ESP8266 - HTTP RESPONSE STATUS CODE MONITOR

Prepared in partial fulfilment of the
Requirements of the
Study Project EEE F376

Submitted to

Devesh Samaiya
Assistant Professor II
Department of Electronics & Electrical Engineering

By

Jared Dominic Fernandez **2017B3A30588P**

Birla Institute of Technology and Science
Pilani, Rajasthan-333031

Abstract

The Project presents a design and prototype for a HTTP Response Status Code monitor using an ESP8266. The User is alerted by mail if there is an issue in the HTPP response code in any of the URLs. The user can configure the list of URLs and the email address from a password protected local webpage. The system is a very useful set and forget automation technique to monitor the given URLs as long as the device remains in the Wi-Fi network Range. The Arduino IDE was used to program and flash the ESP.

Table of Contents

Introduction

A HTTP Response Status Code is a numerical code that is part of the response returned by the server when a client calls a URL. The Project aims to create an automated system that monitors the HTTP Response Status Code. This is done on the ESP8266 with the help of Arduino IDE. It makes use of the default ESP8266 libraries along with the ESP Async Web Server library and the Serial Peripheral Interface Flash File System (SPIFFS) of the ESP for this purpose. In this report we will look into how this works with the explanation of the code. We will also discuss the observations, highlight the shortcomings, and propose possible improvements wherever possible.

HTTP Response Status Code

The Status-Code element in a server response, is a 3-digit integer where the first digit of the Status-Code defines the class of response and the last two digits do not have any categorization role.[1] It indicates whether a specific HTTP Request has been successfully completed or not. There are 5 different response classes:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)
- Client errors (400–499)
- Server errors (500–599)

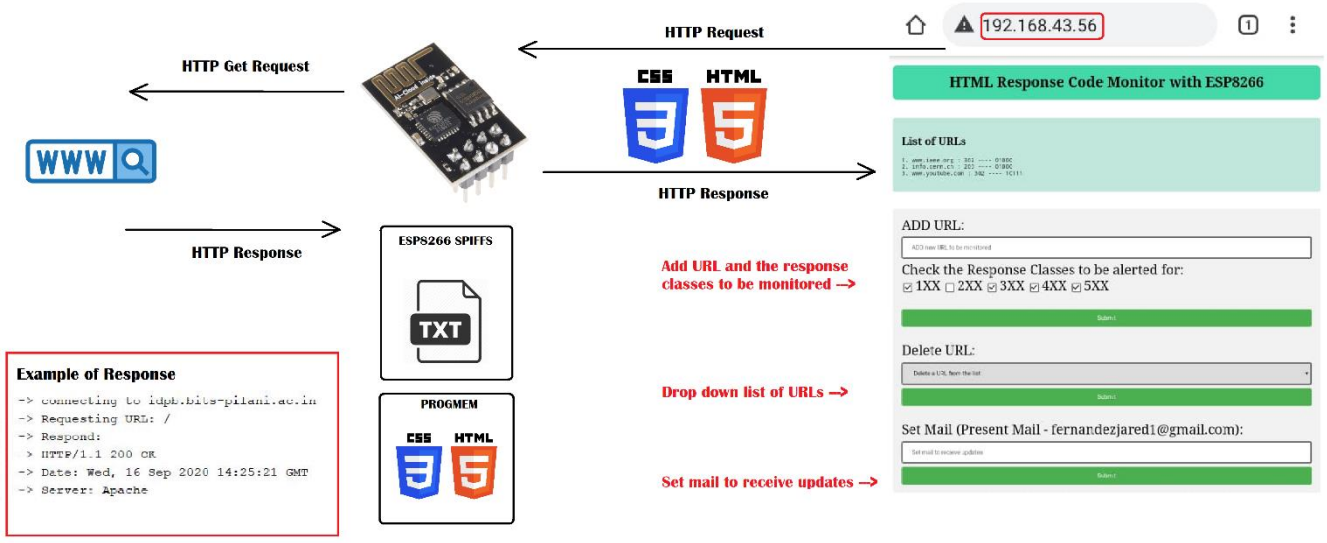
Some of the common Response Codes are:

- 200 – Ok – the page is successfully loaded
- 301 – Permanent redirect
- 302 – Temporary redirect (technically “found”)
- 403 – Forbidden (i.e. password protected)
- 404 – Not Found
- 410 – Gone/removed page
- 500 – Internal server error
- 503 – Service unavailable

Need for a Status Code Monitor

A simple scenario is when a user is hosting a website and wants to be alerted when there is an issue. Or suppose the user want to access a website (for ex: a college application), but it is not possible now and the user wants to be alerted when the site is functioning again. In another scenario, suppose you have a website that handles financial transactions, you might want to monitor any dormant phishing sites so that you are alerted when these are running again. In all of these cases, it can be seen why a Status Code Monitoring System will be useful.

