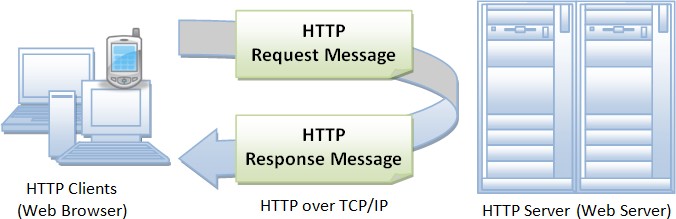
|  |
| --- |
| **TITLE: ESP32 USING HTTP PROTOCOL** |
| **Objectives:**  Upon completion of this practical work, you should be able to:   * Write codes to establish a connection to the internet and display data from sensor to web browser successfully.   **Equipments:**   1. ESP32 Module Board 2. Jumper wire , 2 x 330 Ohm Resistor, 2 x LED, Breadboard. 3. PC installed with Arduino IDE 4. Internet connection (Wireless Router / Access Point dedicated to the Laboratory) 5. DS18B20 Temperature Sensor, 5V DC Motor   **Theory:**  **What is a Web server and how it works?**  Web server is a place which stores, processes and delivers web pages to Web clients. Web client is nothing but a web browser on our laptops and smartphones. The communication between client and server takes place using a special protocol called Hypertext Transfer Protocol (HTTP).    In this protocol, a client initiates communication by making a request for a specific web page using HTTP and the server responds with the content of that web page or an error message if unable to do so (like famous 404 Error). Pages delivered by a server are mostly HTML documents.  **HyperText Transfer Protocol (HTTP)**  HTTP (Hypertext Transfer Protocol) is perhaps the most popular application protocol used in the Internet (or The WEB).   * HTTP is an *asymmetric request-response client-server* protocol as illustrated. An HTTP client sends a request message to an HTTP server. The server, in turn, returns a response message. In other words, HTTP is a *pull protocol*, the client *pulls* information from the server (instead of   server *pushes*information down to the client). |



* HTTP is a stateless protocol. In other words, the current request does not know what has been done in the previous requests.
* HTTP permits negotiating of data type and representation, so as to allow systems to be built independently of the data being transferred.

## The GET Method

GET is used to request data from a specified resource. GET is one of the most common HTTP methods.

Note that the query string (name/value pairs) is sent in the URL of a GET request:

## /test/demo\_form.php?name1=value1&name2=value2

Some other notes on GET requests:

* + GET requests can be cached
  + GET requests remain in the browser history
  + GET requests can be bookmarked
  + GET requests should never be used when dealing with sensitive data
  + GET requests have length restrictions
  + GET requests is only used to request data (not modify)

## The POST Method

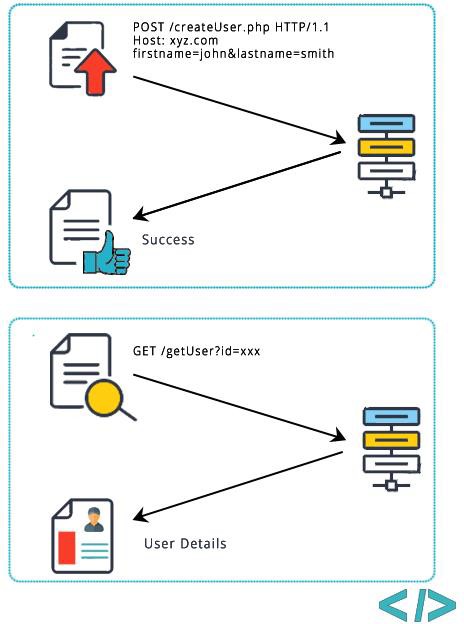
**POST is used to send data to a server to create/update a resource.**

The data sent to the server with POST is stored in the request body of the HTTP request: POST /test/demo\_form.php HTTP/1.1

Host: w3schools.com name1=value1&name2=value2

## POST is one of the most common HTTP methods.

Some other notes on POST requests:



* + POST requests are never cached
  + POST requests do not remain in the browser history
  + POST requests cannot be bookmarked
  + POST requests have no restrictions on data length

