

2nd Edition

BUILD WEB SERVERS

with ESP32 and ESP8266



Build web servers with ESP32 and ESP8266 using Arduino Core. Learn HTML, CSS and JavaScript to control outputs and monitor sensors remotely.

Rui Santos and Sara Santos

2nd Edition - version 2.1

Security Notice

Sorry for writing this notice, but the evidence is clear: piracy for digital products is over the entire internet.

For that reason, we've taken certain steps to protect our intellectual property contained in this eBook.

This eBook contains hidden random strings of text that only apply to your specific eBook version that is unique to your email address. You won't see anything different since those strings are hidden in this PDF. We apologize for having to do that. It means if someone were to share this eBook, we know who did it, and we can take further legal consequences.

You cannot redistribute this eBook. This eBook is for personal use and is only available for purchase at:

- <https://randomnerdtutorials.com/courses>
- <https://rntlab.com/shop>

Please send an email to the author (Rui Santos - hello@ruisantos.me), if you find this eBook anywhere else.

What we really want to say is thank you for purchasing this eBook and we hope you learn a lot and have fun with it!

Disclaimer

This eBook was written for information purposes only. Every effort was made to make this eBook as complete and accurate as possible. The purpose of this eBook is to educate. The authors (Rui Santos and Sara Santos) do not warrant that the information contained in this eBook is fully complete and shall not be responsible for any errors or omissions.

The authors (Rui Santos and Sara Santos) shall have neither liability nor responsibility to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by this eBook.

Throughout this eBook, you will find some links, and some of them are affiliate links. This means the authors (Rui Santos and Sara Santos) earn a small commission from each purchase with that link. Please understand that the authors have experience with all those products and recommend them because they are useful, not because of the small commissions. Please do not spend any money on products unless you feel you need them.

Other Helpful Links:

- [Ask questions in our Forum](#)
- [Join Private Facebook Group](#)
- [Terms and Conditions](#)

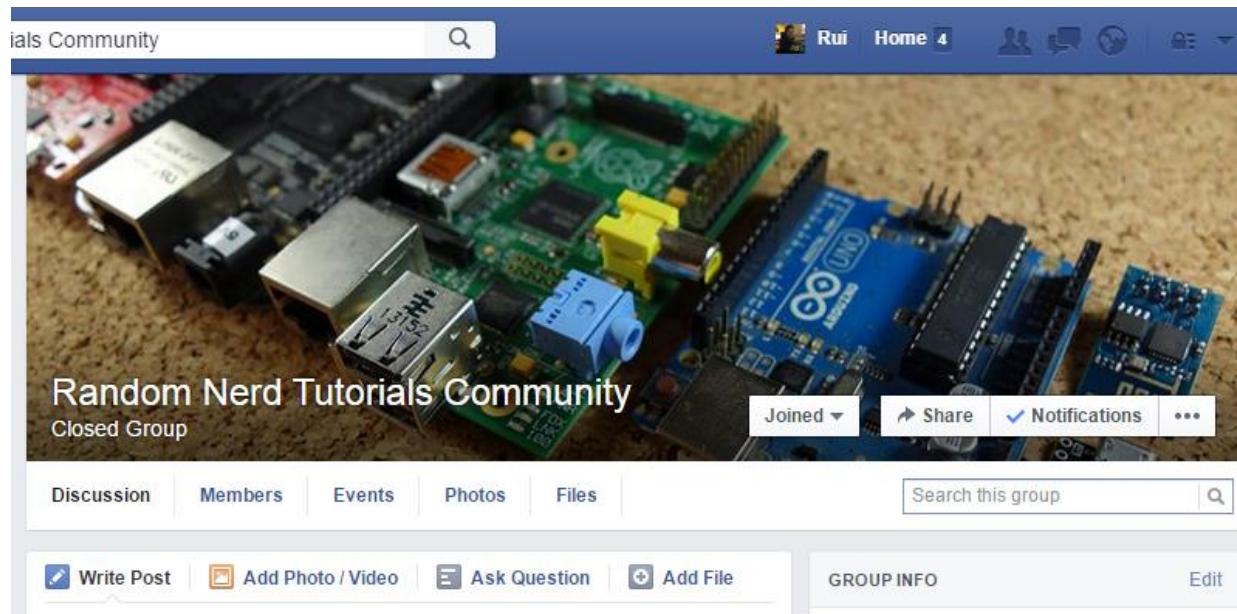
Join the Private Facebook Group

This eBook comes with the opportunity to join a private community of like-minded people. If you purchased this eBook, you can join our private Facebook Group today!

Inside the group, you can ask questions and create discussions about everything related to ESP32, ESP8266, Arduino, Raspberry Pi, etc.

See it for yourself!

- Step #1: Go to -> <http://randomnerdtutorials.com/fb>
- Step #2: Click "Join Group" button
- Step #3: We'll approve your request within less than 24 hours.



About the Authors

This eBook was developed and written by Rui Santos and Sara Santos. We both live in Porto, Portugal, and we have known each other since 2009. If you want to learn more about us, feel free to read our [about page](#).



Hi! I'm Rui Santos, the founder of the Random Nerd Tutorials blog. I have a master's degree in Electrical and Computer Engineering from FEUP. I'm the author of the books: "BeagleBone For Dummies" and "[20 Easy Raspberry Pi Projects: Toys, Tools, Gadgets, and More!](#)". I was the Technical reviewer of the "Raspberry Pi For Kids For Dummies" book. I have worked at the RNT since 2013.



Hi! I'm Sara Santos, and I work with Rui at Random Nerd Tutorials. I have a master's degree in Bioengineering from FEUP, and I'm the co-author of the "20 Easy Raspberry Pi Projects" book. I create, write and edit the tutorials and articles for the RNT and Maker Advisor blogs and help you with your concerns wherever I can. I love books, writing, cats, and a hot cup of tea.

Table of Contents

MODULE 0.....	8
Introduction.....	8
Welcome to "Build Web Servers with the ESP32 and ESP8266"	9
MODULE 1.....	16
Installing VS Code, PlatformIO, and Arduino IDE	16
Installing Visual Studio Code	17
Installing Arduino IDE.....	48
MODULE 2.....	68
Getting Started with HTML, CSS, and JavaScript	68
Getting Started with HTML	69
Styling HTML Content with CSS	104
Getting Started with JavaScript.....	125
MODULE 3.....	142
ESP32 and ESP8266 Web Servers	142
Introducing Web Servers	143
1.1 - Hello World Web Server.....	154
1.2 - Hello World Web Server (Serve Files from Filesystem).....	172
1.3 - Hello World Web Server (Arduino IDE Software)	186
2.1 - Web Server - Control Outputs (ON/OFF Buttons)	193
2.2 - Web Server - Control Multiple Outputs (Toggle Switches)	227

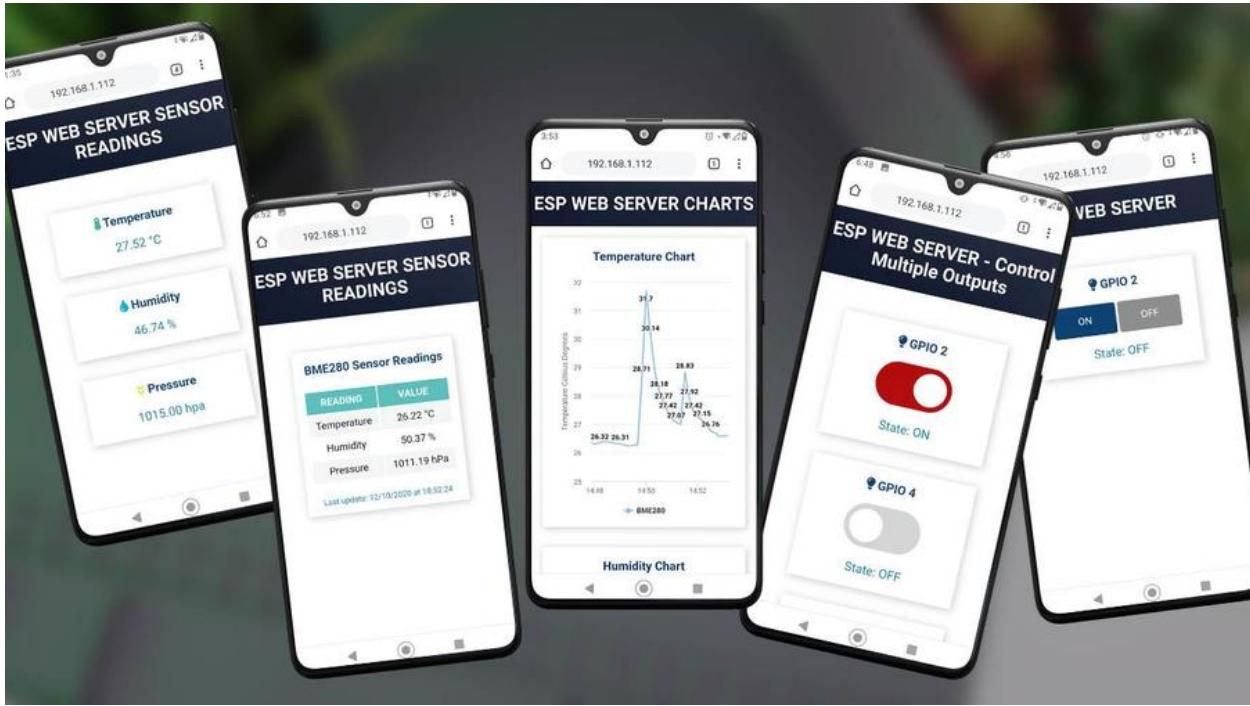
2.3 - WebSocket Web Server: Control Outputs (ON/OFF Buttons)	263
2.4 - Web Server with Slider: Control LED Brightness (PWM).....	288
2.5 – WebSocket Web Server: Control Multiple Outputs	314
3.1 - Web Server: Display Sensor Readings (SSE)	343
3.2 - Web Server: Display Sensor Readings (Table)	370
3.3 - Web Server: Display Sensor Readings (Charts)	394
3.4 - Web Server: Display Sensor Readings from File (Charts)	420
4.1 - Web Server with Input Fields (HTML Form)	462
4.2 – HTTP Authentication: Password Protected Web Server	486
4.3 – Wi-Fi Manager for Web Server	506
4.4 – Multiple Web Pages (with Navigation Bar)	533
4.5 – Over-the-air (OTA) Updates for Web Server	557
EXTRA	583
Useful Resources	583
Access Your ESP32 and ESP8266 Web Servers from Anywhere	584
Setting the ESP32 and ESP8266 as an Access Point.....	593
ESP32/ESP8266 Static IP Address.....	603
CONGRATULATIONS	605
For Completing This eBook.....	605
Other RNT Courses/eBooks.....	607

MODULE 0

Introduction

Introduction: get started with HTML, CSS, JavaScript, and web servers with the ESP32 and ESP8266 boards. Read this section to learn how to follow this eBook, download all code and other valuable resources.

Welcome to "Build Web Servers with the ESP32 and ESP8266"



If you search the web, you'll quickly find hundreds of HTML, CSS, and JavaScript tutorials. However, with this eBook, we have a unique approach. This training aims to teach you the basics of these technologies applied to web servers that you can build with the ESP32 or ESP8266 boards.

- **HTML** to define the content of web pages
- **CSS** to style the web pages
- **JavaScript** to program the behavior of web pages

This tutorial series walks you through building web pages with practical examples to follow and test in real-time. The examples involve a web page hosted in your ESP32 or ESP8266 boards featuring buttons, charts, tables, input fields, and more. You can control outputs and monitor sensors remotely.

If you want to control the ESP board's GPIOs or display sensor readings using a web server, this is the perfect chance for you to learn how to do it from scratch.

Here's an overview of what you'll learn throughout this eBook:

- HTML, CSS, JavaScript, Web Servers, ESP32 and ESP8266 Web Programming;
- Use VS Code (Visual Studio Code) with PlatformIO IDE to program your boards;
- HTML basics: building a basic web page;
- CSS basics: customizing your web pages' appearance;
- JavaScript basics: update variables, handle events, make requests, and more;
- Build your ESP32 or ESP8266 web servers from scratch;

Overview

This eBook is divided into three Modules:

1. Preparing your tools
2. HTML, CSS and JavaScript Basics
3. ESP32/ESP8266 Web Servers

In Module 1, you'll install all the tools required to write HTML, CSS, JavaScript and program the ESP32 and ESP8266. We'll use VS Code with the PlatformIO IDE extension. However, if you prefer, you can use Arduino IDE to program your boards and any other text editor for the web pages.

The second Module teaches you the basics of HTML, CSS, and JavaScript. We'll address the most relevant topics that can be useful to build web pages to control your ESP32 or ESP8266 boards. For this section of the eBook, you don't need an ESP32 or ESP8266 board. You'll learn how to use HTML, CSS, and JavaScript to build simple web pages.

Finally, in the third Module, you'll learn how to build a web server with the ESP32 and ESP8266 boards to host the web pages you've created in Module 2. The ESP32 and ESP8266 boards will be programmed using the Arduino Core and its programming language. You'll also learn how to use those web pages to control the ESP32/ESP8266 outputs and monitor sensors for real-world applications.

HTML, CSS and JavaScript - Quick Introduction

A web page usually requires three things: HTML, CSS, and JavaScript.

HTML stands for Hypertext Markup Language. It is a markup language used to create web pages. Web browsers were created to read HTML files. A web browser reads HTML tags to interpret what content goes on the page and where it is placed. You'll learn about HTML tags later in this eBook.

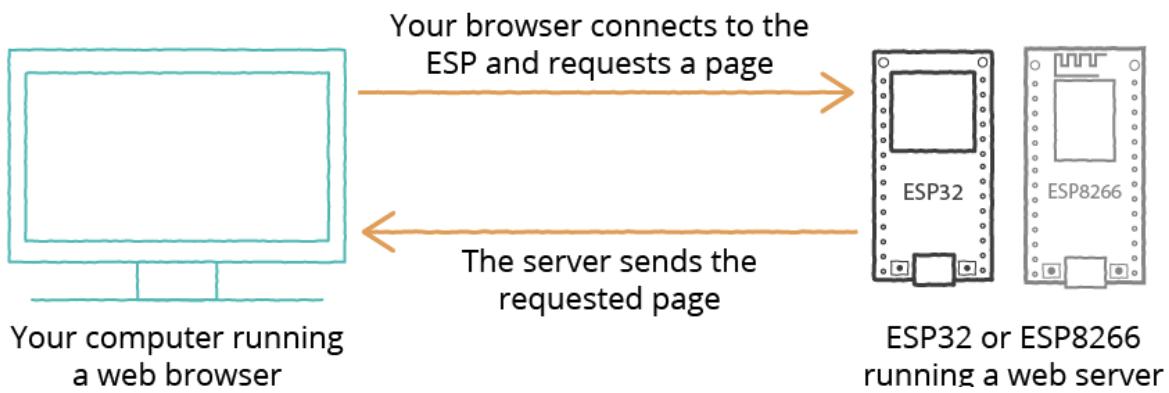
HTML is a markup language, not a programming language. A markup language turns your text into an image, a link, a list, a table, etc.

