

DHT dht(DHTPIN, DHTTYPE);
AsyncWebServer server(80);

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

  // Initialize DHT sensor
  dht.begin();

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Define API endpoint for temperature and humidity data
  server.on("/api/temperature", HTTP_GET, [](AsyncWebServerRequest *request){
    float temperature = dht.readTemperature();
    float humidity = dht.readHumidity();

    // Check if readings are valid
    if (isnan(temperature) || isnan(humidity)) {
      request->send(500, "application/json", "{\"error\":\"Failed to read from DHT sensor\"}");
      return;
    }

    // Create JSON response
    String json = "{ \"temperature\": " + String(temperature) +
                  ", \"humidity\": " + String(humidity) + " }";

    // Send response
    request->send(200, "application/json", json);
  });

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

void loop() {
  // Nothing here, as we're using asynchronous web server
}
```


© Copyright National University of Singapore. All Rights Reserved.

Control GPIO Over API

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

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";
const int ledPin = 2;  // LED connected to GPIO 2

AsyncWebServer server(80);

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

  // Initialize LED pin
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Define API endpoint to turn the LED on
  server.on("/api/led/on", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, HIGH);  // Turn LED on
    request->send(200, "application/json", "{\"status\":\"LED is ON\"}");
  });

  // Define API endpoint to turn the LED off
  server.on("/api/led/off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);  // Turn LED off
    request->send(200, "application/json", "{\"status\":\"LED is OFF\"}");
  });

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

  void loop() {
    // Nothing here, as we're using asynchronous web server
  }
```


© Copyright National University of Singapore. All Rights Reserved.

Serving a Web Page with API Interactions

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <DHT.h>

#define DHTPIN 4       // Pin where the DHT sensor is connected
#define DHTTYPE DHT11  // Change to DHT22 if you're using that sensor

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

DHT dht(DHTPIN, DHTTYPE);
AsyncWebServer server(80);
const int ledPin = 2;

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  dht.begin();

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Serve web page
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/html", "<h1>ESP32 Web Server</h1><button onclick=\"fetch('/api/led/on').then(() => alert('LED ON'))\">Turn ON LED</button><button onclick=\"fetch('/api/led/off').then(() => alert('LED OFF'))\">Turn OFF LED</button>");
  });

  // Define API endpoints
  server.on("/api/led/on", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, HIGH);
    request->send(200, "application/json", "{\"status\":\"LED is ON\"}");
  });

  server.on("/api/led/off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(200, "application/json", "{\"status\":\"LED is OFF\"}");
  });

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

void loop() {
  // No code in loop as it's handled by the web server
}
```


© Copyright National University of Singapore. All Rights Reserved.

Asynchronous connections

- Asynchronous connection: Sends a request and continues executing other tasks while waiting for a response.
  - *AsyncTCP.h* and *ESPAsyncWebServer.h* in addition to *WiFi.h.*
  - Notify the calling function when the response is ready.
  - More efficient for handling multiple tasks, especially in real-time applications.
  - The program is not blocked waiting for responses.
  - Better for handling high-latency networks or slow servers.
  - More complex to implement (requires callbacks, event handlers, etc.).
  - Harder for debugging due to the nonlinear nature of execution.

© Copyright National University of Singapore. All Rights Reserved.

Asynchronous Server

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

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

AsyncWebServer server(80);

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Define an asynchronous web server route
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Hello from ESP32 Async Server!");
  });

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

void loop() {
  // Running other code while handling server requests asynchronously
}
```


© Copyright National University of Singapore. All Rights Reserved.