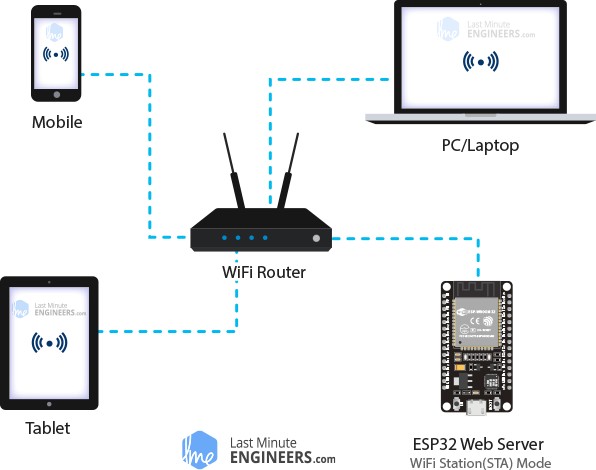
**ESP32 Operating Modes**

One of the greatest features ESP32 provides is that it cannot only connect to an existing WiFi network and act as a Web Server, but it can also set up a network of its own, allowing other devices to connect directly to it and access web pages. This is possible because ESP32 can operate in three different modes:

* + 1. Station mode,
    2. Soft Access Point mode,
    3. and both at the same time.

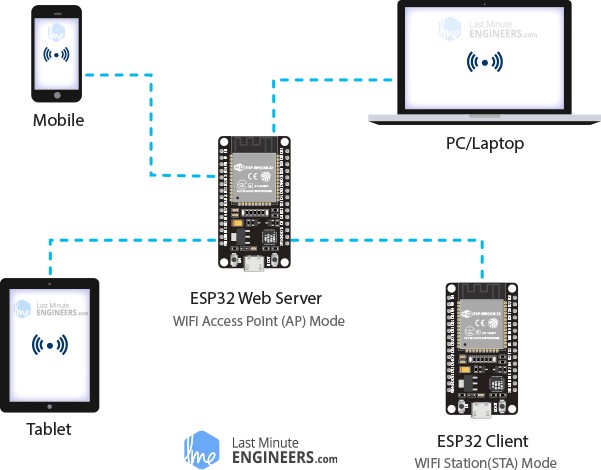
This provides possibility of building [mesh networks](https://en.wikipedia.org/wiki/Mesh_networking).

**Station (STA) Mode**



The ESP32 that connects to an existing WiFi network (one created by your wireless router) is called Station (STA)

In STA mode ESP32 gets IP from wireless router to which it is connected. With this IP address, it can set up a web server and deliver web pages to all connected devices under existing WiFi network.



**Soft Access Point (AP) Mode**

The ESP32 that creates its own WiFi network and acts as a hub (Just like WiFi router) for one or more stations is called Access Point (AP). Unlike WiFi router, it does not have interface to a wired network. So, such mode of operation is called Soft Access Point (soft-AP). Also the maximum number of stations that can connect to it is limited to five.

In AP mode ESP32 creates a new WiFi network and sets SSID (Name of the network) and IP address to it. With this IP address, it can deliver web pages to all connected devices under its own network.

Further reading available at: <https://lastminuteengineers.com/creating-esp32-web-server-arduino-ide/>

## DHT11 Temperature and Humidity Sensor

### Pin Identification and Configuration:

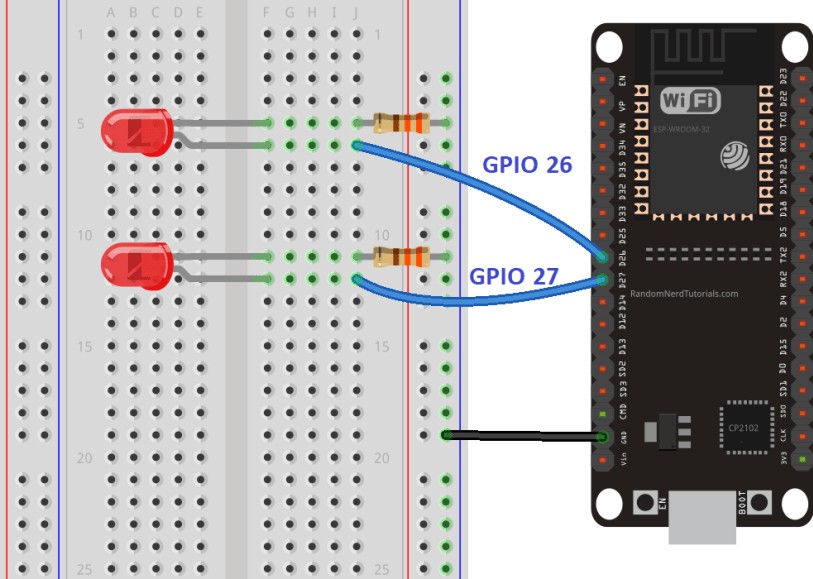
|  |  |  |
| --- | --- | --- |
| **No:** | **Pin Name** | **Description** |
| **For DHT11 Sensor** | | |
| 1 | Vcc | Power supply 3.5V to 5.5V |
| 2 | Data | Outputs both Temperature and Humidity through serial Data |
| 3 | NC | No Connection and hence not used |
| 4 | Ground | Connected to the ground of the circuit |
| **For DHT11 Sensor module** | | |
| 1 | Vcc | Power supply 3.5V to 5.5V |
| 2 | Data | Outputs both Temperature and Humidity through serial Data |
| 3 | Ground | Connected to the ground of the circuit |