Project Overview



We can build an asynchronous web server using the ESPAsyncWebServer library. This webpage can be accessed by a browser in the network that the ESP is connected to. It presents 3 different configuration options:

1. Add URL Input Text Field and Checkbox for the HTML Response Series to be monitored
2. Drop Down List of URLs from which one can be selected to be deleted
3. Input Text Field for email.

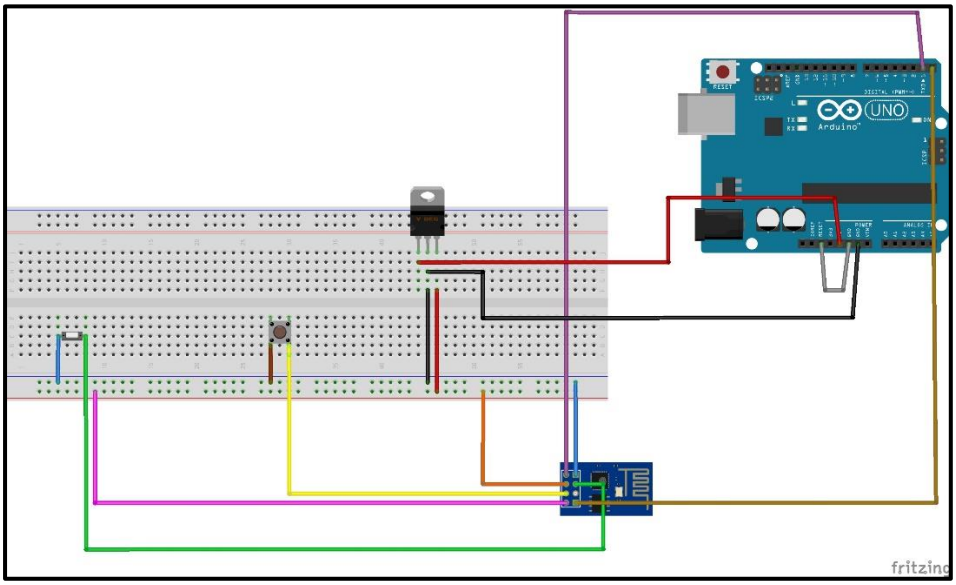
On pressing Submit an HTTP Get Request is sent to the ESP. For ex: If `<inputMessage>` is set as the mail in the input field and submitted. Then a `<ESPID>/get?inputMail=<inputMessage>` Get Request saves `<inputMessage>` in the email variable. The Delete and Add Functions work

in the same way. The list of URLs is stored in a SPIFFS text file. This way, the data is not lost even if the ESP turns off. To get the HTTP Response Code we use the WifiClient class to connect to s specified internet IP address.

Parts Used

1. Wi-Fi Module - ESP8266-01
2. Arduino UNO Board
3. Power Source (Battery)
4. AMS 1117 3.3 V DC - DC Step down Converter
5. Jumper Wires
6. Breadboard
7. Push Button
8. Multimeter

Connections

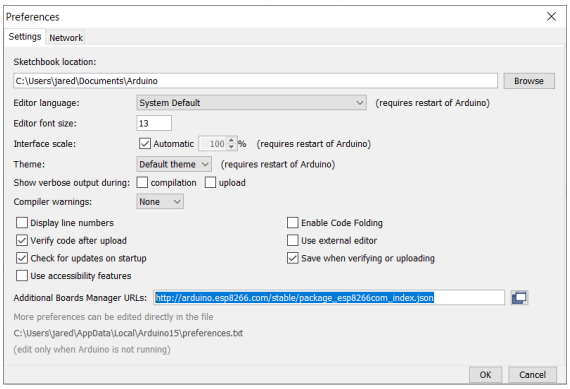


Sketch made on Fritzing

ESP8266	UNO
RX	RX
TX	TX
GND	GND
EN (CH_PD)	VCC(3.3V)
RST	GND(via reset button)
GPIO 0	-------
GPIO2	GND (Flash) or OPEN
VCC(3.3V)	VCC(5 V) Via 3.3V Regulator