Asynchronous Client

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <AsyncHTTPClient.h>

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";
const char* serverName = "http://example.com/api/data";

AsyncHTTPClient httpClient;

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Define the callback for when the request is complete
  httpClient.onReadyStateChange([](void *arg, AsyncHTTPClient* client, int readyState){
    if (readyState == 4) {  // Response is complete
      Serial.println("Response: " + client->responseText());
    }
  });

  // Send the asynchronous HTTP request
  httpClient.open("GET", serverName);
  httpClient.send();
}

void loop() {
  // The program can continue to execute other tasks while waiting for the response
  Serial.println("ESP32 is free to do other tasks!");
  delay(5000);  // Emulate some other tasks
}
```


© Copyright National University of Singapore. All Rights Reserved.

AsyncTCP library (non-core)

- Library for creating non-blocking, asynchronous network communication, which are highly effective.
  - Asynchronous handling of HTTP requests.
  - Support for WebSockets, making real-time data communication possible.
  - HTTP GET, POST, PUT, and DELETE handling.

© Copyright National University of Singapore. All Rights Reserved.

AsyncTCP library (non-core)

- Key APIs for AsyncClient.
  - AsyncClient class represents a single TCP client that connects to a server and performs asynchronous TCP operations.

```
AsyncClient *client = AsyncClient();
```

  - connect(IPAddress ip, uint16_t port): Connects to a TCP server using the provided IP address and port.

```
client->connect(IPAddress(192, 168, 1, 100), 1234);
```

  - onConnect(AcConnectHandler): Registers a callback function for when the client successfully connects to the server.
  - onDisconnect(AcConnectHandler): Registers a callback function that is triggered when the client disconnects from the server..
  - onData(AcDataHandler): Sets a callback for when data is received from the server.

© Copyright National University of Singapore. All Rights Reserved.

AsyncTCP library (non-core)

- Key APIs for AsyncClient.
  - write(uint8_t *data, size_t len): Sends data to the connected server.

```
const char* message = "Hello, server!";
client->write((uint8_t*)message, strlen(message));
```

  - close(): Closes the connection.

```
client->close();
```

  - onError(AcErrorHandler): Registers a callback function to handle errors like connection failures.

```
client->onError([](void *arg, AsyncClient *client,
                                        err_t error) {
      Serial.printf("Error: %s\n", lwip_strerr(error));
      });
```


© Copyright National University of Singapore. All Rights Reserved.

AsyncTCP library (non-core)

- Key APIs for AsyncServer.
  - AsyncServer(uint16_t port): Creates a TCP server on the specified port.

```
AsyncServer *server = new AsyncServer(1234);
```

  - begin(): Starts the TCP server, making it ready to accept incoming client connections.

```
server->begin();
```

  - onClient(AcConnectHandler): Registers a callback for when a new client connects to the server.
  - onTimeout(AcTimeoutHandler): Handles the timeout of TCP connections if the server is unable to respond within a given time.
  - onData(AcDataHandler): Sets a callback for when data is received from the server.

© Copyright National University of Singapore. All Rights Reserved.

Common Use Cases of AsyncTCP

- Non-blocking Data Streaming
- IoT Gateways
- Real-time Control
- WebSocket or MQTT over TCP

© Copyright National University of Singapore. All Rights Reserved.

Connects to a remote TCP server and sends periodic messages to the server

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

const char* ssid = "yourSSID";
const char* password = "yourPASSWORD";

AsyncClient *client = new AsyncClient();

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  // Connect to Wi-Fi
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }

  Serial.println("Connected to WiFi!");

  // Connect to the TCP server
  client->onConnect([](void *arg, AsyncClient *client) {
    Serial.println("Connected to server!");
    client->write("Hello from ESP32!");
  });

  client->onData([](void *arg, AsyncClient *client, void *data, size_t len) {
    Serial.print("Received data: ");
    Serial.write((char*)data, len);
  });

  client->onDisconnect([](void *arg, AsyncClient *client) {
    Serial.println("Disconnected from server!");
  });

  client->connect(IPAddress(192, 168, 1, 100), 1234);  // Connect to TCP server
}
```

```
void loop() {
  // No need to implement anything here, the client is fully async
}
```


© Copyright National University of Singapore. All Rights Reserved.