### DHT11 Specifications:

* Operating Voltage: 3.5V to 5.5V
* Operating current: 0.3mA (measuring) 60uA (standby)
* Output: Serial data
* Temperature Range: 0°C to 50°C
* Humidity Range: 20% to 90%
* Resolution: Temperature and Humidity both are 16-bit
* Accuracy: ±1°C and ±1%



## Procedures:



**A. Controlling LED using web server**

1. **The circuit**

Connect two LEDs to the ESP32 as shown in the following schematic diagram – one LED connected to GPIO 26, and the other to GPIO 27.

## The source code Step by Step

1. Importing Libraries.

// Load Wi-Fi library #include <WiFi.h>

you need to insert your ssid and password in the following lines inside the double quotes.

// Replace with your network credentials const char\* ssid = "Your SSID";

const char\* password = "Your PASSWORD";

Then, you set your web server to port 80.

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

The following line creates a variable to store the header of the HTTP request:

// Variable to store the HTTP request String header;

Next, you create auxiliar variables to store the current state of your outputs. If you want to add more outputs and save its state, you need to create more variables.

// Auxiliar variables to store the current output state String output26State = "off";

String output27State = "off";

You also need to assign a GPIO to each of your outputs. Here we are using **GPIO 26** and **GPIO 27**. You can use any other suitable GPIOs.

// Assign output variables to GPIO pins const int output26 = 26;

const int output27 = 27;

### setup() condition

Now, let’s go into the setup(). First, we start a serial communication at a baud rate of 115200 for debugging purposes.

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

You also define your GPIOs as OUTPUTs and set them to LOW.

// Initialize the output variables as outputs pinMode(output26, OUTPUT);

pinMode(output27, OUTPUT);

// Set outputs to LOW digitalWrite(output26, LOW); digitalWrite(output27, LOW);

The following lines begin the Wi-Fi connection with WiFi.begin(ssid, password), wait for a successful connection and print the ESP IP address in the Serial Monitor.

// Connect to Wi-Fi network with SSID and password Serial.print("Connecting to "); Serial.println(ssid);

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) { delay(500);

Serial.print(".");

}

// Print local IP address and start web server Serial.println("");

Serial.println("WiFi connected."); Serial.println("IP address: "); Serial.println(WiFi.localIP()); server.begin();

}

### loop() condition

In the loop() we program what happens when a new client establishes a connection with the web server.

The ESP32 is always listening for incoming clients with the following line:

void loop(){

WiFiClient client = server.available(); // Listen for incoming clients

When a request is received from a client, we’ll save the incoming data. The while loop that follows will be

running as long as the client stays connected. We don’t recommend changing the following part of the code unless you know exactly what you are doing.

if (client) { // If a new client connects,

Serial.println("New Client."); // print a message out in the serial port String currentLine = ""; // make a String to hold incoming data from the client while (client.connected()) { // loop while the client's connected

if (client.available()) { // if there's bytes to read from the client, char c = client.read(); // read a byte, then

Serial.write(c); // print it out the serial monitor header += c;

if (c == '\n') { // if the byte is a newline character

// if the current line is blank, you got two newline characters in a row.