CSS stands for Cascading Style Sheets, and it describes the appearance of your HTML documents. Cascading Style Sheets can format all elements or just one element when more than one style is applied (hence, the name Cascading). This is what makes your web page pretty: colors, background, alignment, font size, etc.

JavaScript is a programming language most commonly used on websites - it's the programming language of HTML and the Web. If you've ever visited a website that does incredible things through interactive buttons, sliders, gauges, charts, alert messages, or pop-up windows, some JavaScript was certainly working on the back end.

ESP32/ESP8266 Web Servers - Quick Introduction

Simply, a web server is a piece of "software + hardware" that provides web pages. It stores the website's files, including all HTML documents and related assets like images, CSS style sheets, JavaScript files, fonts, and/or other files. It also brings those files to the user's device web browser when the user makes a request to the server.



In our case, the server of these files is the ESP32 or ESP8266 board. This means the board stores the files that are needed to build the web pages. Your board should handle what happens when it receives a request from the client (you on your web browser). The request can be triggered when you click something on the web page like a button to change the state of a GPIO, or it can be triggered by a JavaScript function to request data like sensor readings to be displayed on tables or charts. We'll address ESP32/ESP8266 web servers in greater detail in a later section.

In the examples provided throughout this eBook, the web page will be hosted locally. This means that it will be one or more files on your ESP32 or ESP8266 board (server) that you can access on your local network or on the ESP network. It is not something you can access through the Internet from a different network. However, there's a section in the eBook where you'll learn how to make your web servers available from anywhere, which can be applied to all examples presented.

How to Follow this eBook?

First, you should follow Module 1 to install the required software to write code for building web pages and program the ESP32 and ESP8266 boards. We'll use VS Code with the PlatformIO IDE extension.

Module 2 covers the basics of HTML, CSS, and JavaScript. Even if you're already familiar with HTML, CSS, and JavaScript, we recommend taking a quick look at this module before proceeding to Module 3.

Module 3 covers building web servers with the ESP32 and ESP8266. We recommend following Projects 1.1 and 1.2 first. Then, you can follow the projects in order or pick any project that you find more interesting.

Download Source Code and Resources



Each section contains the code, schematics, and all the resources you need to complete the project. You can download all the resources for a specific project at the end of its Unit.

Alternatively, you can download the "Build Web Servers with the ESP32 and ESP8266" eBook repository and instantly download all the resources for this eBook.

- [Download All eBook Resources »](#)

(<https://github.com/RuiSantosdotme/build-web-servers-ebook/archive/main.zip>)

Getting Parts

To follow the projects, you need some electronics components. In each section, we provide a list of the required parts and links to [Maker Advisor](#) so that you can find the part you're looking for in your favorite store at the best price.



If you buy your parts through Maker Advisor links, we'll earn a small affiliate commission (you won't pay more for it). By getting your parts through our affiliate links, you are supporting our work. If there's a component or tool you're looking for, we advise you to look at [our favorite tools and parts here](#).

List of Parts

- [ESP32](#) – read [Best ESP32 development boards](#)
- [ESP8266](#) – read Best [ESP8266 development boards](#)
- [LEDs](#)
- [220 Ω resistors](#) or similar values
- [Jumper wires](#)
- [BME280 temperature, humidity, and pressure sensor](#)
- [Breadboard](#)

To follow the projects in the eBook, you can either use an ESP32 or ESP8266 board. We usually work with the [ESP32 DEVKIT DOIT](#) board, but you can use any other ESP32 board. As for the ESP8266, we use the [ESP8266-12E NodeMCU Kit](#), but once again, you are free to use any other ESP8266 board.

The projects throughout this eBook don't require an extensive list of parts. You just need some LEDs and resistors, jumper wires, a breadboard, and a BME280 sensor.

Leave Feedback

Your feedback is very important so that we can improve the eBook and our learning materials. Suggestions, rectifications, and your opinion are essential.

You can use the following channels to leave feedback:

- [Facebook group](#)
- [RNT Lab Forum](#)

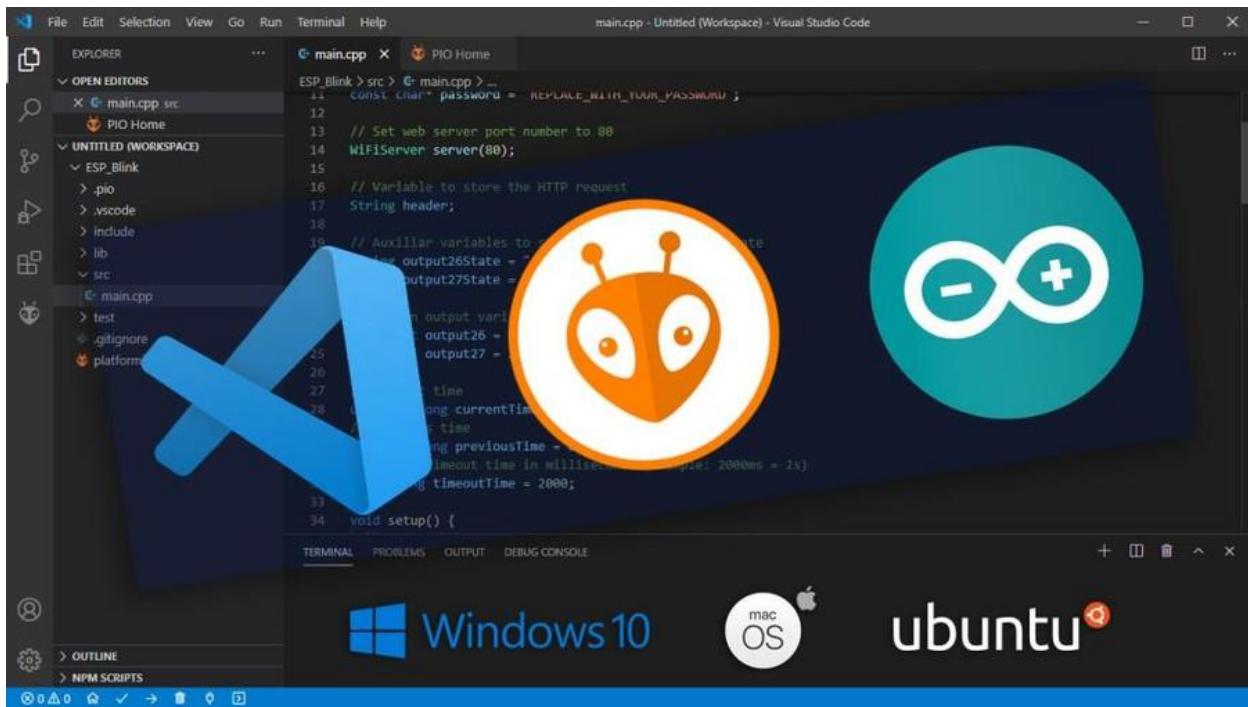
MODULE 1

Installing VS Code, PlatformIO, and Arduino IDE

Install all the tools required throughout the eBook:

- Visual Studio Code
- PlatformIO IDE Extension
- Arduino IDE (not mandatory)

Installing Visual Studio Code



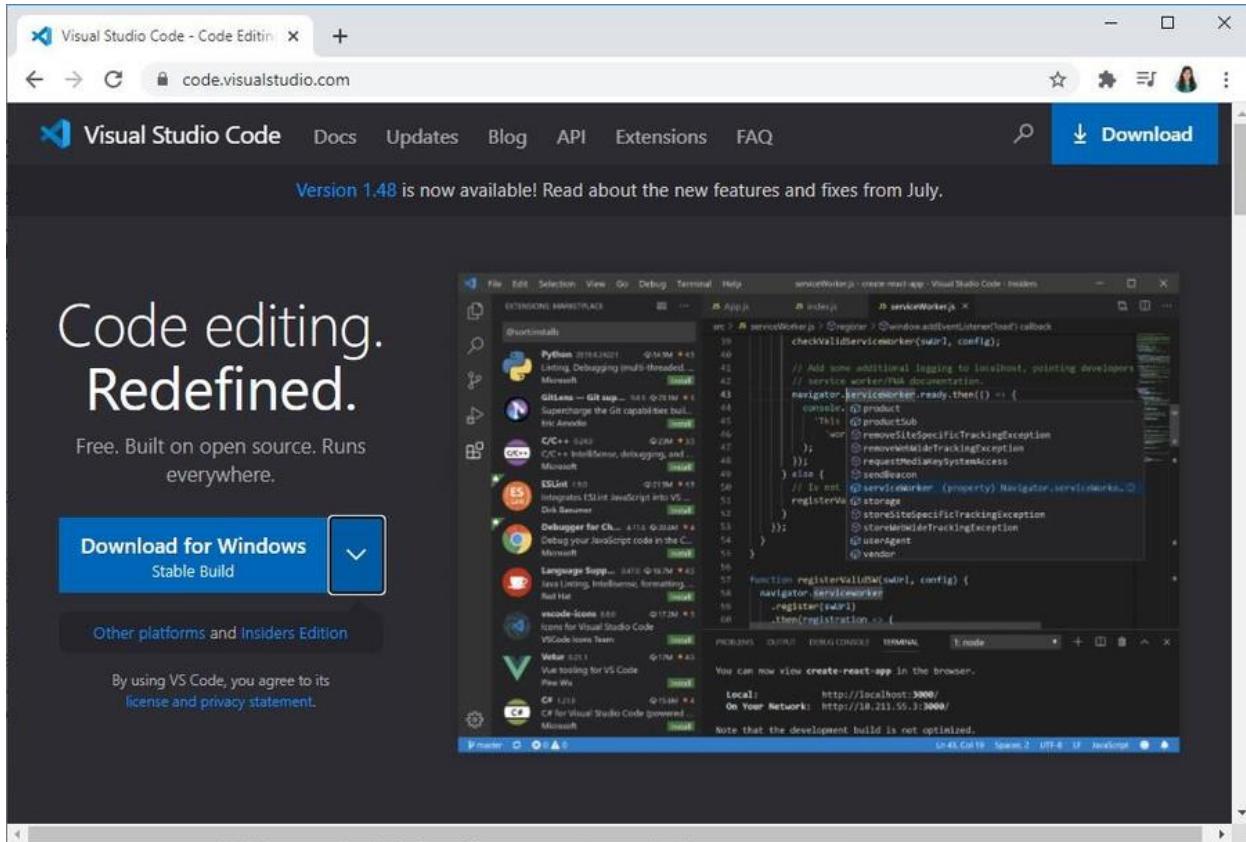
To write your HTML, CSS, and JavaScript files to build web pages for your web servers, you need a text editor. There are many text editors you can use. At the moment, our favorite text editor is Visual Studio Code. It runs on your computer, and it's freely available for Windows, Mac OS X, and Linux operating systems.

Additionally, you can install the PlatformIO extension that allows you to program the ESP32 and ESP8266 boards using the Arduino core. This is convenient because you can use a single piece of software to write your files and program the ESP32 and ESP8266 boards.

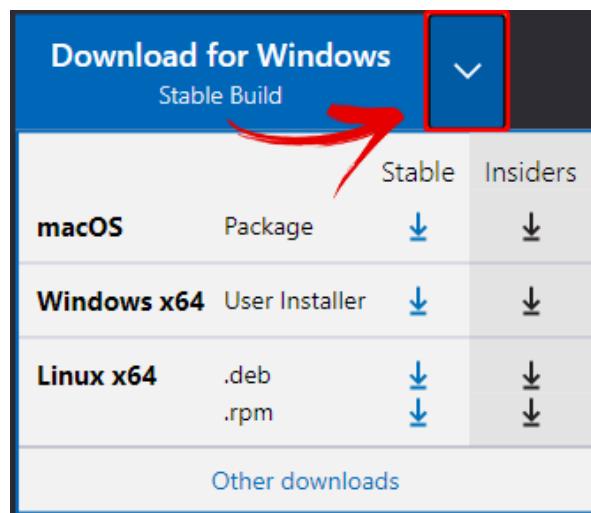
You can use any other text editor software like Notepad++, Atom IDE, Sublime Text, etc. If you're already used to any other text editor, you can stick with it. Otherwise, we recommend following the next instructions to install Visual Studio Code on your computer.

Installing Visual Studio Code – Windows

- 1) Go to <https://code.visualstudio.com/> and download the stable build for your operating system.

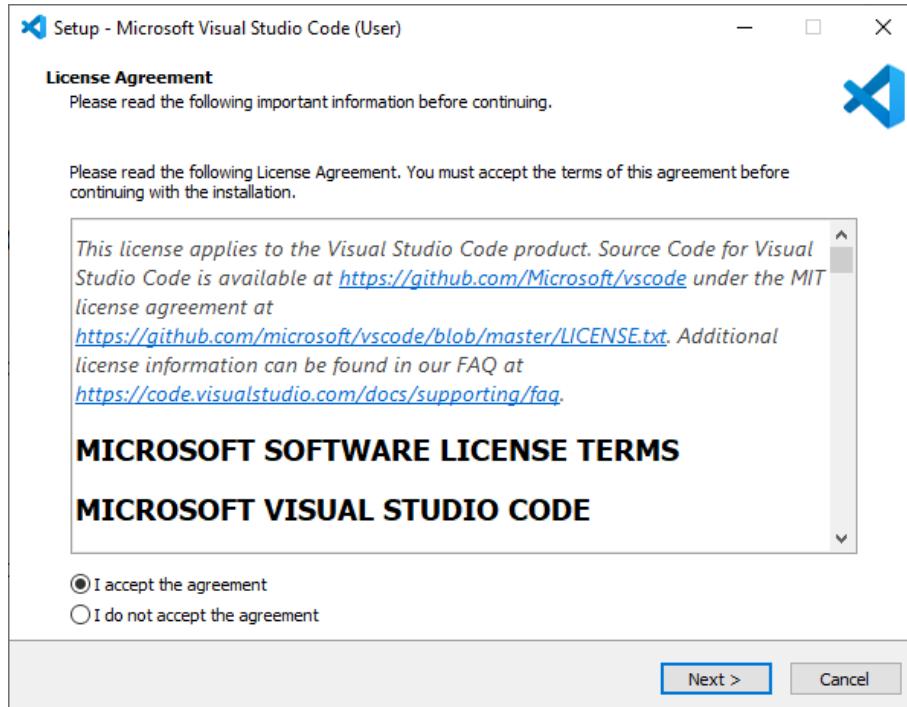


You can click on the arrow to select the correct build for your OS.