Getting Started

1. Installing ESP8266 Board

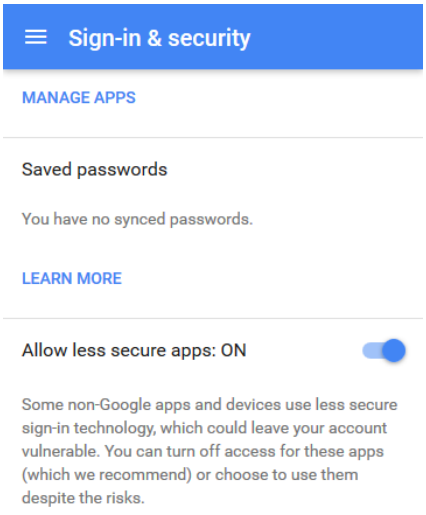


To program the ESP, WE USE THE Arduino IDE. So the necessary files must be installed, Go to Preferences (Ctrl + Comma), in the "Additional Boards Manager URL", add the link: `http://arduino.esp8266.com/stable/package_esp8266com_index.json`. In the Boards Manager (Tools →Board → Boards Manager), search for **ESP8266** and install **esp8266 by ESP8266 community**.

2. Manual Installation of Libraries

To build the asynchronous web server we need the **ESPAsyncWebServer** and the **ESPAsyncTCP** libraries. In the Arduino IDE, go to Sketch → Include Library → Add .zip Library and select the appropriate files.

3. Gmail Account Set-up



We are going to use SMTP to send messages. In SMTP Authentication only email and password are provided, by default Google uses more complex verification methods so we need to change settings. Go to your Google account settings and enable "Allow less secure apps" at the bottom of the page. This mean apps only need your email and password when login to your Gmail account. If case of security concerns, use a backup account, since this is the mail used to the send the message, and the not where you receive the alerts.

How the Code Works

- Including Libraries

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <Hash.h>
#include <FS.h>
#include <ESPAsyncWebServer.h>
#include "Gsender.h"
```

The `Gsender.h` file is a header file that is stored in the same folder as the Arduino code. We import this to add the email facility later. `ESPAsyncWebServer.h` and `ESPAsyncTCP.h` are needed to create the Asynchronous Web Server. The `FS.h` is needed to use the SPIFFS.

Using an asynchronous network means that you can handle more than one connection at the same time. You are called once the request is ready and parsed. When you send the response, you are immediately ready to handle other connections while the server is taking care of sending the response in the background. This is fully asynchronous server and as such does not run on the loop thread and you cannot send more than one response to a single request. [4]

- Declaring Variables

```
String email ="<add recipient email>”;
// NETWORK CREDENTIALS
const char* ssid = "<WIFI_SSID>";
const char* password = "<WIFI_PASS>";

//LOCAL WEBPAGE LOGIN
const char* http_username = "<user_name>";
const char* http_password = "<password>";

//PARAMETERS TO BE USED IN THE HTML CODE
const char* PARAM_STRING = "inputString";
const char* PARAM_DEL = "deleteString";
const char* PARAM_MAIL = "inputMail";
```

Insert the values for the Network Credentials, Webpage Login and the Recipient email. The Parameters to be used in the HTML form are also initialised.

- HTML Form and Processor Function

```
const char index_html[] PROGMEM = R"rawliteral(<html_code>)rawliteral";
```

The HTML code is stored in the PROGMEM, so the data is stored in the flash (memory) instead of the SRAM.[5]

```
<form action="/get" target="hidden-form">
ADD URL: <input type="text" name="inputString" placeholder="ADD new URL
to be monitored"><br>
Check the Response Classes to be alerted for:<br>
<input type="checkbox" name="1XX" value="1"checked>
<label for="1XX"> 1XX </label>
<input type="checkbox" name="2XX" value="2">
<label for="2XX"> 2XX </label>
<input type="checkbox" name="3XX" value="3"checked>
<label for="3XX"> 3XX </label>
<input type="checkbox" name="4XX" value="4"checked>
<label for="4XX"> 4XX </label>
<input type="checkbox" name="5XX" value="5"checked>
<label for="5XX"> 5XX </label><br><br>
<input type="submit" value="Submit" onclick="submitMessage()">
</form><br><br><br><br>
```

The first form option is to enter a URL and select to Response code classes for which the user should be altered by email if that comes up. We use the `<form>` tag to do this. The form has two input types. For the URL we use the `text` input type. For the response code classes we use the `checkbox` input type. The `action` attribute sets where the data is sent on pressing the submit button. Here the values are sent to the ESP via a GET Request. The `onclick` attribute fires on a mouse click on the element. Here it redirects to a JavaScript function that pops up an alert on submission.