// that's the end of the client HTTP request, so send a response:

if (currentLine.length() == 0) {

// HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)

// and a content-type so the client knows what's coming, then a blank line: client.println("HTTP/1.1 200 OK");

client.println("Content-type:text/html"); client.println("Connection: close"); client.println();

The next section of if and else statements checks which button was pressed in your web page, and

controls the outputs accordingly. As we’ve seen previously, we make a request on different URLs depending on the button pressed.

// turns the GPIOs on and off

if (header.indexOf("GET /26/on") >= 0) { Serial.println("GPIO 26 on"); output26State = "on"; digitalWrite(output26, HIGH);

} else if (header.indexOf("GET /26/off") >= 0) { Serial.println("GPIO 26 off");

output26State = "off"; digitalWrite(output26, LOW);

} else if (header.indexOf("GET /27/on") >= 0) { Serial.println("GPIO 27 on");

output27State = "on"; digitalWrite(output27, HIGH);

} else if (header.indexOf("GET /27/off") >= 0) { Serial.println("GPIO 27 off");

output27State = "off"; digitalWrite(output27, LOW);

}

For example, if you’ve press the GPIO 26 ON button, the ESP32 receives a request on the **/26/ON URL** (we can see that that information on the HTTP header on the Serial Monitor). So, we can check if

the header contains the expression **GET /26/on**. If it contains, we change the output26state variable to ON, and the ESP32 turns the LED on.

This works similarly for the other buttons. So, if you want to add more outputs, you should modify this part of the code to include them.

### Displaying the HTML web page

The next thing you need to do, is creating the web page. The ESP32 will be sending a response to your browser with some HTML code to build the web page.

The web page is sent to the client using this expressing client.println(). You should enter what you want to send to the client as an argument.

The first thing we should send is always the following line, that indicates that we are sending HTML.

// Display the HTML web page client.println("<!DOCTYPE html><html>");

Then, the following line makes the web page responsive in any web browser.

client.println("<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">");

And the following is used to prevent requests on the favicon. – You don’t need to worry about this line.

client.println("<link rel=\"icon\" href=\"data:,\">");

### Styling the Web Page

Next, we have some CSS text to style the buttons and the web page appearance. We choose the Helvetica font, define the content to be displayed as a block and aligned at the center.

// CSS to style the on/off buttons

// Feel free to change the background-color and font-size attributes to fit your // preferences

client.println("<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align: center;}");

We style our buttons with the #4CAF50 color, without border, text in white color, and with this padding: 16px 40px. We also set the text-decoration to none, define the font size, the margin, and the cursor to a pointer.

client.println(".button { background-color: #4CAF50; border: none; color: white; padding: 16px 40px;");

client.println("text-decoration: none; font-size: 30px; margin: 2px; cursor: pointer;}");

We also define the style for a second button, with all the properties of the button we’ve defined earlier, but with a different color. This will be the style for the off button.

client.println(".button2 {background-color: #555555;}</style></head>");

### Setting the Web Page First Heading

In the next line you can set the first heading of your web page. Here we have “**ESP32 Web Server**”, but you can change this text to whatever you like.

// Web Page Heading client.println("<h1>ESP32 Web Server</h1>");

// The HTTP response ends with another blank line

client.println();

// Break out of the while loop break;

} else { // if you got a newline, then clear currentLine currentLine = "";

}

} else if (c != '\r') {

return character,

currentLine += c;

}

}

}

// if you got anything else but a carriage

// add it to the end of the currentLine

### Displaying the Buttons and Corresponding State

Then, you write a paragraph to display the GPIO 26 current state. As you can see we use

the output26State variable, so that the state updates instantly when this variable changes.

// Display current state, and ON/OFF buttons for GPIO 26 client.println("<p>GPIO 26 - State " + output26State + "</p>");

Then, we display the on or the off button, depending on the current state of the GPIO. If the current state of the GPIO is off, we show the ON button, if not, we display the OFF button.

// If the output26State is off, it displays the ON button if (output26State=="off") {

client.println("<p><a href=\"/26/on\"><button class=\"button\">ON</button></a></p>");

} else {

client.println("<p><a href=\"/26/off\"><button class=\"button button2\">OFF</button></a></p>");

}

We use the same procedure for GPIO 27.