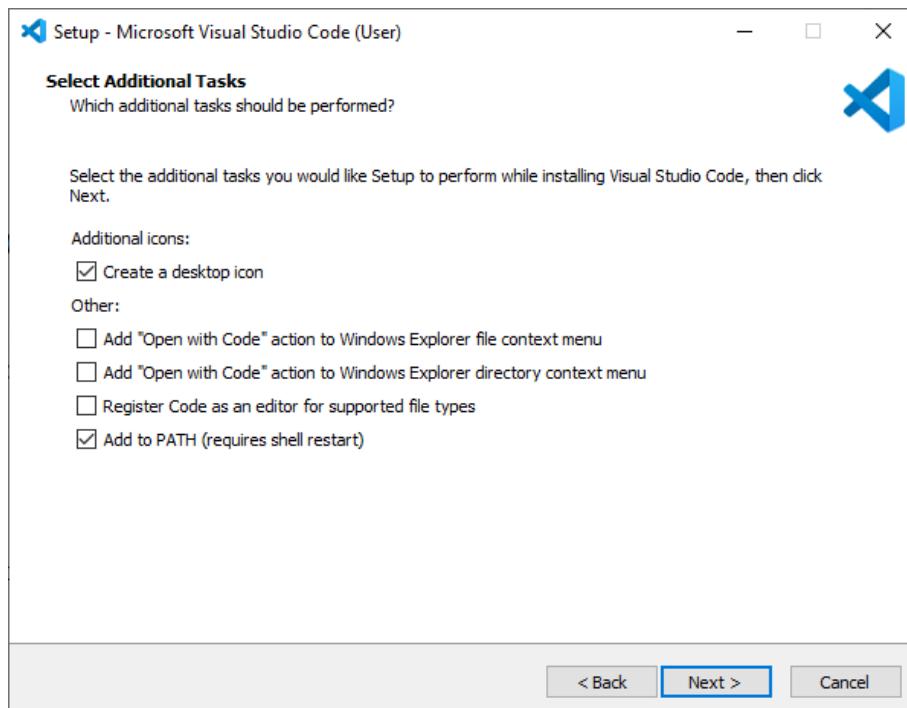


2) Click on the installation wizard to start the installation and follow all the steps to complete the installation.

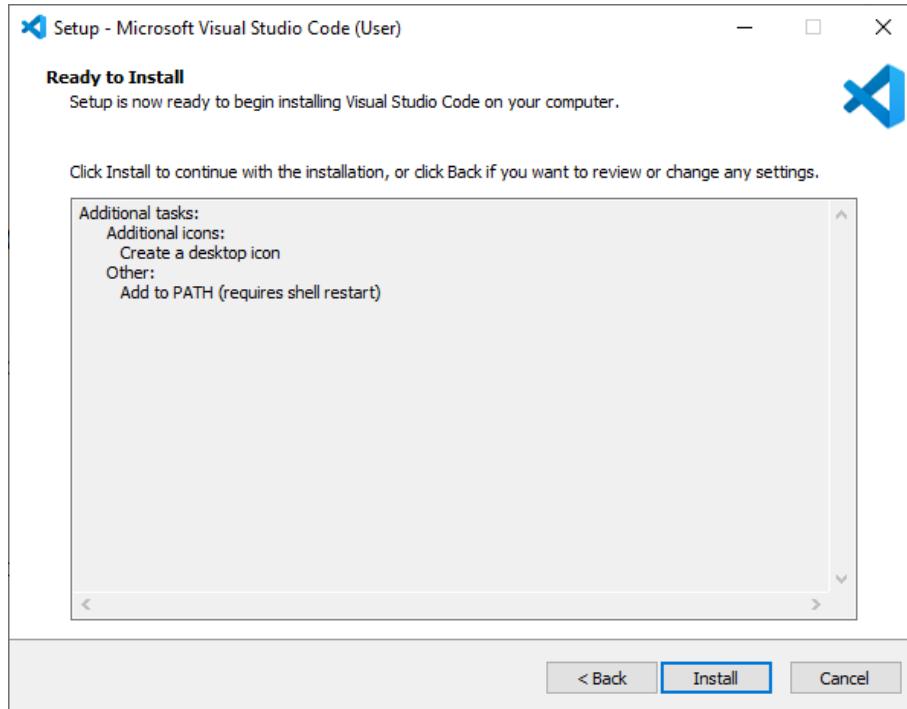
3) Accept the agreement and click the **Next** button.



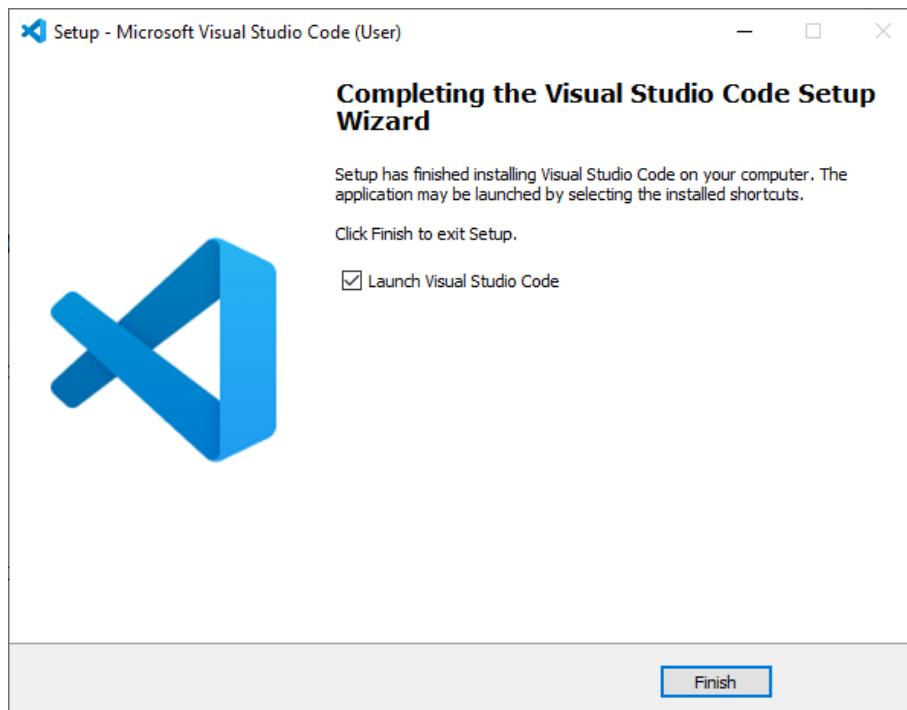
4) Select the following options and click **Next**.



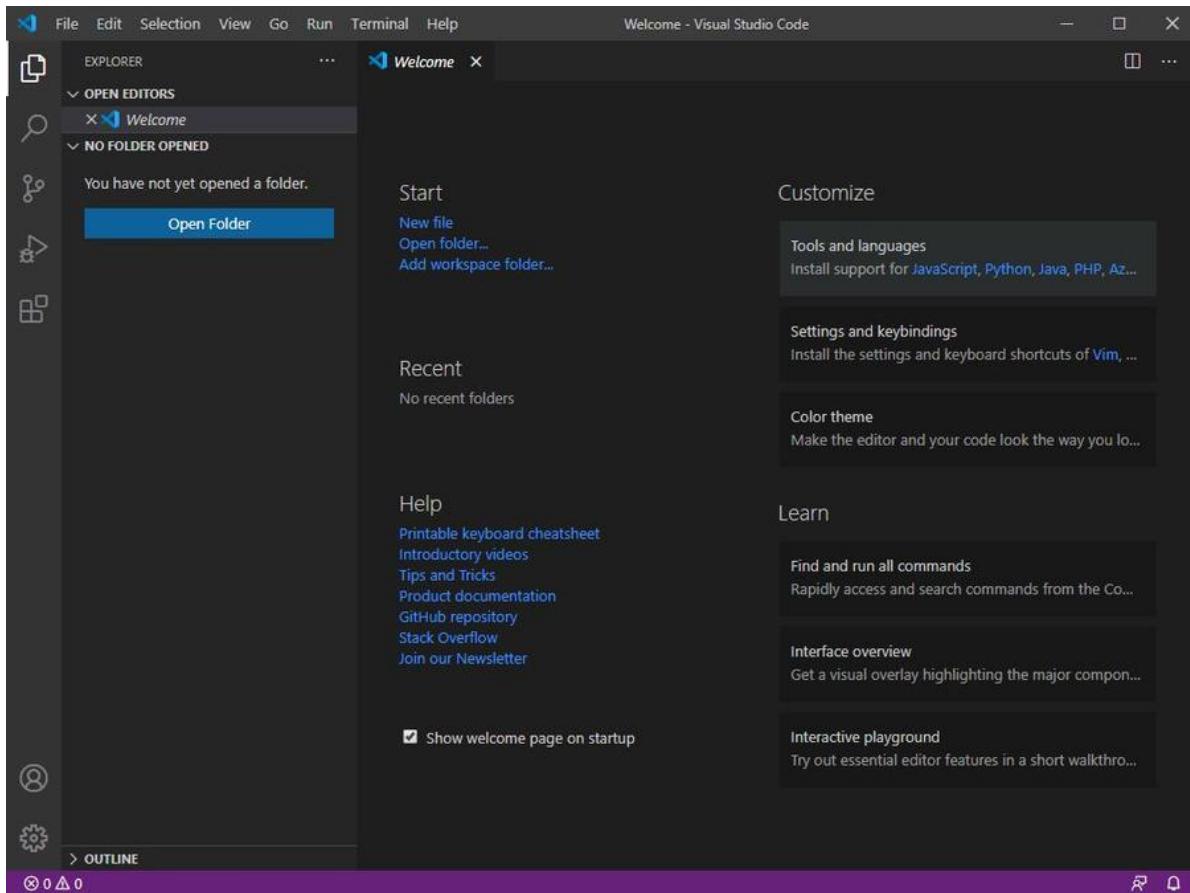
5) Click the **Install button.**



6) Finally, click **Finish to complete the installation.**



Open VS Code, and you'll be greeted by a Welcome tab with the newest version's released notes.



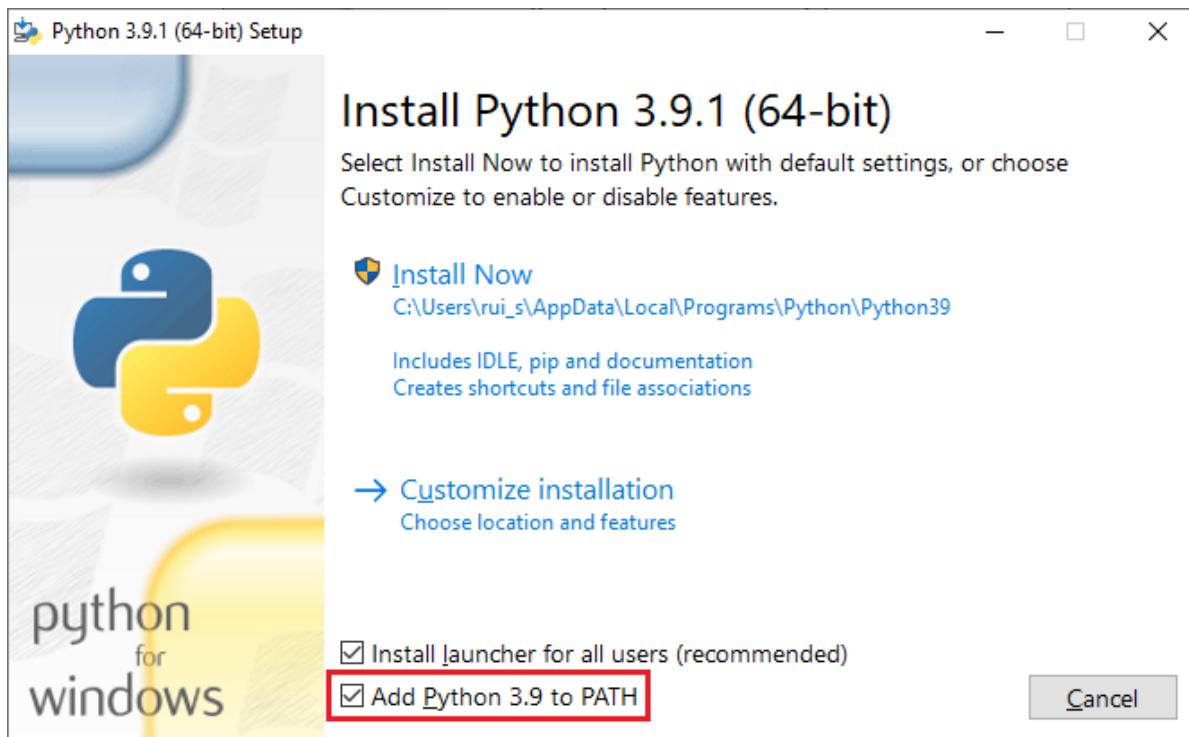
That's it. You installed Visual Studio Code successfully.

Installing Python on Windows

It is possible to program the ESP32 and ESP8266 boards using Visual Studio Code with the PlatformIO extension. To program the ESP32 and ESP8266 boards with PlatformIO IDE, you need Python 3.5 or a later version installed on your computer. We're using Python 3.9.1.