TCP server listens for incoming connections

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

const char* ssid = "yourSSID";
const char* password = "yourPASSWORD";

AsyncServer *server = new AsyncServer(1234);  // Create server on port 1234

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  // Connect to Wi-Fi
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }

  Serial.println("Connected to WiFi!");

  // Start the TCP server
  server->onClient([](void *arg, AsyncClient *client) {
    Serial.println("New client connected!");

    client->onData([](void *arg, AsyncClient *client, void *data, size_t len) {
      Serial.print("Received data: ");
      Serial.write((char*)data, len);

      // Send response back to the client
      client->write("Hello from ESP32 server!");
    });

    client->onDisconnect([](void *arg, AsyncClient *client) {
      Serial.println("Client disconnected");
    });
  });

  server->begin();
}

void loop() {
  // The server runs asynchronously.
}
```


© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library

- Powerful libraries for handling asynchronous webserver functionalities on the ESP32 or ESP8266.
- Provide a non-blocking, event-driven framework for handling multiple client connections. – High performance and low latency.
- Offering WebSockets for real-time data, file serving capabilities, and the ability to control GPIOs or sensors via a web interface.

© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library

- Features of ESPAsyncServer
  - Non-blocking: It allows handling multiple client requests without blocking the main loop, which improves performance.
  - Event-driven: Each event (like a client request or connection) triggers a callback function, allowing efficient handling of asynchronous tasks.
  - WebSocket support: Built-in support for WebSocket communication.
  - File Serving: Easily serve files from SPIFFS, LittleFS, or SD cards.
  - SSL/TLS Support: For secure communication.

© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library key classes and objects

- **AsyncWebServer**: The main class for creating a web server.
- **AsyncWebServerRequest**: Represents an incoming HTTP request.
- **AsyncWebServerResponse**: Represents an HTTP response sent back to the client.
- **AsyncWebSocket**: Used for creating and handling WebSocket connections.
- **AsyncEventSource**: Used for handling Server-Sent Events (SSE).

© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library main APIs

- `AsyncWebServer server(PORT);`

Initializes the web server on the specified port (e.g., 80 for HTTP).

- `server.on("/path", HTTP_GET, handlerFunction);`

Handles incoming requests to a specific route (/path) for a GET request.

- `server.on("/path", HTTP_POST, handlerFunction);`

Similar to GET but for POST requests.

- `server.onNotFound(handlerFunction);`

Handles requests to undefined routes, returning a 404 error or custom message.

- `server.on("/ws", HTTP_GET, WebSocket handler);`

Allows creating real-time, bi-directional communication between client and server using WebSockets.

© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library main APIs

- `request->send(200, "text/plain", "Hello World!");`

Sends a plain text HTTP response with the status code (e.g., 200 for success).

- `request->send(SPIFFS, "/index.html", "text/html");`

Sends an HTML file from the SPIFFS (or LittleFS) filesystem.

- `server.onFileUpload([](AsyncWebServerRequest *request, const String& filename, size_t index, uint8_t *data, size_t len, bool final) {...});`

Handles file uploads asynchronously, useful for uploading large files without blocking the main loop.

- `server.serveStatic("/static", SPIFFS, "/static").setCacheControl("max-age=600");`

Serves static files (e.g., CSS, JavaScript, images) from the file system.

- `request->getParam("key")->value();`

Retrieves URL query parameters from an incoming request (e.g., /path?key=value).

© Copyright National University of Singapore. All Rights Reserved.

ESPAsyncWebServer library

- ESPAsyncServer use cases
  - Web-Based Configuration Portal: Host a web interface on the ESP32 for configuring Wi-Fi credentials, sensor thresholds, etc.
  - Real-Time Monitoring Dashboards: Display sensor data in real-time using WebSockets for continuous updates without refreshing the page.
  - IoT Device Control: Control GPIOs and actuators (e.g., turning on lights, controlling motors) from a web interface.
  - Remote Firmware Updates (OTA): Use a web interface to upload and apply firmware updates remotely.
  - Data Logging Server: Serve sensor data to multiple clients or send updates to a cloud service.

© Copyright National University of Singapore. All Rights Reserved.

Basic Web Server Example

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>

// Replace with your network credentials
const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

void setup() {
  // Serial port for debugging
  Serial.begin(115200);

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Serve the HTML page
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/html", "<h1>Hello, this is ESP32!</h1>");
  });

  // Handle GPIO control (e.g., toggle an LED)
  server.on("/toggleGPIO", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));  // Toggle the LED
    request->send(200, "text/plain", "Toggled GPIO!");
  });

  // Start server
  server.begin();
}

void loop() {
  // No need to handle client connections manually; handled
asynchronously
}
```