// Display current state, and ON/OFF buttons for GPIO 27 client.println("<p>GPIO 27 - State " + output27State + "</p>");

// If the output27State is off, it displays the ON button if (output27State=="off") {

client.println("<p><a href=\"/27/on\"><buttonclass=\"button\">ON</button></a></p>");

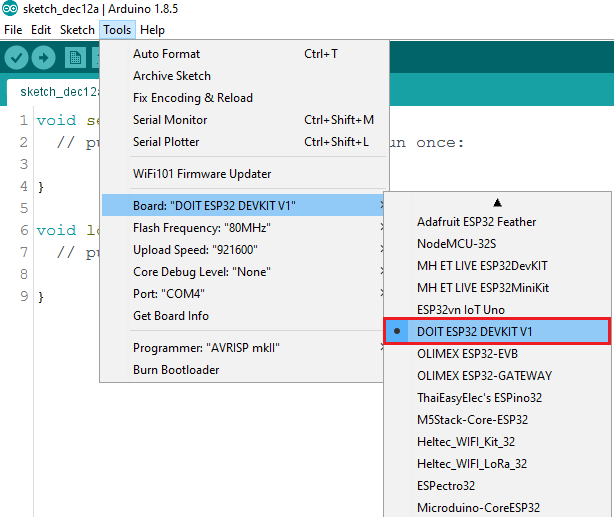
} else {

client.println("<p><a href=\"/27/off\"><button class=\"button button2\">OFF</button></a></p>");

}

client.println("</body></html>");

Then, The HTTP response ends with another blank line



### Closing the Connection

Finally, when the response ends, we clear the header variable, and stop the connection with the client with client.stop().

// Clear the header variable header = "";

// Close the connection client.stop();

Serial.println("Client disconnected."); Serial.println("");

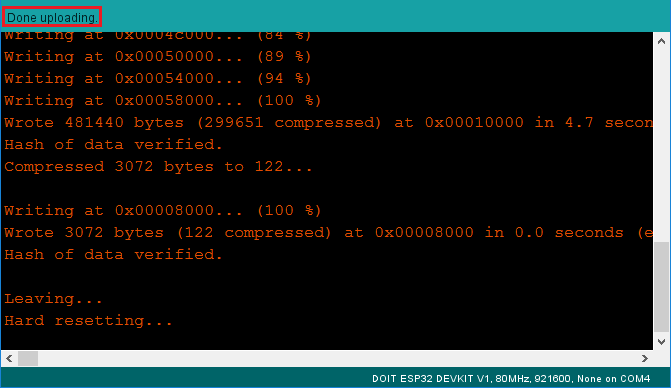
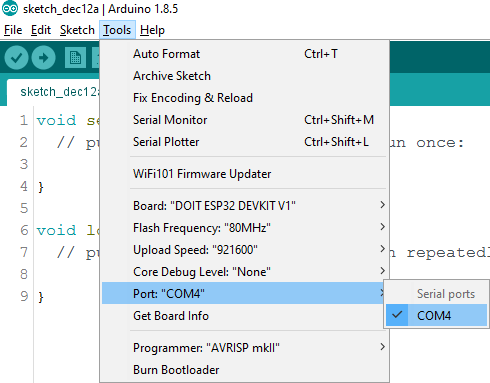
}

}

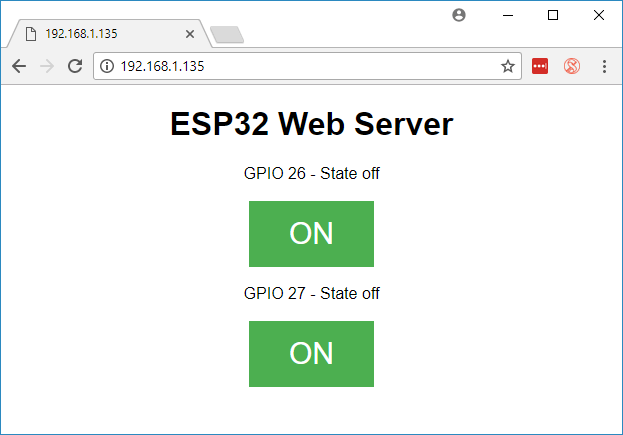
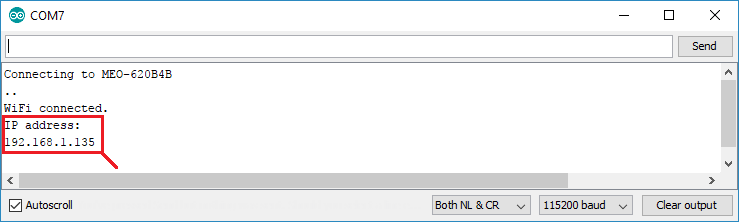
1. **Uploading the Code**