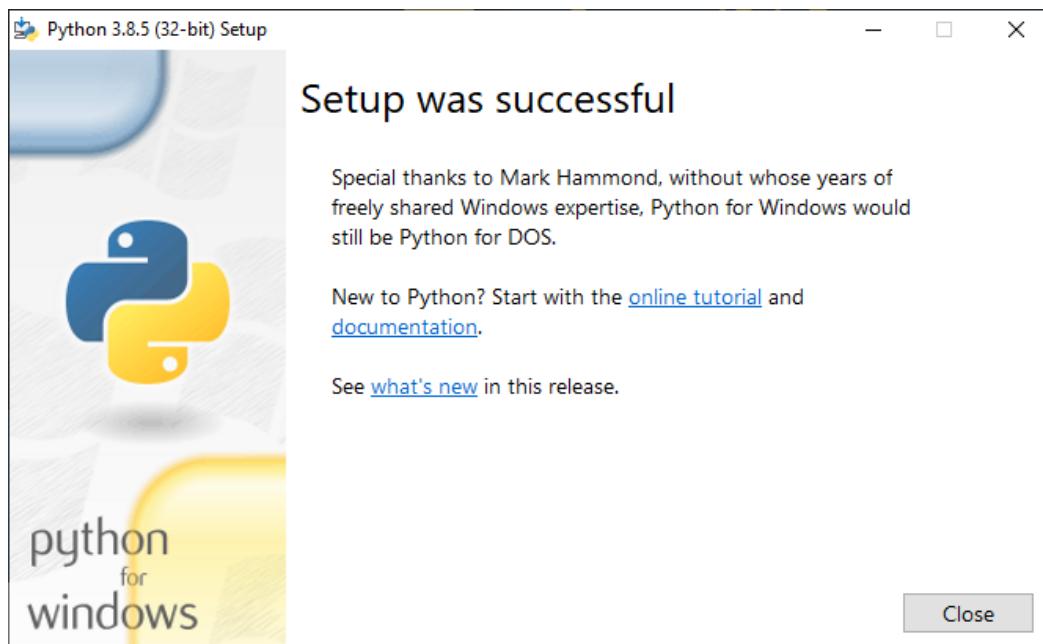
Go to python.org/download and download Python 3.9.1 or a later version.

Open the downloaded file to start the Python installation wizard. The following windows show up.



Important: make sure you check the option Add Python 3.9 to PATH. Then, you can click on the **Install Now** button.

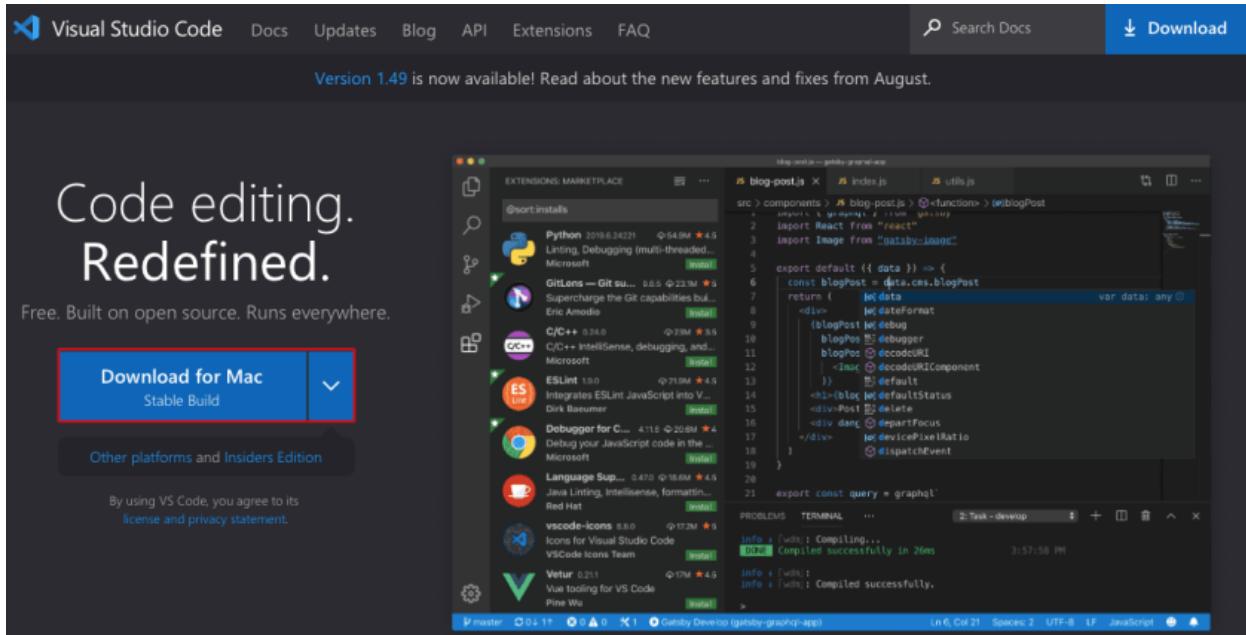
If the installation is successful, you'll get the following message.



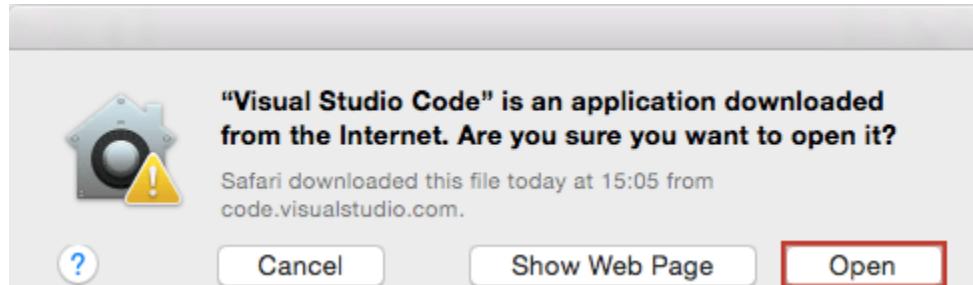
Finally, click the **Close** button.

Installing Visual Studio Code - Mac OS X

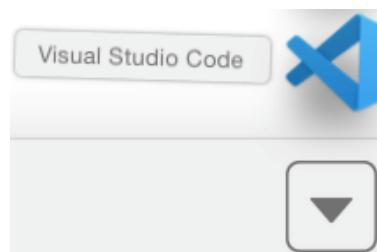
Go to <https://code.visualstudio.com/> and download the stable build for your operating system (Mac OS X).



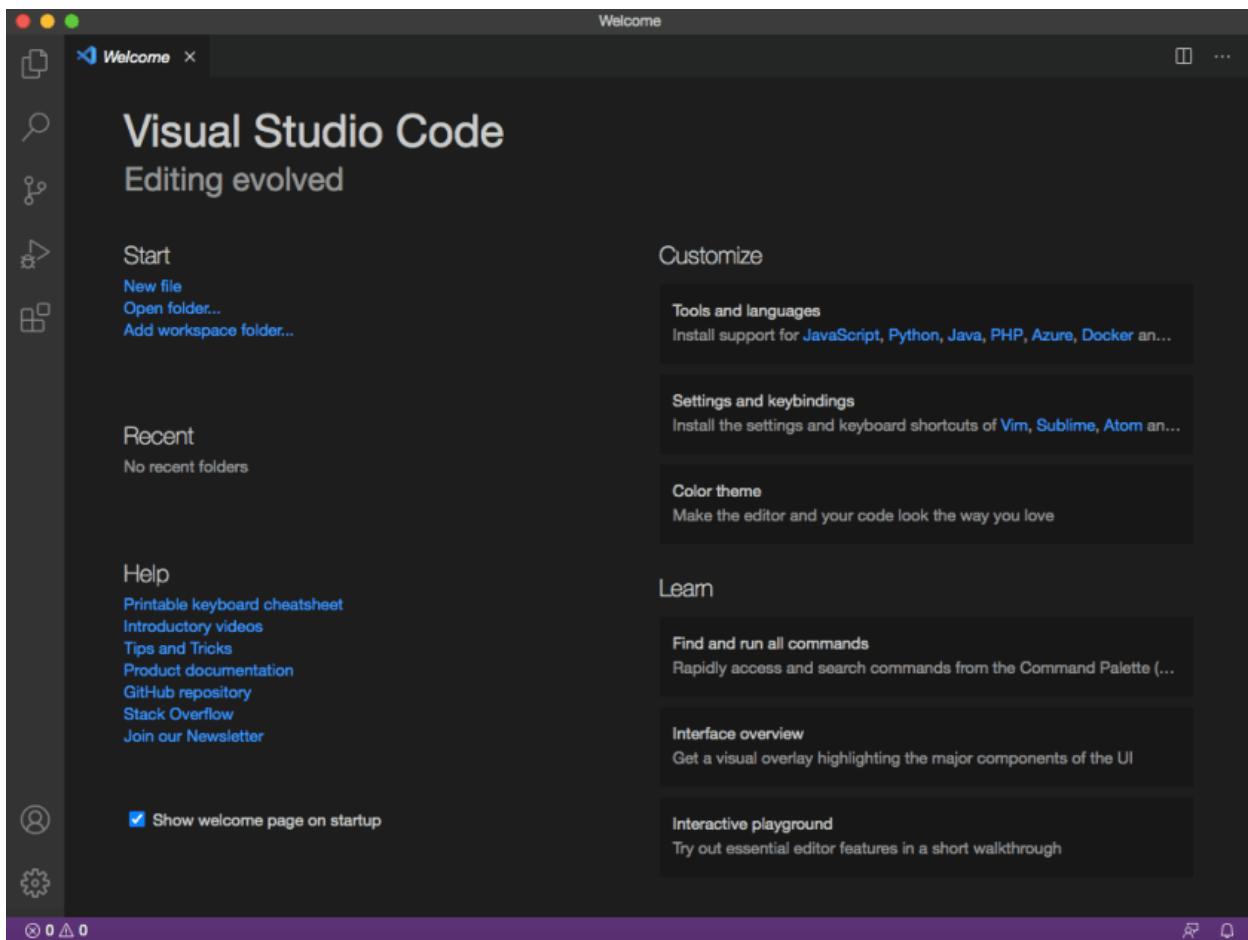
After downloading the Visual Studio Code application file, you'll be prompted with the following message. Press the "Open" button.



Or open your Downloads folder and open Visual Studio Code.



You'll be greeted by a Welcome tab with the latest version's released notes.



That's it. Visual Studio Code was successfully installed.

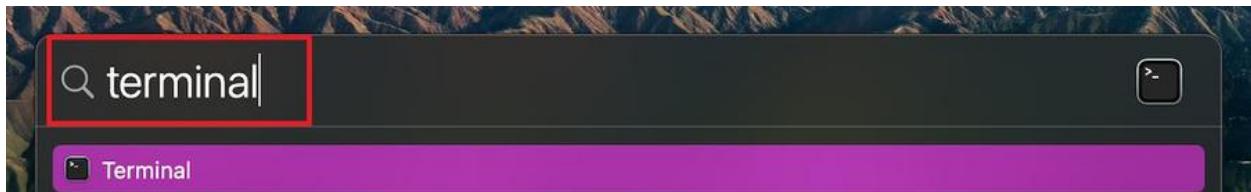
Installing Python on Mac OS X

To program the ESP32 and ESP8266 boards with PlatformIO IDE, you need Python 3.5 or a later version installed on your computer. We're using Python 3.9.1.

You need to open a Terminal window. There are several ways to open a Terminal window. You can open it through Spotlight Search. To launch Spotlight, click the small magnifying glass icon in your menu bar (or press **Cmd+Space**).



Then, type "terminal" and click on the Terminal icon to launch it.



To install Python, I'll be using *Homebrew*. If you don't have the *brew* command available, type the next command in the Terminal window:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

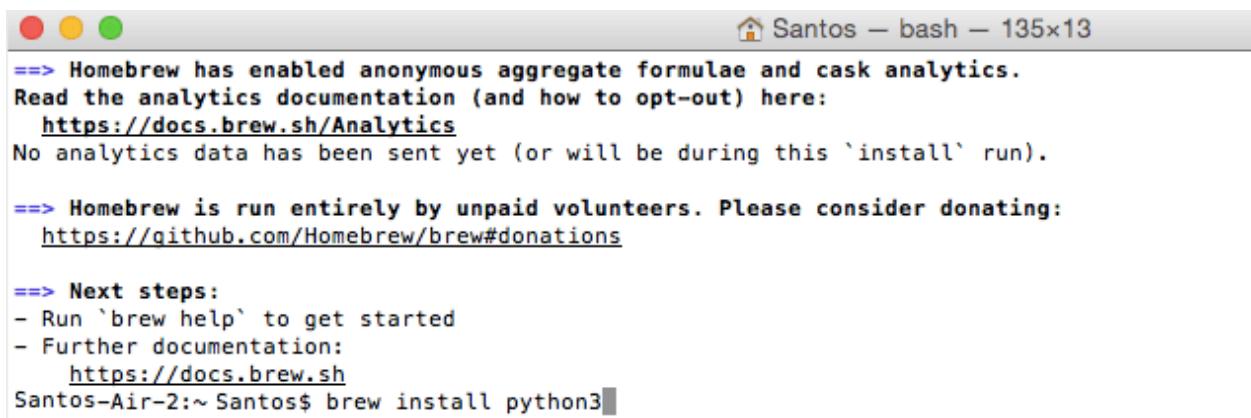
Here's how it looks on the Terminal window:



Then, run the *brew* command to install Python 3.X:

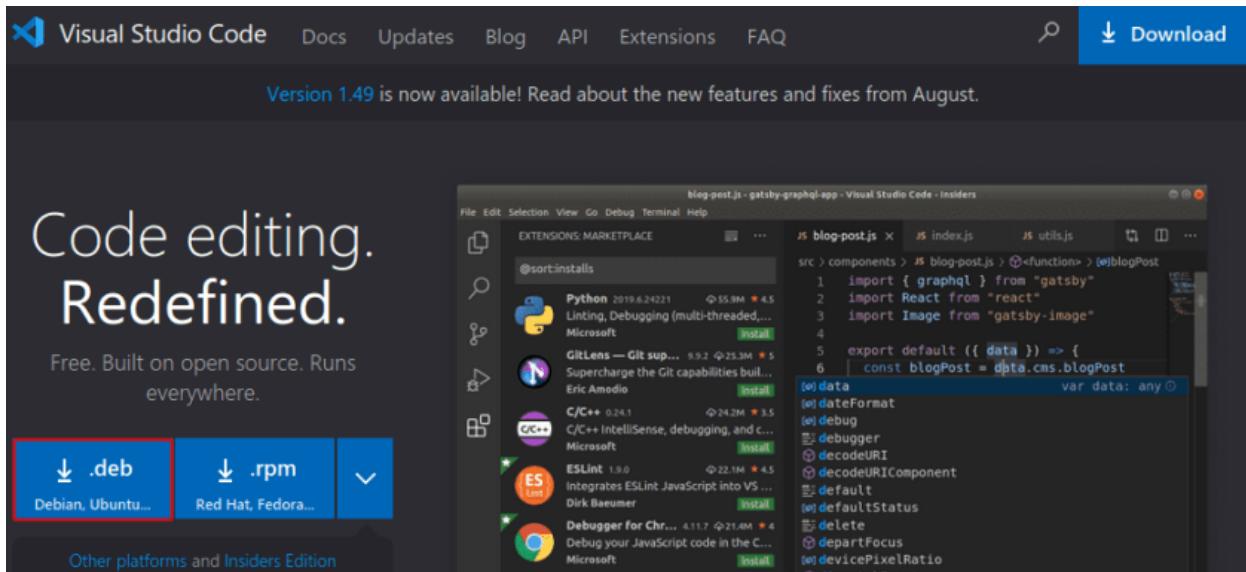
```
$ brew install python3
```

Here's how it looks on the Terminal window:

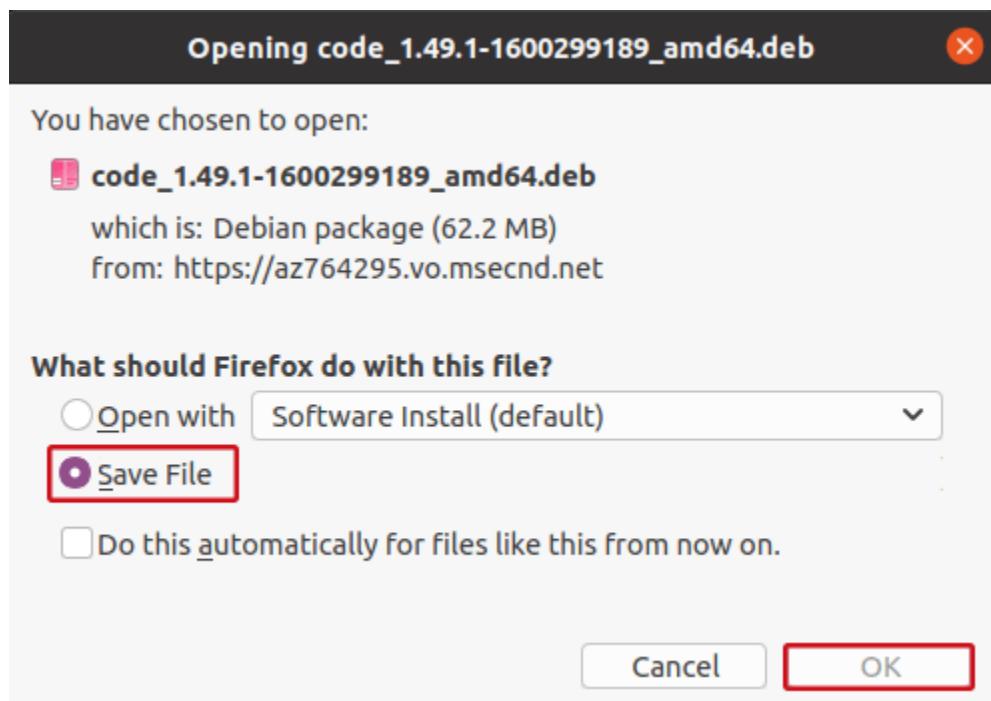


Installing Visual Studio Code - Linux Ubuntu

Go to <https://code.visualstudio.com/> and download the stable build for your operating system (Linux Ubuntu).



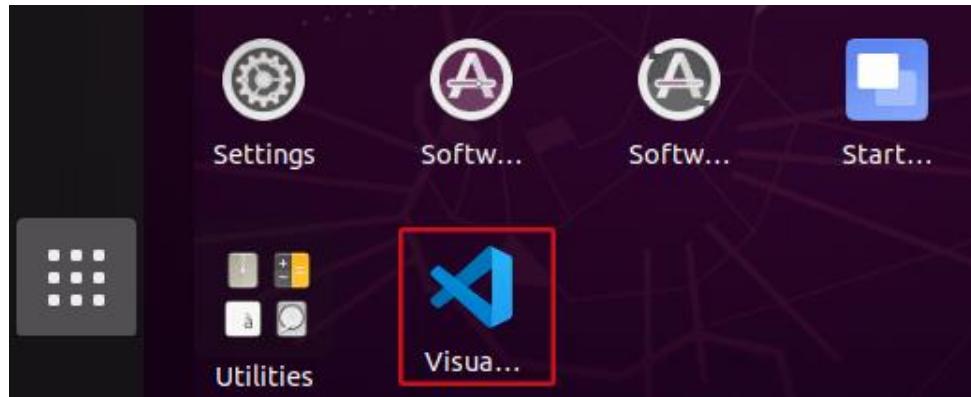
Save the installation file:



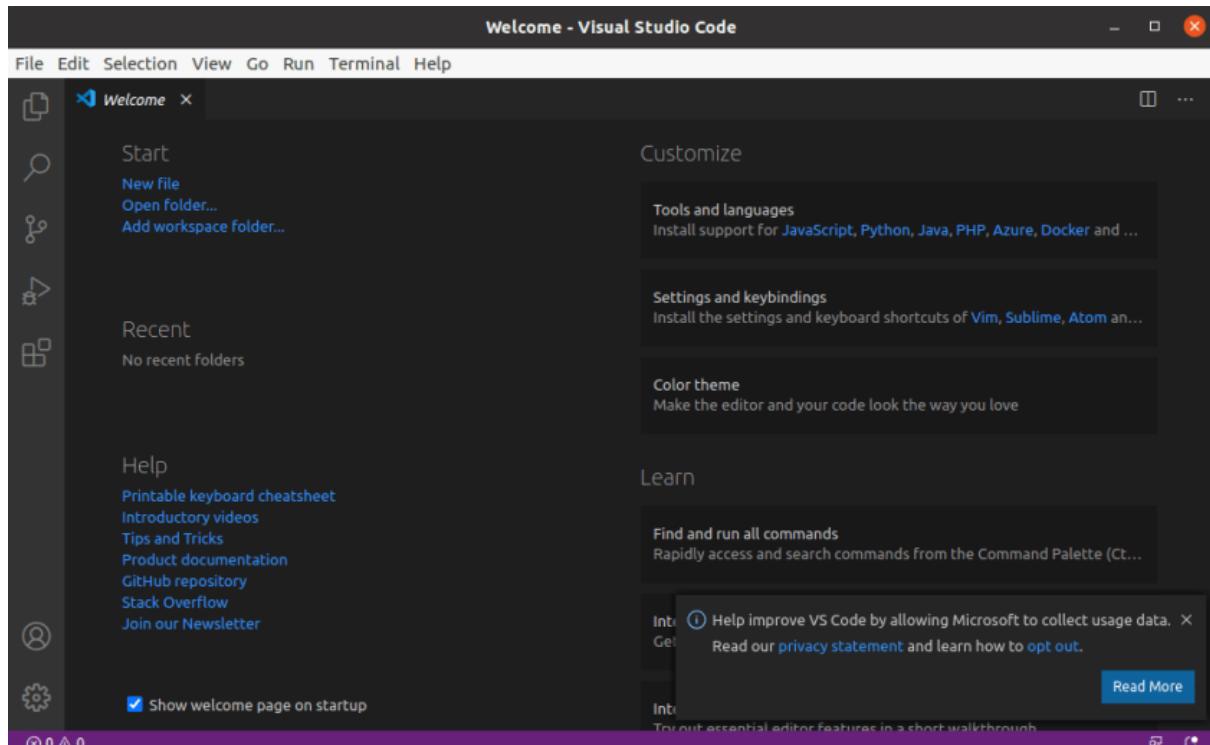
To install it, open a Terminal window, navigate to your *Downloads* folder and run the following command to install VS Code.

```
$ cd Downloads  
~/Downloads $ sudo apt install ./code_1.49.1-1600299189_amd64.deb
```

When the installation is finished, VS Code should be available in your applications menu.



You'll be greeted by a Welcome tab with the newest version's released notes.



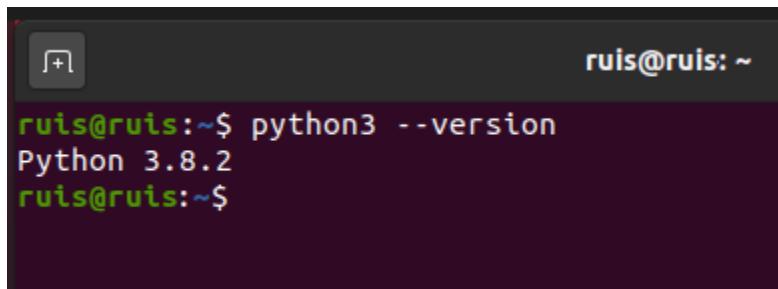
That's it. Visual Studio Code is installed successfully.

Installing Python on Linux Ubuntu

To program the ESP32 and ESP8266 boards with PlatformIO IDE, you need Python 3.5 or a later version installed on your computer. We're using Python 3.8.

Open the Terminal window and check that you already have Python 3 installed.

```
$ python3 --version
```



A screenshot of a terminal window on a dark background. The window title bar says "ruis@ruis: ~". The terminal prompt is "ruis@ruis:~\$". The user has run the command "python3 --version" and the output shows "Python 3.8.2". The prompt then changes to "ruis@ruis:~\$".

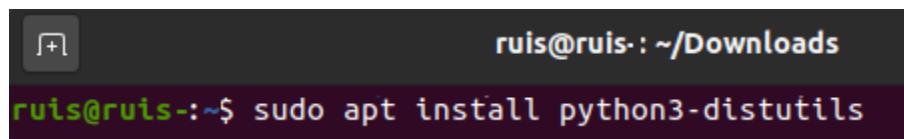
As you can see in the preceding figure, Python 3.8.2 is already installed.

If you don't have Python 3.8.X installed, run the next command to install it:

```
$ sudo apt install python3
```

Whether you already have Python installed or not, you need to run the following command to install Python utilities.

```
$ sudo apt install python3-distutils
```

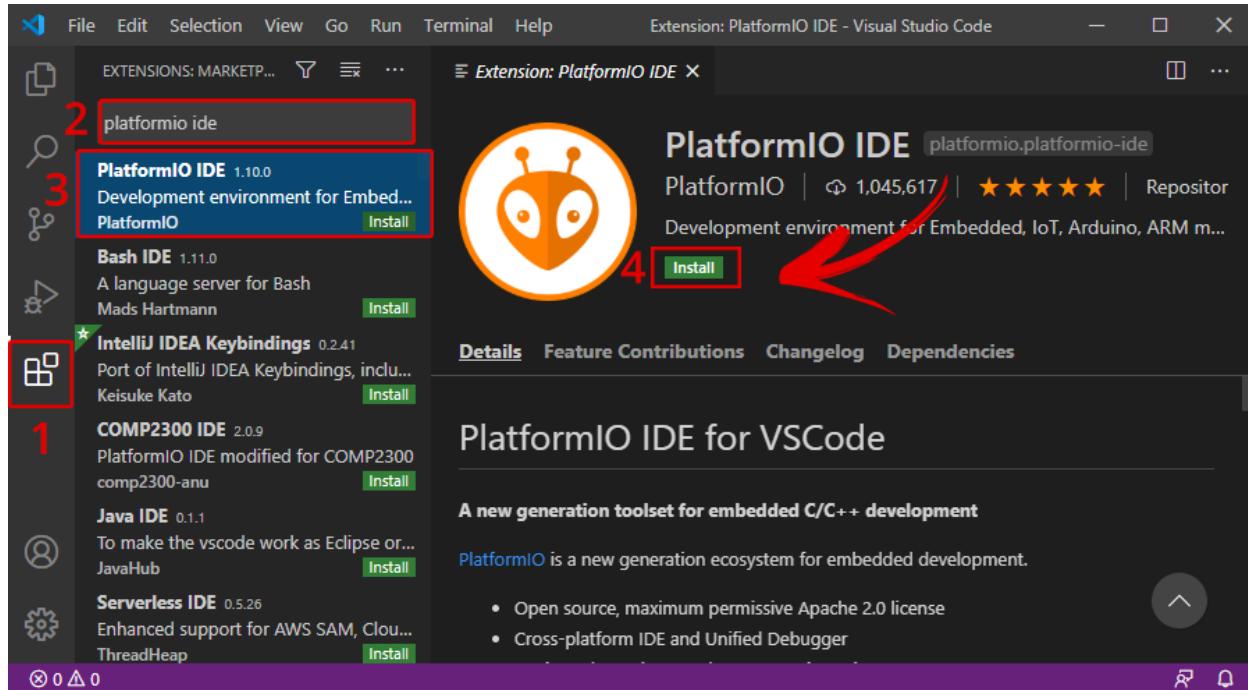


A screenshot of a terminal window on a dark background. The window title bar says "ruis@ruis: ~/Downloads". The terminal prompt is "ruis@ruis:~/Downloads\$". The user has run the command "sudo apt install python3-distutils". The output shows the command being processed.

PlatformIO IDE Extension

At this point, we can install the PlatformIO IDE extension on VS Code. Open VS Code and click on the **Extensions** icon or click **Ctrl+Shift+X** to open the **Extensions** tab.