© Copyright National University of Singapore. All Rights Reserved.

WebSocket Example for Real-Time Data

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <WebSocketsServer.h>

const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

// Web server and WebSocket server
AsyncWebServer server(80);
WebSocketsServer webSocket(81);

// Handle incoming WebSocket messages
void handleWebSocketMessage(uint8_t num, WStype_t type, uint8_t * payload, size_t length) {
  if(type == WStype_TEXT) {
    Serial.printf("[%u] Received: %s\n", num, payload);
    webSocket.sendTXT(num, "Message received");  // Send response back
  }
}

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

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Serve WebSocket requests
  webSocket.begin();
  webSocket.onEvent(handleWebSocketMessage);

  // Serve the WebSocket HTML page
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/html", "<h1>WebSocket
Test</h1>");
  });

  server.begin();
}

void loop() {
  webSocket.loop();
}
```


© Copyright National University of Singapore. All Rights Reserved.

File Serving with SPIFFS

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <SPIFFS.h>

// Network credentials
const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASSWORD";

AsyncWebServer server(80);

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

  // Initialize SPIFFS
  if(!SPIFFS.begin(true)){
    Serial.println("SPIFFS Mount Failed");
    return;
  }

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Serve index.html
  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/index.html", "text/html");
  });

  // Serve additional files (CSS, JS, etc.)
  server.serveStatic("/css", SPIFFS, "/css/");
  server.serveStatic("/js", SPIFFS, "/js/");

  server.begin();
}

void loop(){
  // Nothing needed in the loop
}
```


© Copyright National University of Singapore. All Rights Reserved.

WebSocket

- **Full-Duplex**: Both client and server can send and receive messages independently at any time, making it ideal for real-time applications.
- **Persistent Connection**: The connection remains open until either the client or the server decides to close it.
- **Efficient**: Low overhead compared to HTTP because there's no need to repeatedly open and close connections for each message.
- **Applications**: Real-time applications like chat applications, online gaming, IoT device data streaming, financial market data, etc.

© Copyright National University of Singapore. All Rights Reserved.

WebSocket

- ESPAsyncWebServer and AsyncWebSocket libraries are commonly used to set up WebSocket servers.
  - Device can handle multiple WebSocket clients simultaneously without blocking the main loop.
- WebSocket use cases
  - Real-time sensor monitoring
  - Device control from the browser
  - Live data dashboards
  - Push notifications

© Copyright National University of Singapore. All Rights Reserved.

A simple WebSocket server running on the ESP32

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <AsyncTCP.h>

// WiFi credentials
const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

// WebSocket instance
AsyncWebServer server(80);
AsyncWebSocket ws("/ws");  // WebSocket server endpoint at /ws

// Callback to handle WebSocket events
void onWsEvent(AsyncWebSocket *server, AsyncWebSocketClient *client,
AwsEventType type, void *arg, uint8_t *data, size_t len) {
  if (type == WS_EVT_CONNECT) {
    Serial.println("WebSocket client connected");
    client->text("Hello from ESP32 WebSocket server!");
  } else if (type == WS_EVT_DISCONNECT) {
    Serial.println("WebSocket client disconnected");
  } else if (type == WS_EVT_DATA) {
    Serial.printf("Data received: %s\n", (char*)data);
    client->text("Message received: " + String((char*)data));
  }
}
```

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

  // Connect to WiFi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Setup WebSocket event handler
  ws.onEvent(onWsEvent);

  // Add WebSocket to the web server
  server.addHandler(&ws);

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

void loop() {
  // Cleanup disconnected clients
  ws.cleanupClients();
}
```


© Copyright National University of Singapore. All Rights Reserved.