```
<form action="/get" target="hidden-form">
    Delete URL: <select id="deleteString" name="deleteString">
    <option value="" disabled selected>Delete a URL from the list</option>
    %DeleteList%
  </select>
  <input type="submit" value="Submit" onclick="submitMessage()">
  </form><br><br><br><br>
```

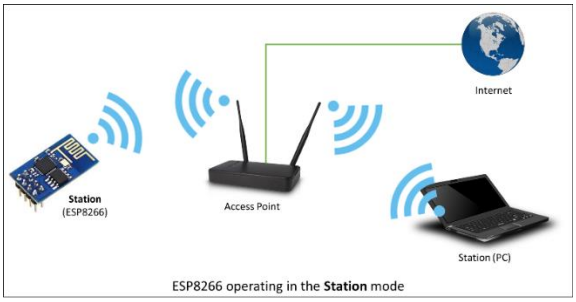
In this form we use the `<select>` tag to create a drop-down list of the URLs. `%DeleteList%` is a placeholder for the HTML code for the options in the drop-down list. For example, we had two options in the list – “OPT_1” and “OPT2”, then the `%DeleteList%` placeholder will be replaced by:

```
<option value="OPT_1">OPT_1</option>
<option value="OPT_2">OPT_2</option>
```

Similarly, a third form is used for the set the email id to get alerts. In the HTML code in we use a TEMPLATE_PLACEHOLDER in some places, these variables are replaced by some String of HTML Code which is determined by the processor function. Ex: `%DeleteList%, %Array%, %PresentMail%`

```
// Replaces placeholder with stored values
String processor(const String& var){
  if(var == "Array"){
    return printarray();
  }
  else if (var=="DeleteList"){
    return dropdown();
  }
  else if (var=="PresentMail"){
    return email;
  }
  return String();
 }
```

- Setup Function



ESP8266 operating in the **Station** mode

In the start of the setup(), we have to connect to the local network. We connect in the WIFI_STA mode which is used to get the ESP8266 connected to a Wi-Fi network established by an access point.

```
//CONNECT TO WIFI
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  if (WiFi.waitForConnectResult() != WL_CONNECTED) {
    Serial.println("WiFi Failed!");
    return;
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());
```

Handling of the GET Requests is also done in the setup function. The below code snippet sends the web page with authentication with the input fields to the client (browser).

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
```

```
    if(!request->authenticate(http_username, http_password))
      return request->requestAuthentication();
    request->send_P(200, "text/html", index_html, processor);
  });
```

Every GET Request will have certain parameters. For example, when you enter the email ID and press submit, the GET Request is `<ESP_IP>/get?inputMail=<inputMessage>`. Here `inputMail` is the parameter. `request->hasParam(<PARAM>)` checks if a request has the parameter `<PARAM>` and `request->getParam(<PARAM>)->value()` returns the value.

```
server.onNotFound(notFound);
server.begin();
```

The `notFound()` function is called if the a request is made on an invalid URL. `server.begin()` starts the server to handle the clients

- Other Custom Functions

Function Name	What It Does
`void notFound(AsyncWebServerRequest *request)`	Handles the Invalid URL Case.
`String readFile(fs::FS &fs, const char * path)`	Reads the data of URL and response classes to be monitored from the file specified by `path` and stores the values in the corresponding array variable.
`void writeFile(fs::FS &fs, const char * path, String message,String codes)`	Adds the URL (`message`) and response classes to be monitored (`codes`) to the file specified by `path` and to the array. If the URL alreadys exists in the list, the classes to be monitoered are updated.
`void delfromFile(fs::FS &fs, const char * path, String message)`	Deletes the URL (`message`) and response classes to be monitored (`codes`) from the file specified by `path` and the array.
`void setmail(String s)`	Sets the `email` variable to string `s`.
`String printarray()`	Returns a string in HTML format to the `processor` function to display the current list of URLs being monitored and its latest response code.
`String dropdown()`	Returns a string in HTML format to the `processor` function to display the drop-down list of URLs.