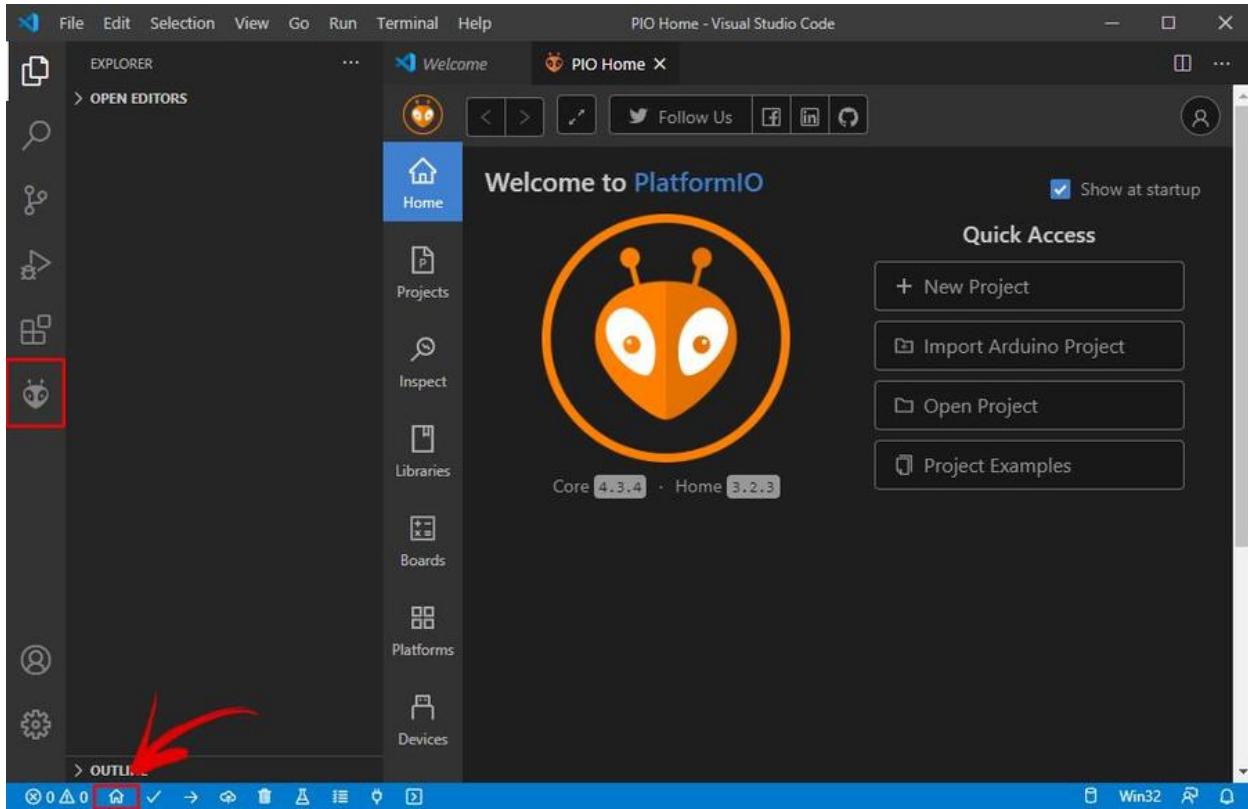
Search for "PlatformIO IDE". Select the first option and click the **Install** button. The installation may take some time.



After installing, make sure that the PlatformIO IDE extension is enabled, as shown below.



After that, the PlatformIO IDE (PIO) icon should show up on the left sidebar as well as a **Home** icon that redirects you to **PlatformIO Home**.



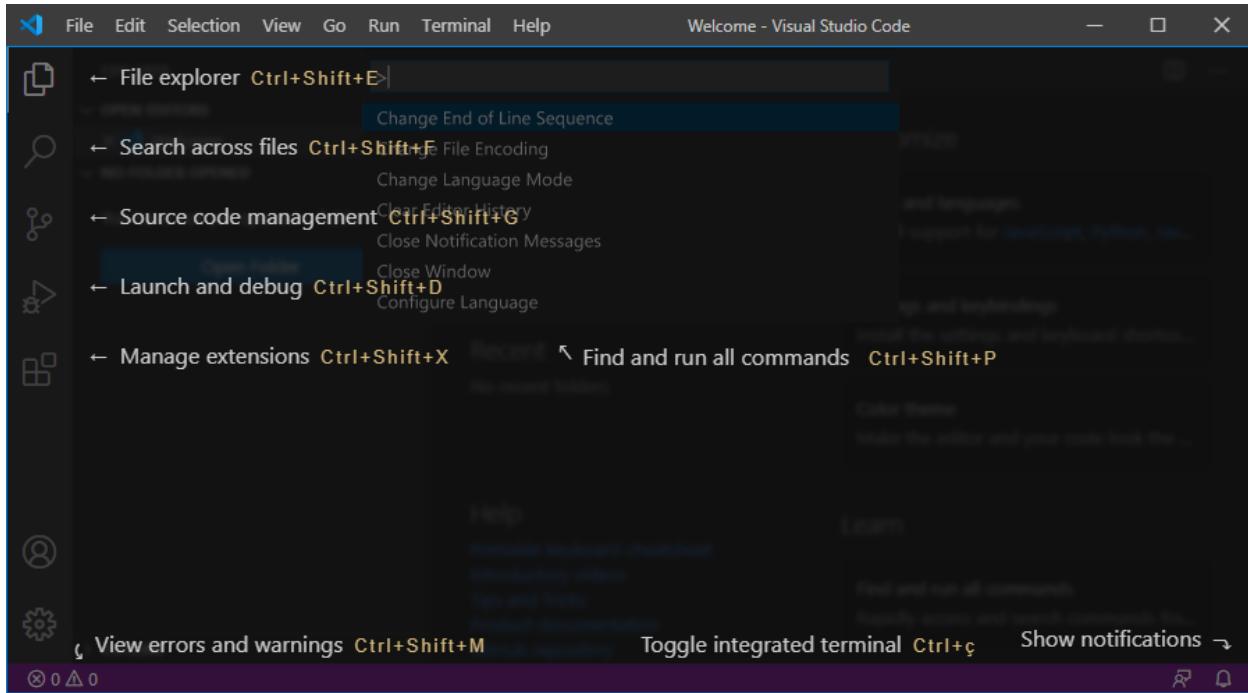
That's it, the PlatformIO IDE extension was added to VS Code.

If you don't see the PIO icon and the quick tools at the bottom, you may need to restart VS Code for the changes to take effect.

Either way, we recommend restart VS Code before proceeding.

Visual Studio Quick Interface Overview

The next image illustrates the meaning of each icon on the left and bottom sidebars and respective shortcuts:



- File explorer
- Search across files
- Source code management (using gist)
- Launch and debug your code
- Manage extensions

To see the previous screen, go to **Help > Welcome > Interface overview**.

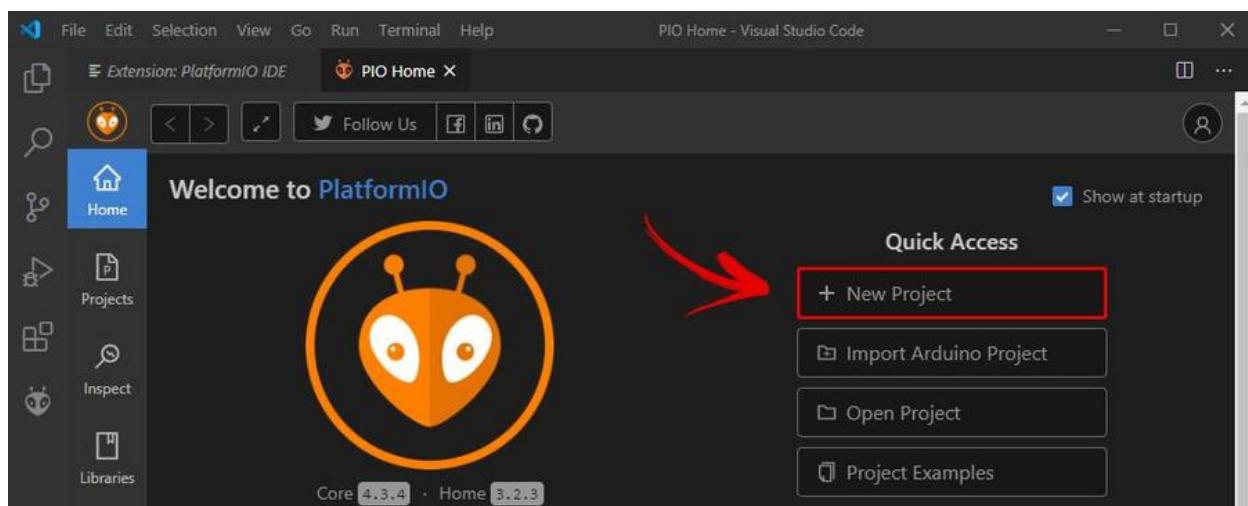
Additionally, you can press **Ctrl+Shift+P** or go to **View > Command Palette...** to show all the available commands. If you're searching for a command and don't know where it is or its shortcut, you just need to go to the Command Palette and search for it.

PlatformIO Overview

For you to get an overview of how PlatformIO works on VS code, we'll show you how to create, save and upload a "*Blinking LED*" sketch to your ESP32 or ESP8266 board.

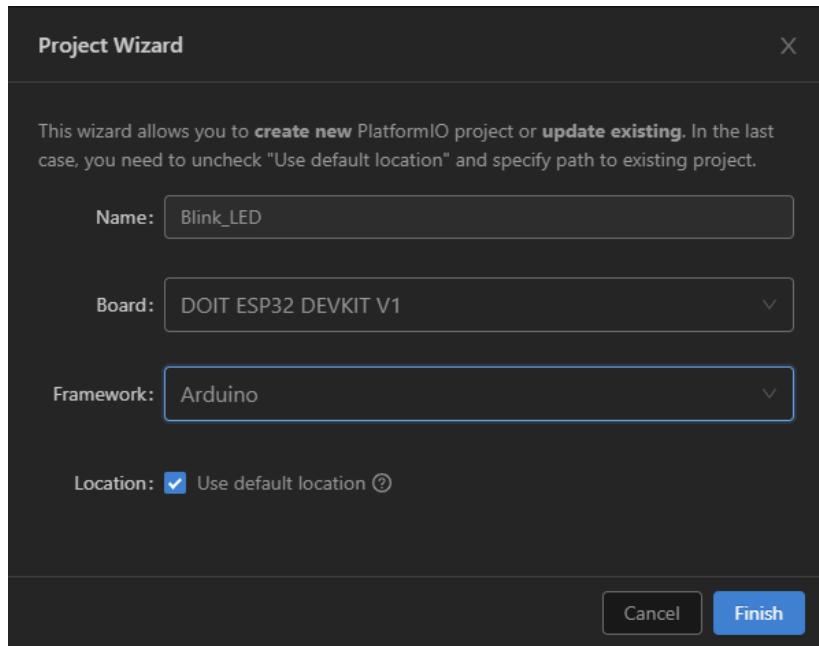
Create a New Project

On VS Code, click on the PlatformIO **Home** icon. Click on **+ New Project** to start a new project.

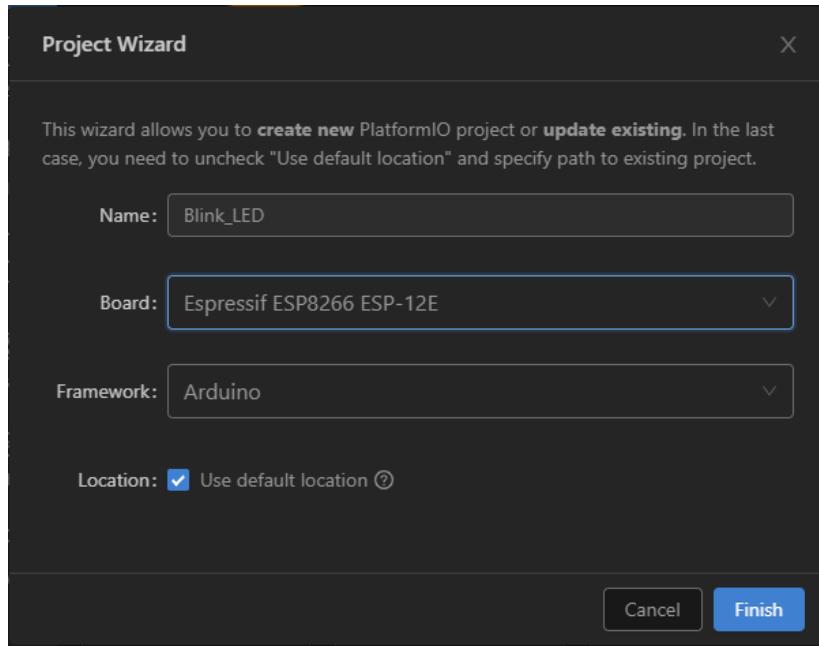


Give your project a name (for example *Blink_LED*) and select the board you're using. In our case, we're using the DOIT ESP32 DEVKIT V1. The Framework should be **"Arduino"** to use the Arduino core.

You can choose the default location to save your project or a custom location. The default location is in this path *Documents > PlatformIO > Projects*. For this test, you can use the default location. Finally, press the **"Finish"** button.



For this example, we'll be using the DOIT ESP32 DEVKIT board. If you are using an ESP8266 NodeMCU board, the process is very similar. You just need to select your ESP8266 board:



If this is your first time using VS Code with the ESP32 or ESP8266, VS Code will download additional data about the selected board after clicking the **Finish** button.

The *Blink_LED* project should be accessible from the **Explorer** tab.



VS Code and PlatformIO have a folder structure that is different from the standard *.ino* project. If you click on the **Explorer** tab, you'll see all the files it created under your project folder. It may seem like a lot of files to work with. But, don't worry. Usually, you'll just need to work with one or two of those files.

WARNING: you shouldn't delete, modify or move the folders. Otherwise, you will no longer be able to compile your project using PlatformIO.

PlatformIO Tasks

After creating a PlatformIO project, the PlatformIO commands will show up in a blue bar at the bottom.