Example of WebSocket Client (HTML/JavaScript)

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP32 WebSocket Demo</title>
  <script>
    var ws;

    function initWebSocket() {
      ws = new WebSocket('ws://' + window.location.hostname + '/ws');

      ws.onopen = function() {
        document.getElementById('status').innerHTML = "WebSocket connection established";
      };

      ws.onmessage = function(event) {
        document.getElementById('messages').innerHTML += '<br>' + event.data;
      };

      ws.onclose = function() {
        document.getElementById('status').innerHTML = "WebSocket connection closed";
      };
    }

    function sendMessage() {
      var message = document.getElementById('msg').value;
      ws.send(message);
    }
  </script>
</head>
<body onload="initWebSocket();">
  <h1>ESP32 WebSocket Demo</h1>
  <p id="status">Connecting...</p>
  <div>
    <input type="text" id="msg" placeholder="Type a message">
    <button onclick="sendMessage()">Send Message</button>
  </div>
  <div id="messages">
    <h2>Messages:</h2>
  </div>
</body>
</html>
```


© Copyright National University of Singapore. All Rights Reserved.

Real-Time Sensor Data Monitoring

```
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <AsyncTCP.h>

const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

AsyncWebServer server(80);
AsyncWebSocket ws("/ws");

void onWsEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
               void *arg, uint8_t *data, size_t len) {
  if (type == WS_EVT_CONNECT) {
    Serial.println("WebSocket client connected");
  } else if (type == WS_EVT_DISCONNECT) {
    Serial.println("WebSocket client disconnected");
  }
}

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  ws.onEvent(onWsEvent);
  server.addHandler(&ws);
  server.begin();
}
```

```
void loop() {
  // Simulating sensor data (replace this with actual sensor reading)
  int sensorValue = analogRead(34);  // Example for ESP32's analog pin

  // Sending sensor data to all WebSocket clients
  ws.textAll(String(sensorValue));

  // Cleanup WebSocket clients
  ws.cleanupClients();

  delay(1000); // Send data every 1 second
}
```


© Copyright National University of Singapore. All Rights Reserved.

Web Client for Real-Time Sensor Data Monitoring

```
<!DOCTYPE html>
<html>
<head>
  <title>Sensor Data</title>
  <script>
    var ws;