Now, you can upload the code and and the web server will work straight away. Follow the next steps to upload code to the ESP32:

1. Plug your ESP32 board in your computer;
2. In the Arduino IDE select your board in **Tools** > **Board** (in our case we’re using the ESP32 DEVKIT DOIT board);



1. Select the COM port in **Tools** > **Port**.
2. Press the **Upload** button in the Arduino IDE and wait a few seconds while the code compiles and uploads to your board.
3. Wait for the “**Done uploading**” message.



1. **Finding the ESP32 IP Address**

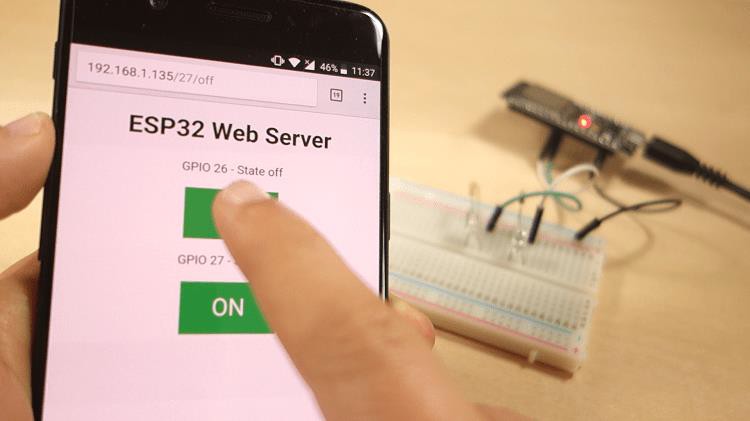
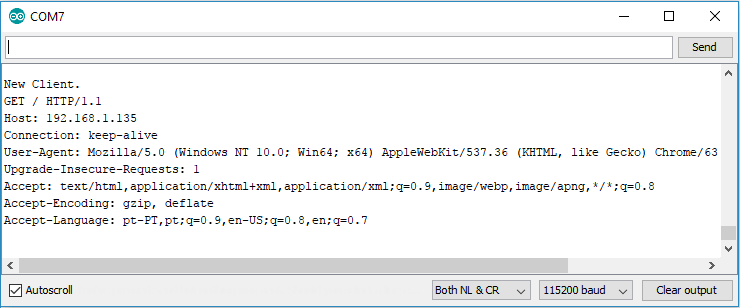
After uploading the code, open the Serial Monitor at a baud rate of 115200.

Press the ESP32 EN button (reset). The ESP32 connects to Wi-Fi, and outputs the ESP IP address on the Serial Monitor. **Copy that IP address**, because you **need it to access the ESP32 web server**.

1. **Accessing the Web Server**

To access the web server, open your browser, paste the ESP32 IP address, and you’ll see the following page. In our case it is **192.168.1.135**. (**depend on your IP stated in Serial Monitor in Step 4)**

If you take a look at the Serial Monitor, you can see what’s happening on the background. The ESP receives an HTTP request from a new client (in this case, your browser).