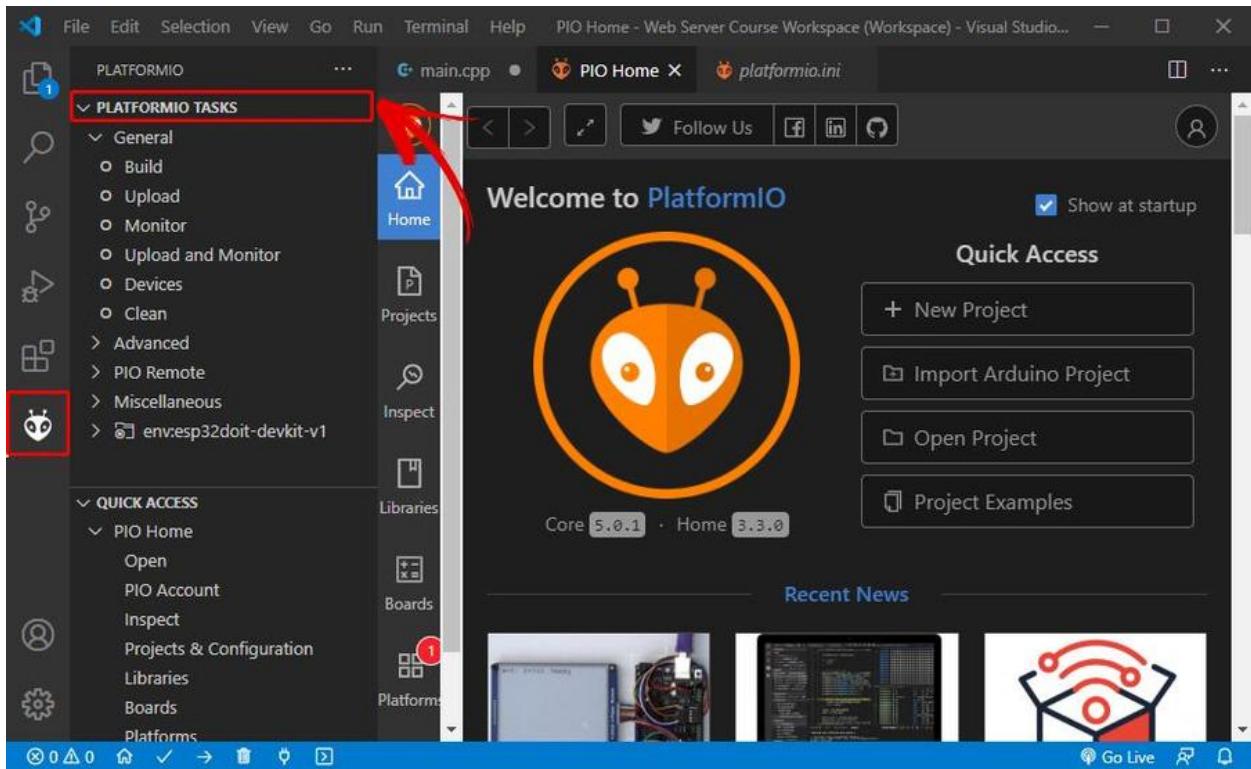
Here's the meaning of each icon from left to right:

- PlatformIO Home
- Build/Compile
- Upload
- Clean
- Serial Monitor
- New Terminal

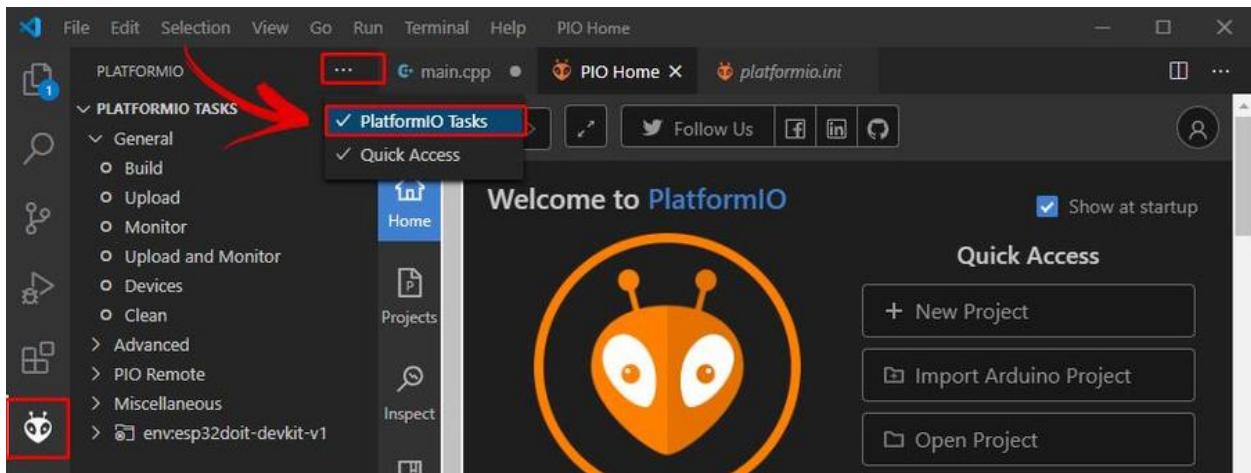
The task bar only shows up when you are working on a PlatformIO project. If no PlatformIO project is open, the blue bar doesn't show up by default in VS Code.

If you hover your mouse over the icons, it will show what each icon does.

Alternatively, you can also click on the PIO icon to see all the PlatformIO tasks.

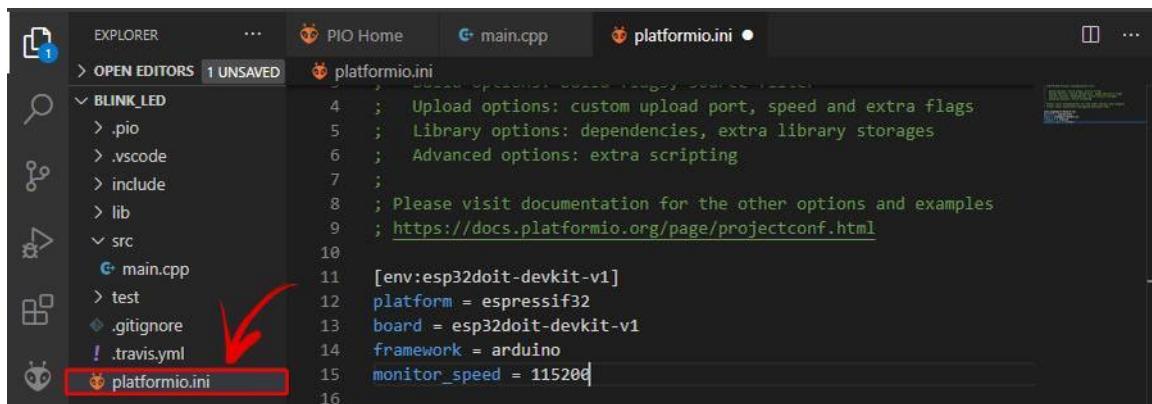


If the tasks don't show up on your IDE when you click the icon, you may need to click on the three-dot icon at the top and enable PlatformIO tasks, as shown below.



platformio.ini file

The platformio.ini file is the PlatformIO Configuration File for your project. It shows the platform, board, and framework for your project. You can also add other configurations like libraries to be included, upload options, changing the Serial Monitor baud rate, and other configurations.



- `platform`: which corresponds to the SoC (system on a chip) used by the board.
- `board`: the development board you're using.
- `framework`: the software environment that will run the project code.

With the ESP32 and ESP8266, if you want to use a baud rate of 115200 in your Serial Monitor, you just need to add the following line to your platformio.ini file.

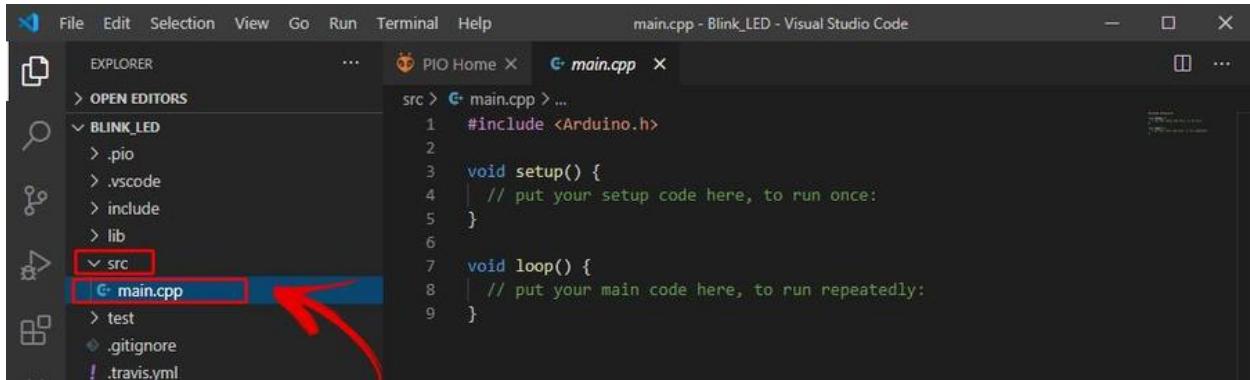
```
monitor_speed = 115200
```

After that, make sure you save the changes made to the file by pressing **Ctrl+S**.

In this file, you can also add libraries that you want to include in your project. PlatformIO will download the libraries and all their dependencies automatically. We'll take a look at this subject later.

src folder

The *src* folder is your working folder. Under the *src* folder, there's a *main.cpp* file. That's where you write your code. Click on that file. The structure of an Arduino program should open with the `setup()` and `loop()` functions.



In PlatformIO, all your Arduino sketches should start with `#include <Arduino.h>`.

Uploading Code using PlatformIO

Copy the following code to your *main.cpp* file.

```
#include <Arduino.h>

#define LED 2

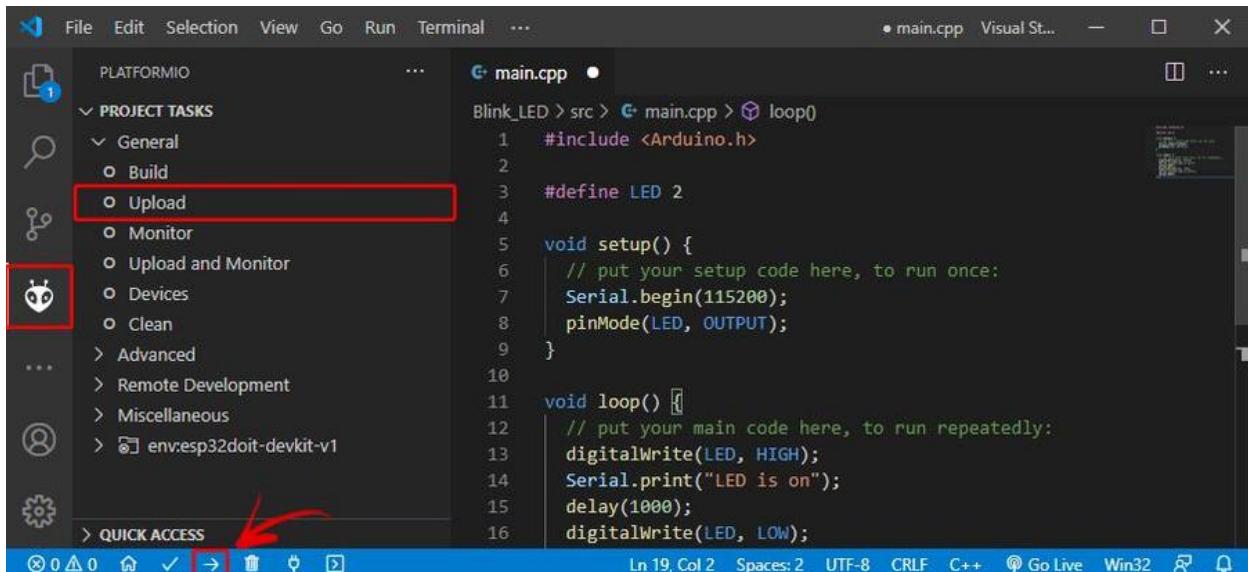
void setup() {
    // put your setup code here, to run once:
    Serial.begin(115200);
    pinMode(LED, OUTPUT);
}

void loop() {
    // put your main code here, to run repeatedly:
    digitalWrite(LED, HIGH);
    Serial.println("LED is on");
    delay(1000);
    digitalWrite(LED, LOW);
    Serial.println("LED is off");
    delay(1000);
}
```

This code blinks the on-board LED every second. It works for the ESP32 and ESP8266 boards (both have the on-board LED connected to GPIO 2).

We recommend that you copy this code manually to see the autocomplete and other interesting features of the IDE in action. Additionally, if you have a syntax error somewhere in your program, it underlines it in red even before compiling.

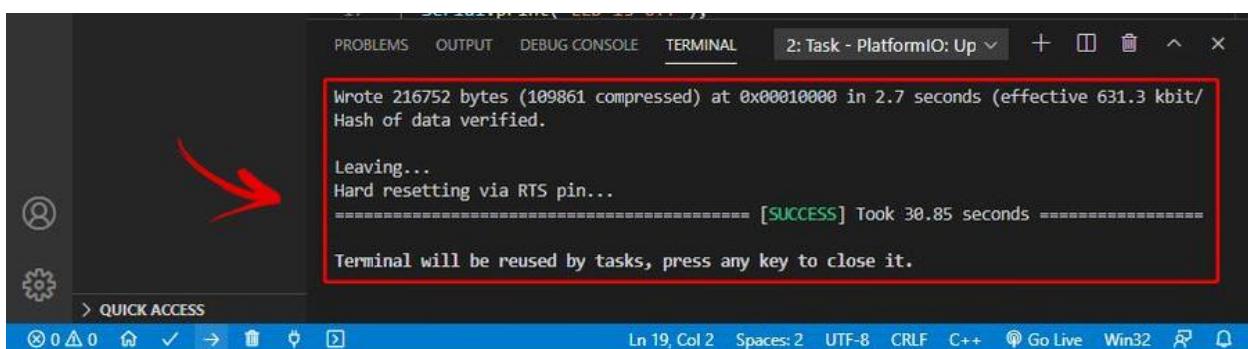
Press **Ctrl+S** or go to **File > Save** to save the file. Now, you can click on the **Upload** icon to compile and upload the code. Alternatively, you can go to the **PIO Project Tasks** menu and select **Upload**.



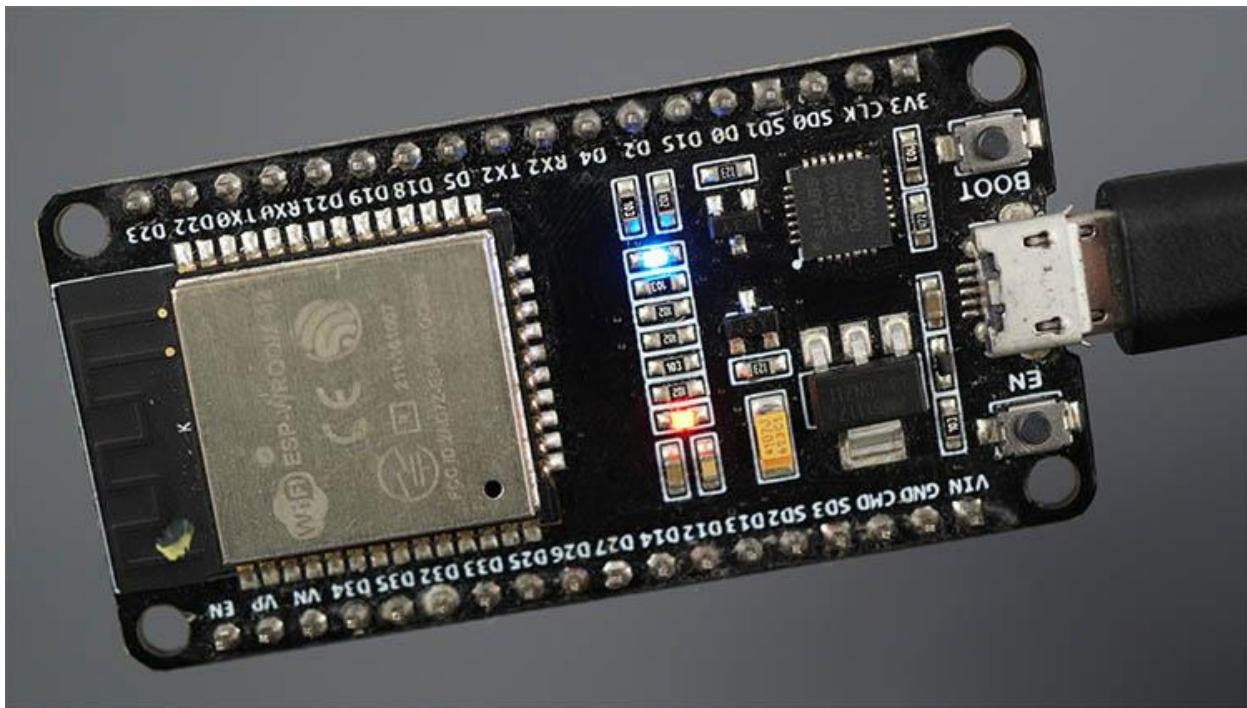
Important: make sure you close all programs that might be using the ESP serial port.