- Inside the Loop (getting the HTTP Response Code)

We iterate through each of the URLs in a for loop. In each iteration we use the `WiFiClient` class to create a TCP connection. The code snippet below, sends a request to the server and gets the Response. The response code is parsed from the first line of the response received.

```
WiFiClient client;
const int httpPort = 80;
String temphost=host[i];
String tempcodes=classes[i];
if (!client.connect(temphost, httpPort)) {
  Serial.println("connection failed");
  return;
}
Serial.print("Requesting URL: ");
Serial.println(url);
// This will send the request to the server
client.print(String("GET ") + url + " HTTP/1.1\r\n" +
             "Host: " + temphost + "\r\n" +
             "Connection: close\r\n\r\n");
delay(2000);
// Read the line of the reply from server and print them to Serial
Serial.println("Respond:");
String line = client.readStringUntil('\r');
Serial.print(line);
//Substring gives only the 3 digit response code
response[i]=line.substring(9,12);
```

`tempcodes` is a string of length 5 characters which indicates for which classes of response codes an email alert should be issued. For ex: if `tempcodes` was “01100”, then an email alert would be issued whenever a 2XX and 3XX response code.

```
for(int z=0;z<5;z++)
{if(tempcodes.charAt(z)=='1')
   {
       if(int(response[i].charAt(0)-'0')==(z+1))
        {
          <Do_Something>
        }
    }
}
```

- Inside the Loop (sending the email alert)

Email alerts are periodically sent whenever an issue arises. `issue` is a Boolean variable that is used to checked whether an alert should be sent or not. The `GSender` class helps send e-mails.

This is from the `Gsender.h` file that was imported earlier. The recipient email, mail content and subject are given by `email, mail_mes and subject` respectively.

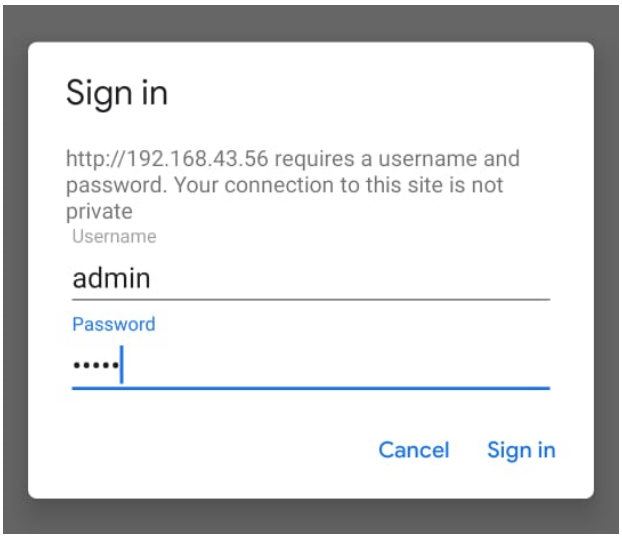
```
if(issue) //Check if mail should be sent
{
 Gsender *gsender = Gsender::Instance();
// Getting pointer to class instance
 String subject = "HTTP RESPONSE Alert";
 if(gsender->Subject(subject)->Send(email, mail_mes)) {
                Serial.println("Message send.");
                mail_mes="";
 } else {
                Serial.print("Error sending message: ");
                Serial.println(gsender->getError());
 }
 }
```

- Gsender.h file

Some minute changes need to be made to this file. The base64 encode email address and the password has to be entered. The `FROM` field is set with email address.