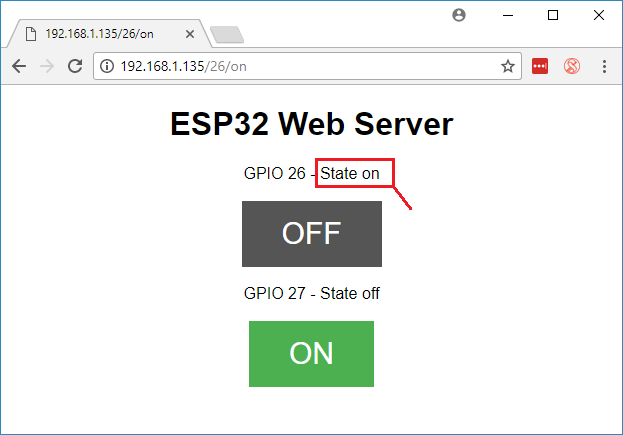
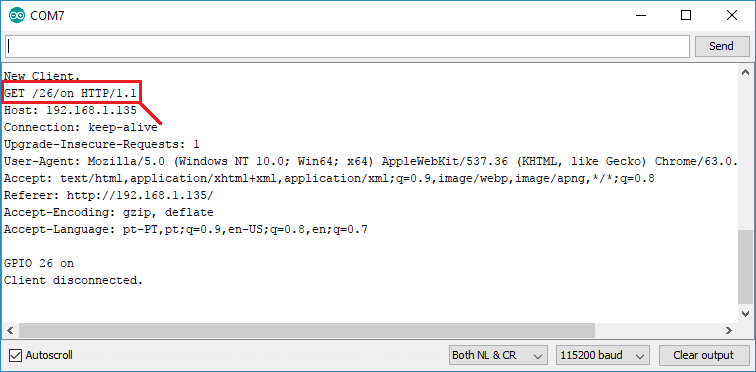


You can also see other information about the HTTP request.

1. **Testing the Web Server**

Now you can test if your web server is working properly using your phone. Make sure your phone using same network as your ESP32 Board. Click the buttons to control the LEDs.

At the same time, you can take a look at the Serial Monitor to see what’s going on in the background. For example, when you click the button to turn GPIO 26 ON, ESP32 receives a request on the **/26/on** URL.



When the ESP32 receives that request, it turns the LED attached to GPIO 26 **ON** and updates its state on the web page.

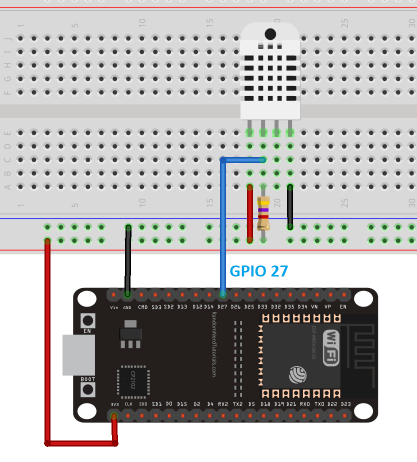
The button for GPIO 27 works in a similar way. Test that it is working properly.

- Capture the output from serial monitor and web browser for the report.

**B. ESP32 DHT11 TEMPERATURE SENSOR WEB SERVER**

## The circuit

Connect the DHT11 temperature and humidity sensor according to the following schematic diagram. In this example, the DHT11 data pin is connected to GPIO 27.



Note: If you’re using a DHT11 sensor module instead, it will only have three pins. Connect the jumper wires according to the labels provided on the DHT11 module. The module already comes together with an internal pull up resistor so there is no need to add one to the circuit.

1. **Installing the DHT Sensor Library**

To interface with the DHT11 sensor, you need to install the [DHT Sensor Library](https://github.com/adafruit/DHT-sensor-library) . Follow the next steps to install those libraries.

1. Download the .zip DHT sensor library.
2. Unzip the DHT-sensor-library-master.zip folder.
3. Rename DHT-sensor-library-master to DHT\_sensor and move it to your Arduino IDE installation libraries folder. Re-launch your Arduino IDE.
4. **Installing the Adafruit Unified Sensor Driver**

You also need to install the [Adafruit Unified Sensor Driver library](https://github.com/adafruit/Adafruit_Sensor" \t "_blank) to work with the DHT

sensor. Follow the next steps to install the library.

1. [Click here to download the Adafruit Unified Sensor library](https://github.com/adafruit/Adafruit_Sensor/archive/master.zip). You should have a .zip folder in your *Downloads*folder
2. Unzip the .zip folder and you should get *Adafruit\_sensor-master* folder
3. Rename your folder from  to *Adafruit\_sensor*
4. Move the *Adafruit\_sensor* folder to your Arduino IDE installation libraries folder
5. Finally, re-open your Arduino IDE
6. **Installing the ESPAsynWebServer Library**

Follow the next steps to install the [ESPAsyncWebServer](https://github.com/me-no-dev/ESPAsyncWebServer" \t "_blank) library:

1. [Click here to download the ESPAsyncWebServer library](https://github.com/me-no-dev/ESPAsyncWebServer/archive/master.zip). You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get *ESPAsyncWebServer-master* folder
3. Rename your folder from  to *ESPAsyncWebServer*
4. Move the *ESPAsyncWebServer*folder to your Arduino IDE installation libraries folder
5. **Installing the Async TCP Library for ESP32**