    function initWebSocket() {
      ws = new WebSocket('ws://' + window.location.hostname + '/ws');
      ws.onmessage = function(event) {
        document.getElementById('sensor').innerHTML = "Sensor Value: " + event.data;
      };
    }
  </script>
</head>
<body onload="initWebSocket();">
  <h1>Sensor Monitoring</h1>
  <p id="sensor">Waiting for data...</p>
</body>
</html>
```


© Copyright National University of Singapore. All Rights Reserved.

Representational State Transfer (REST) API

- **Stateless**: Each request from the client contains all the information needed by the server to fulfill the request.
- **Uses HTTP Methods**: Typically uses HTTP methods like GET (read), POST (create), PUT (update), DELETE (delete) to interact with resources.
- **Client-Server Model**: The client sends requests to the server, which performs operations on resources and returns a response.
- **Applications**: Web services, IoT backends, CRUD (Create, Read, Update, Delete) operations on databases.

© Copyright National University of Singapore. All Rights Reserved.

REST API

- Different systems can **communicate** over HTTP/HTTPS using HTTP methods like GET, POST, PUT, DELETE, etc.
- Server-client model -- REST API server/ REST API client.
- Platform independent: browsers, mobile apps or IoT device if their communication is defined based on HTTP standard.
- Cross-platforms: cloud, other ESP32 devices, home automation.
- It can be expanded with more endpoints and devices.

© Copyright National University of Singapore. All Rights Reserved.

REST API methods

- GET: Retrieve information (e.g., sensor data, device status).
- POST: Send data to a remote server (e.g., log sensor data, trigger an action).
- PUT: Update existing data on the server (e.g., change configuration).
- DELETE: Remove data from the server.

© Copyright National University of Singapore. All Rights Reserved.

REST API use cases

- Control Devices Remotely: Control ESP32-connected devices like lights, motors, or sensors from a mobile app or web interface.
- Retrieve Sensor Data: Expose sensor data over a REST API, which can be accessed by clients like mobile apps or cloud services for real-time data processing.
- Data Logging to Cloud: Act as a REST API client, sending sensor data to cloud platforms like AWS, Google Cloud, or ThingSpeak.
- Integration with Home Automation Platforms: Integrate with home automation platforms such as Node-RED.

© Copyright National University of Singapore. All Rights Reserved.

ESP32 REST API Server for Controlling an LED

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

// Wi-Fi credentials
const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

// Initialize WebServer on port 80
AsyncWebServer server(80);

int ledPin = 2; // GPIO Pin for the LED

void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Route for turning the LED ON
  server.on("/led/on", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, HIGH);   // Turn on LED
    request->send(200, "text/plain", "LED is ON");
  });

  // Route for turning the LED OFF
  server.on("/led/off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);    // Turn off LED
    request->send(200, "text/plain", "LED is OFF");
  });

  // Route to get LED status
    server.on("/led/status", HTTP_GET, [](AsyncWebServerRequest *request){
      String status = digitalRead(ledPin) ? "ON" : "OFF";
      request->send(200, "text/plain", "LED is " + status);
    });

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

void loop() {
    // Nothing required in the loop since we're using AsyncWebServer
}
```


© Copyright National University of Singapore. All Rights Reserved.

Sending Data to a REST API using POST

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

const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

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

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Simulated sensor value
  int sensorValue = analogRead(34);  // Replace with actual sensor read

  if(WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin("http://jsonplaceholder.typicode.com/posts"); // Example API URL
    http.addHeader("Content-Type", "application/json"); // Specify content type

    // JSON payload
    String payload = "{\"sensor\":\"temperature\", \"value\":" + String(sensorValue) + "}";

    // Send HTTP POST request
    int httpResponseCode = http.POST(payload);
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println(httpResponseCode); // Print HTTP response code
      Serial.println(response);         // Print server response
    } else {
      Serial.println("Error on sending POST");
    }
    http.end(); // Free resources
  }
}

void loop() {
  // Nothing required in the loop for this example
}
```


© Copyright National University of Singapore. All Rights Reserved.

Fetching Data from a REST API using GET

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

const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

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

  // Connect to Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (WiFi.status() == WL_CONNECTED) {
    HTTPClient http;
    http.begin("http://jsonplaceholder.typicode.com/posts/1"); // Example API URL

    int httpResponseCode = http.GET(); // Send HTTP GET request
    if (httpResponseCode > 0) {
      String response = http.getString();
      Serial.println(httpResponseCode); // Print response code
      Serial.println(response);         // Print response payload
    } else {
      Serial.println("Error on HTTP request");
    }
    http.end(); // Free resources
  }
}

void loop() {
  // Nothing required in the loop for this example
}
```


© Copyright National University of Singapore. All Rights Reserved.

REST API for Sensor Data Logging

```
server.on("/sensor", HTTP_GET, [](AsyncWebServerRequest *request){
  int sensorValue = analogRead(34); // Example for analog sensor
  String jsonResponse = "{\"sensor_value\": " + String(sensorValue) + "}";
  request->send(200, "application/json", jsonResponse);
});
```

Integrating ESP32 with Cloud Services

```
// ThingSpeak example POST request
http.begin("https://api.thingspeak.com/update?api_key=YOUR_API_KEY&field1=" + String(sensorValue));
```


© Copyright National University of Singapore. All Rights Reserved.