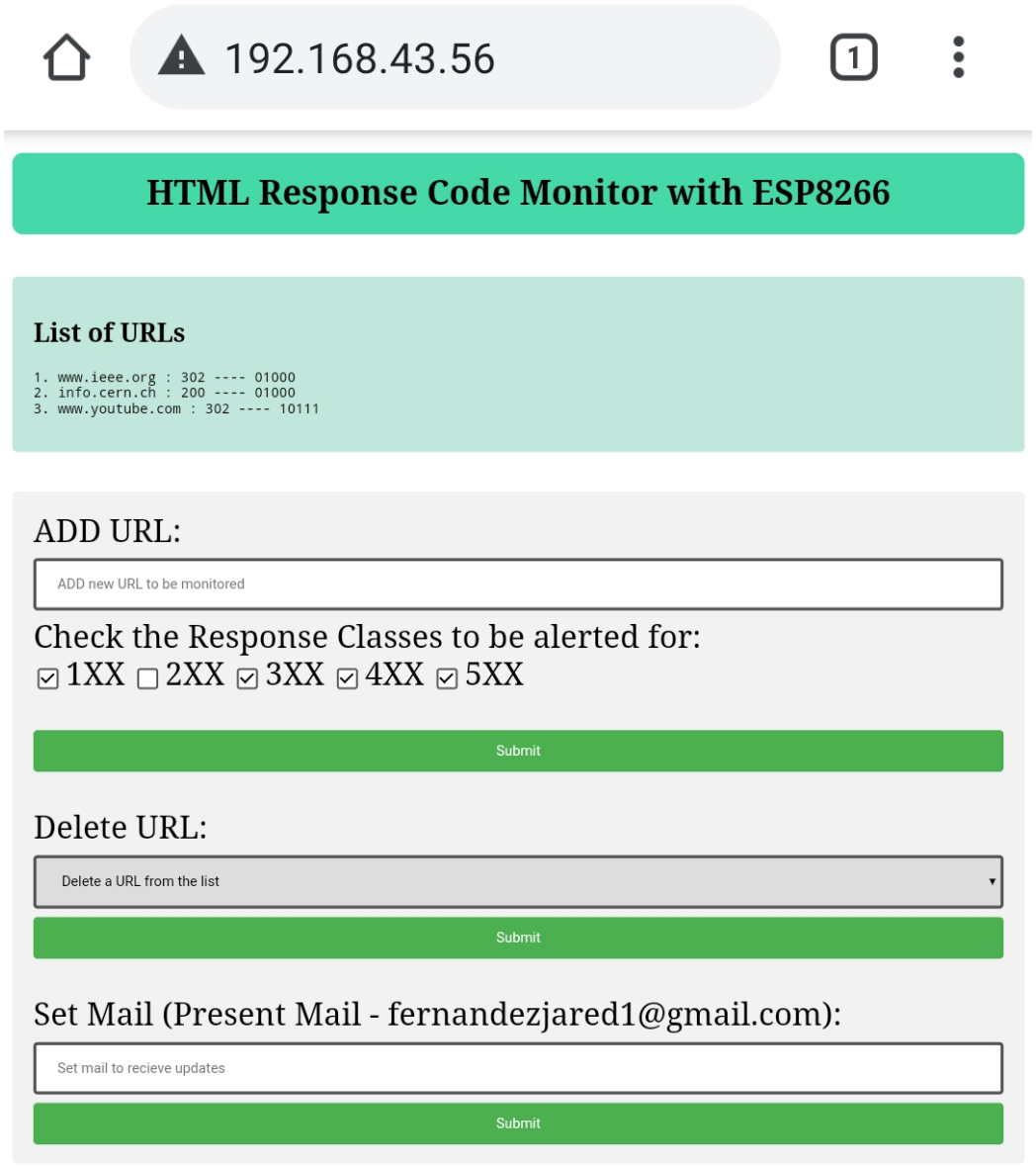
```
const char* EMAILBASE64_LOGIN = "-----------";
const char* EMAILBASE64_PASSWORD = "----------------";
const char* FROM = "<email_here>";
```

Testing and Results

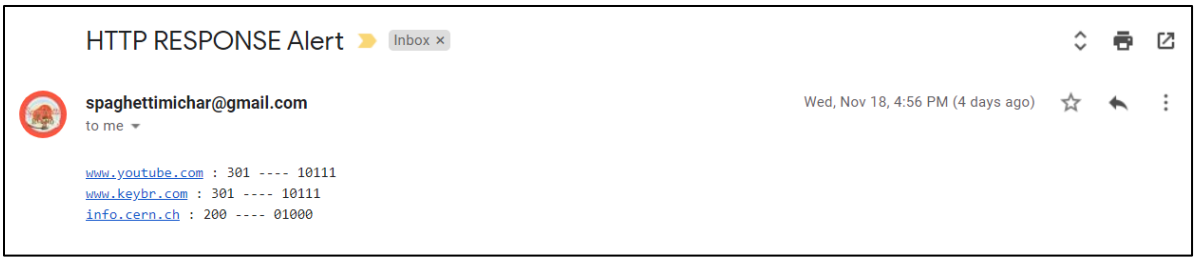


Select the appropriate parameters in Tools menu and upload the code to the ESP8266. Remove the GPIO2 from the GND and press the RST button. On the serial Monitor, the ESP8266 IP address can be seen. Enter this into the web browser.

Enter the login credentials to access the web page.