The [ESPAsyncWebServer](https://github.com/me-no-dev/ESPAsyncWebServer" \t "_blank) library requires the [AsyncTCP](https://github.com/me-no-dev/AsyncTCP" \t "_blank) library to work. Follow the next steps to install that library:

1. [Click here to download the AsyncTCP library](https://github.com/me-no-dev/AsyncTCP/archive/master.zip). You should have a .zip folder in your Downloads folder
2. Unzip the .zip folder and you should get *AsyncTCP-master* folder
3. Rename your folder from to *AsyncTCP*
4. Move the *AsyncTCP*folder to your Arduino IDE installation libraries folder
5. Finally, re-open your Arduino IDE
6. **The Code**

Open your Arduino and copy the following raw code from [GitHub](https://raw.githubusercontent.com/RuiSantosdotme/ESP32-Course/master/code/WiFi_Web_Server_DHT/WiFi_Web_Server_DHT.ino).

Insert your network credentials in the following variables and the code will work straight away.

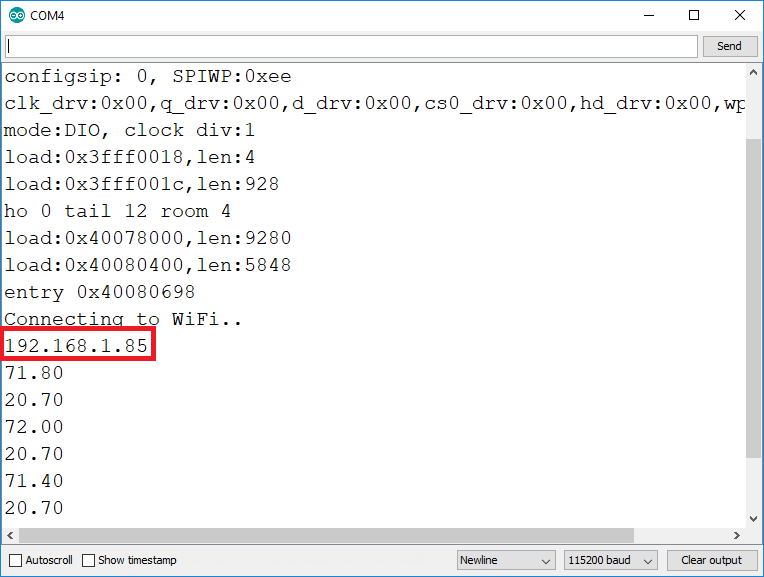
const char\* ssid = "REPLACE\_WITH\_YOUR\_SSID";

const char\* password = "REPLACE\_WITH\_YOUR\_PASSWORD";

Make sure you have the right settings for board and COM port. Upload the code to your ESP32.

After uploading, open the Serial Monitor at a baud rate of 115200. Press the ESP32 reset button. The ESP32 IP address should be printed in the serial monitor.

* 1. **The serial monitor**



Open a browser and type the ESP32 IP address. Your web server should display the latest sensor readings.

* 1. **The Web Server**

Notice that the temperature and humidity readings are updated automatically without the need to refresh the web page.



**C. PROBLEM BASED LEARNING**

En Fahmi Room is very hot. He bought a fan to solve the problem. However, he is not satisfied with the fan system. The fan needs to be ON and OFF manually. Help En Fahmi build a system to control the fan using Web Browser with the condition:

**If temperature >28, led is ON, fan is ON**

**If temperature <28, led is OFF, fan is OFF**

* Sketch the schematic using fritzing and write the code based on **PART A and PART B.**
* Create a table for input and output Pin Connection for the project. Refer to Practical Work 2
* Use 5v DC Motor for the fan.
* Capture the output from serial monitor and web browser for the report.

**Discussions:**

1. Based on the practical,what is the funtion of HTTP? Give an example coding according to this practical.

# (2marks)

1. Describe HTTP method use for this practical.

# (2marks)

1. Differentiate POST and GET Method in HTTP that use in this practical work

# (2marks)

## Conclusion:

Write your conclusion on this practical work.

## Rubric Practical Work 3: refer to the practical work rubric attached.