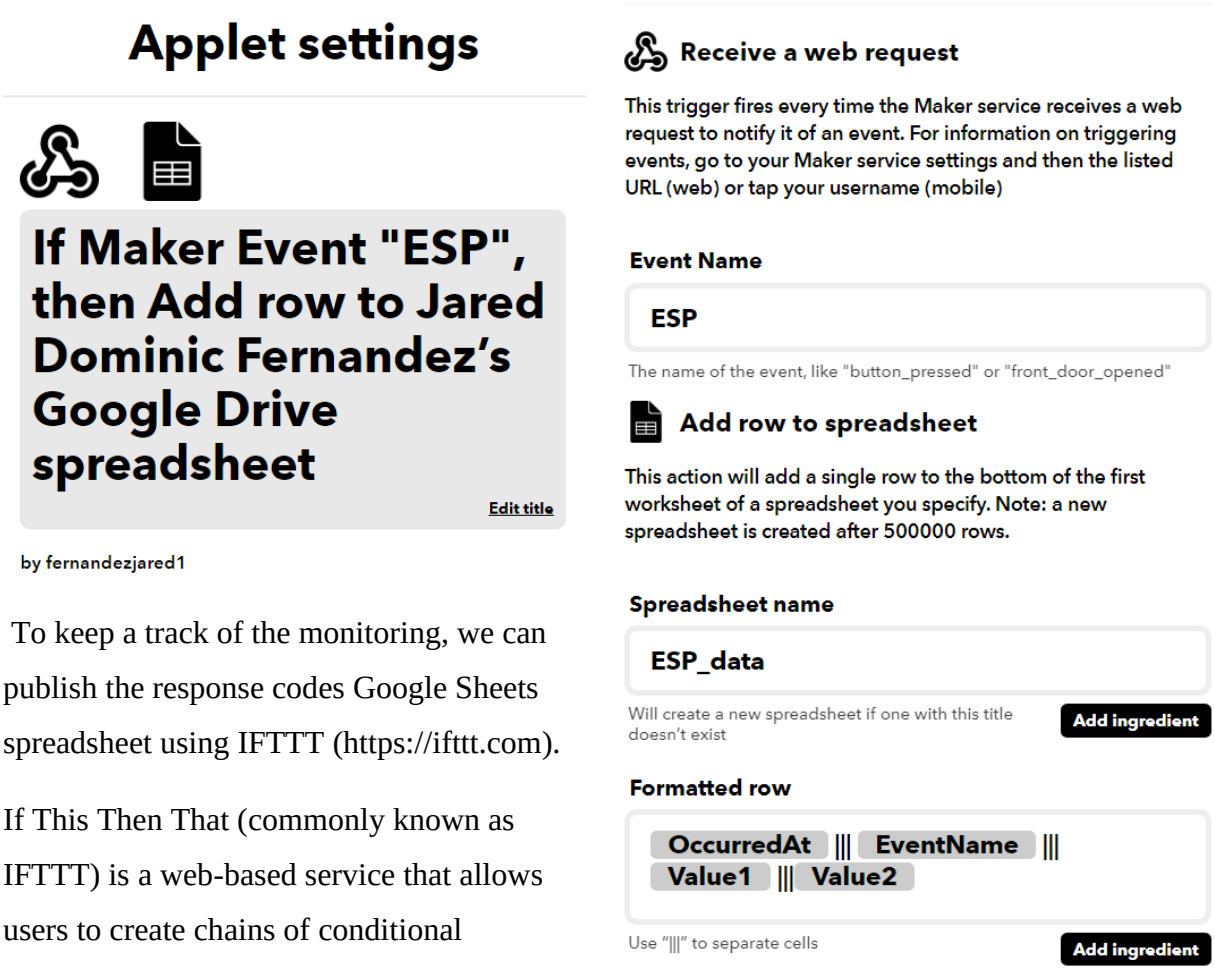


WebPage



Email Alert

Possible Additional Features



To keep a track of the monitoring, we can publish the response codes Google Sheets spreadsheet using IFTTT (https://ifttt.com).

If This Then That (commonly known as IFTTT) is a web-based service that allows users to create chains of conditional statements triggered by changes that occur within another service.[6] We create an applet, that adds the data as a new row to the google sheet whenever it receives a web request notifying it.

Initialise the following variables in the code:

```
char MakerIFTTT_Key[] = "mqxVVHZqvxL9d1xomZ9W4ySNBN3zu1AJNNRDm7N3xjE"; // Obtained when setting up/connecting the Maker channel in IFTTT
char MakerIFTTT_Event[] = "ESP"; // Arbitrary name for the event; used in the IFTTT recipe.
```

The following code must be added in the loop().

```
if(client.connect("maker.ifttt.com",80))
   char post_rqst[256];    // hand-calculated to be big enough

    char *p = post_rqst;
```

```
    p = append_str(p, "POST /trigger/");
    p = append_str(p, MakerIFTTT_Event);
    p = append_str(p, "/with/key/");
    p = append_str(p, MakerIFTTT_Key);
    p = append_str(p, " HTTP/1.1\r\n");
    p = append_str(p, "Host: maker.ifttt.com\r\n");
    p = append_str(p, "Content-Type: application/json\r\n");
    p = append_str(p, "Content-Length: ");

    char *content_length_here = p; // we need to remember where the content
length will go, which is:

    p = append_str(p, "NN\r\n"); // it's always two digits, so reserve space
for them (the NN)
    p = append_str(p, "\r\n"); // end of headers
    char *json_start = p; // construct the JSON; remember where we started
so we will know len
   Serial.println("Sending values to IFTTT");
    p = append_str(p, "{\"value1\":\"");
    p = append_str(p,<ENTER_host_or_URL>);
    p = append_str(p, "\",\"value2\":\"");
    p = append_str_new(p, <ENTER_response_code>);
    p = append_str(p, "\"}");

    // go back and fill in the JSON length
    // we just know this is at most 2 digits (and need to fill in both)
    int i = strlen(json_start);
    content_length_here[0] = '0' + (i/10);
    content_length_here[1] = '0' + (i%10);

    // finally we are ready to send the POST to the server!
    client.print(post_rqst);
```

Powering the device and related issues

The simplest way to power the device while debugging is to power it using the Arduino. Use the 5V pin from the Arduino connected to a 3.3 Voltage regulator to power the ESP. However, we want the device to be battery powered. The ESP can generally operate in a voltage range of 2.5V to 3.6 V. We have the following options:

- 2 AA batteries
- 1 LiFePo4 Battery
- 1 Lipo Battery with voltage regulator
- 1 Li-ion Battery with voltage regulator
- 9V alkaline battery with voltage regulator

Current : ~ 70mA

LiFePo4 seems to be the best case, since it has a very flat discharge curve and can directly be plugged to the ESP. [7]

The issue with is that with a current of *70mA*, even with a 6000mAh 3.2 V LiFePo4 battery, it can run at a stretch for only about 3-4 days. In our case, because we want the local web page to be available all the time, the ESP is connected to the Wi-Fi the whole time. This is very power consuming.

	Modem-sleep	Light-sleep	Deep-sleep
Wi-Fi	OFF	OFF	OFF
System clock	ON	OFF	OFF
RTC	ON	ON	ON
CPU	ON	Pending	OFF
Substrate current	15 mA	0.4 mA	20 µA

There is a solution, to this problem, however it comes at a cost. The ESP8266 has 3 different sleep modes, all in which the Wi-Fi remains off.[8] So to use this, we have to completely eliminate the web page and hard code the URLs to be monitored so that the ESP can check for any issue periodically (say every 30 mins). Using one of the sleeps modes will reduce power consumption.

Conclusions

The use of the ESP8266 board for this system is a very effective set and forget method that can operate on its own. The local webpage hosted on the ESP provides for easy customization. The only needs to worry when an email alert is received. Further improvements can be done using the IFTTT applet. Only a basic knowledge of Arduino programming language, HTML, CSS, and C++ is required to understand the code. However, as we have seen, connecting the ESP to the Wi-Fi for the whole time, increases power consumption. This can we reduced by restricting the time the ESP is connected to the Wi-Fi and leaving it on deep sleep mode otherwise.

References

1. https://www.tutorialspoint.com/http/http_status_codes.htm
2. https://www.restapitutorial.com/httpstatuscodes.html
3. https://stackoverflow.com/questions/39707504/how-to-use-esp8266wifi-library-get-request-to-obtain-info-from-a-website
4. https://github.com/me-no-dev/ESPAsyncWebServer#the-async-web-server
5. https://www.arduino.cc/en/pmwiki.php?n=Reference/PROGMEM
6. https://ifttt.com/about
7. https://www.electriccarpartscompany.com/Bestgo-20Ah-LiFePO4-Lithium-Pouch-Cell
8. https://www.espressif.com/sites/default/files/9b-esp8266-low_power_solutions_en_0.pdf