This is, make sure you don't have an Arduino IDE window opened at the same time.

If the code is successfully uploaded, you should get the following message.



After uploading the code, the ESP32 or ESP8266 should be blinking its on-board LED every second.



Now, click on the Serial Monitor icon, and you should see the current LED state.

```
main.cpp
```

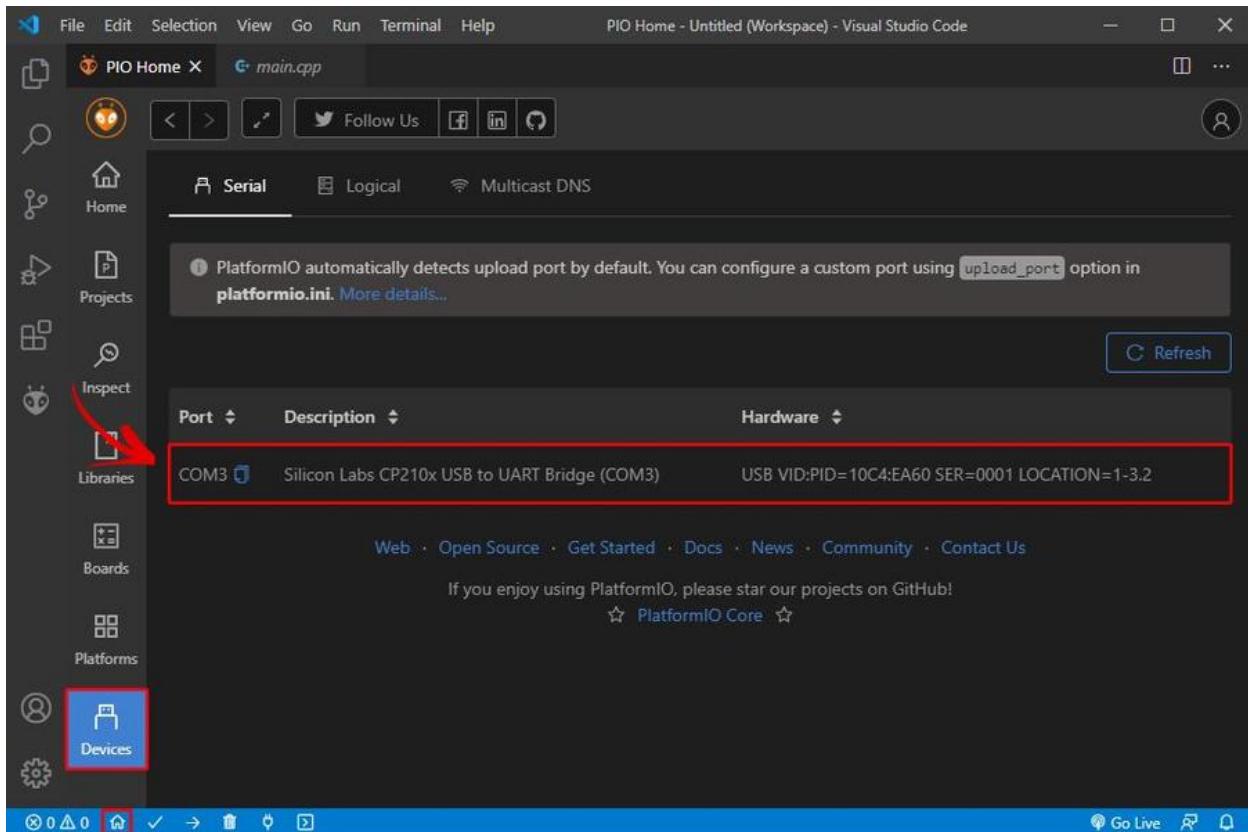
```
#include <Arduino.h>
#define LED 2
void setup() {
    // put your setup code here, to run once:
    Serial.begin(115200);
```

```
LED is on
LED is off
LED is on
```

Note: if you don't see the Terminal window, go to the **Terminal > New Terminal**.

Detect COM Port

PlatformIO will automatically detect the port your board is connected to. To check the connected devices, you can go to the **PIO Home** and click the **Devices** icon.



Installing Drivers

If you don't see your ESP's COM port available, it means you don't have the drivers installed. Take a closer look at the chip next to the voltage regulator on the board and check its name. The ESP32 DEVKIT V1 DOIT board uses the CP2102 chip.

Go to Google and search for your particular chip to find the drivers and install them in your operating system.

You can download the CP2102 drivers on the [Silicon Labs](#) website.

CP210x USB to UART Bridge VCP Drivers

The CP210x USB to UART Bridge Virtual COM Port (VCP) drivers are required for device operation as a Virtual COM Port to facilitate host communication with CP210x products. These devices can also interface to a host using the direct access driver. These drivers are static examples detailed in application note 197: The Serial Communications Guide for the CP210x, download an example below:

[AN197: The Serial Communications Guide for the CP210x](#)

Download Software

The CP210x Manufacturing DLL and Runtime DLL have been updated and must be used with v6.0 and later of the CP210x Windows VCP Driver. Application Note Software downloads affected are AN144SW.zip, AN205SW.zip and AN223SW.zip. If you are using a 5.x driver and need support you can download archived Application Note Software.

[Legacy OS software and driver package download links and support information >](#)

Download for Windows 10 Universal (v10.1.1)

Platform	Software	Release Notes
 Windows 10 Universal	Download VCP (2.3 MB)	Download VCP Revision History

After they are installed, restart the VS Code, and you should see the COM port in the **Devices** menu.

Troubleshooting

If you get the following error while uploading the code: "**Failed to connect to ESP32: Timed out waiting for packet header**" it usually means that your board is not in flashing mode when you're uploading the code.

When this happens, you need to press the ESP32 on-board BOOT button when you start seeing many dots in the debugging window.

For more information about this issue, you can follow this guide:

- [\[SOLVED\] Failed to connect to ESP32: Timed out waiting for packet header](#)

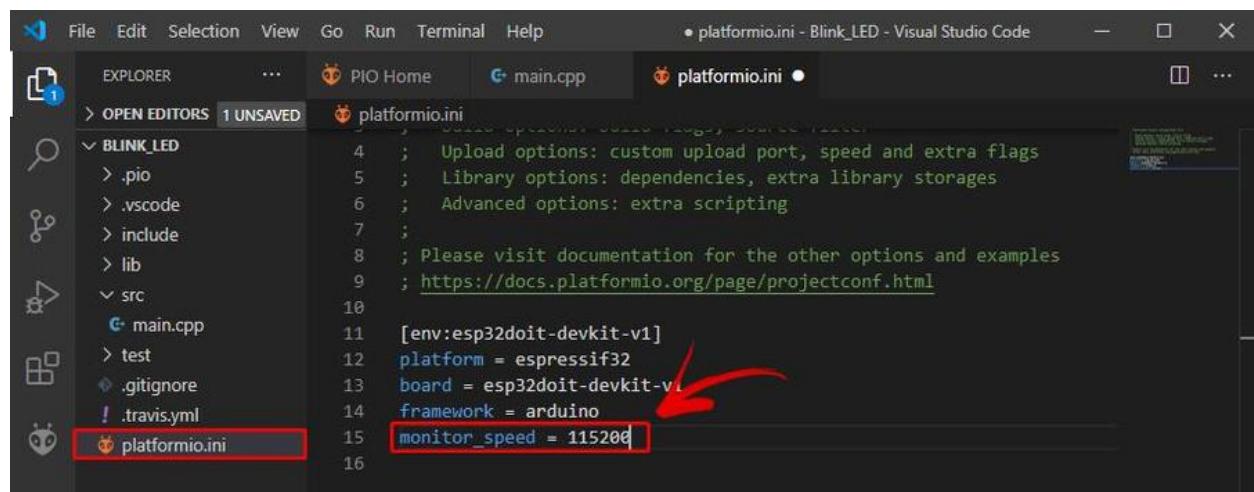
Changing the Serial Monitor Baud Rate - PlatformIO

The default baud rate used by PlatformIO is 9600. However, it is possible to set a different value as mentioned previously. On the **File Explorer**, under your project folder, open the *platformio.ini* file and add the following line:

```
monitor_speed = baud_rate
```

For example:

```
monitor_speed = 115200
```



After that, save your *platformio.ini* file.

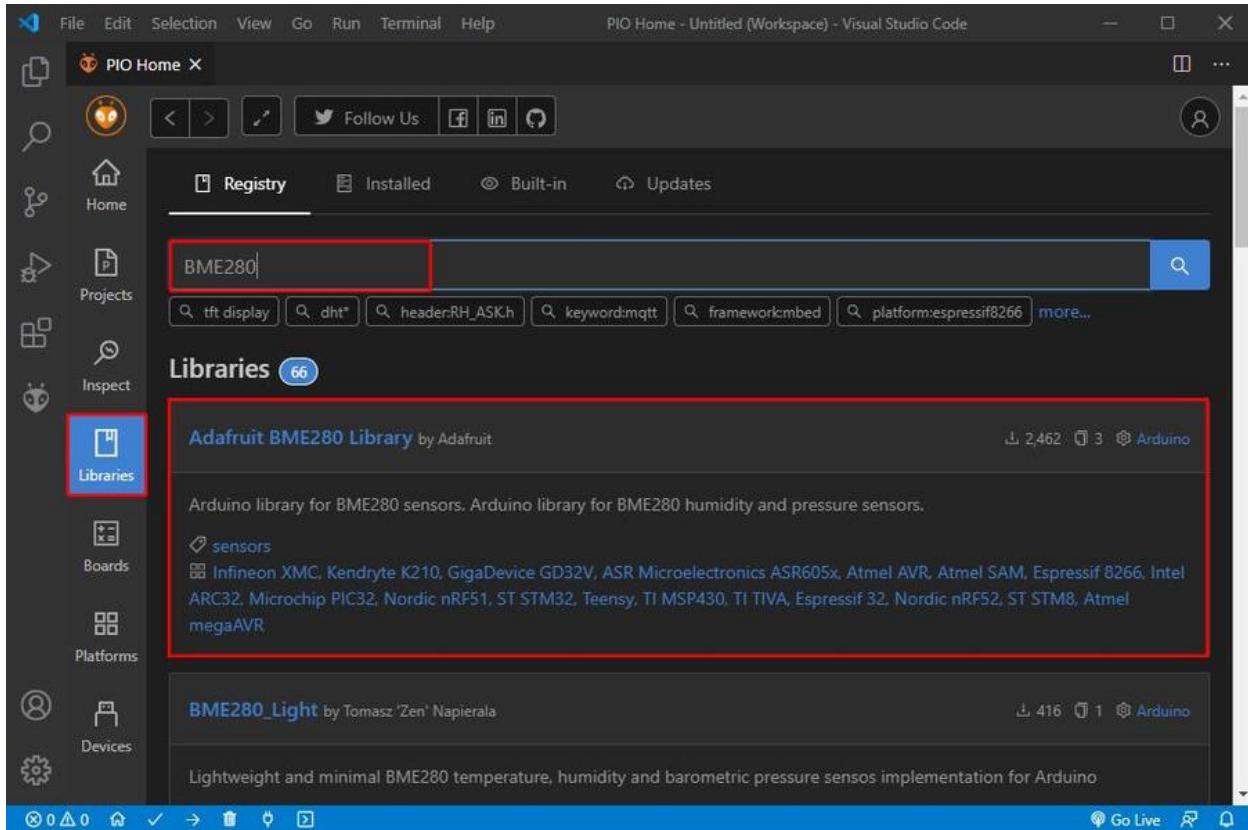
Installing ESP32/ESP8266 Libraries on PlatformIO

PlatformIO has a built-in powerful Library Manager. It allows you to set custom dependencies per project in the Configuration File *platformio.ini* using `lib_deps`. This tells PlatformIO to automatically download the library and all its dependencies when you save the configuration file or compile your project.

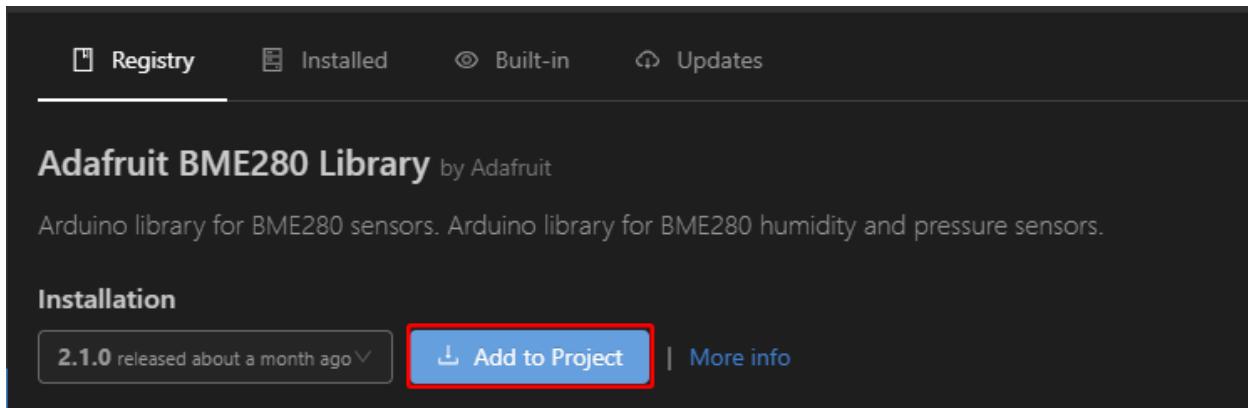
Follow the following procedure if you need to install libraries in PlatformIO.

Click the **Home** icon to go to PlatformIO **Home**. Select the **Libraries** icon on the left menu.

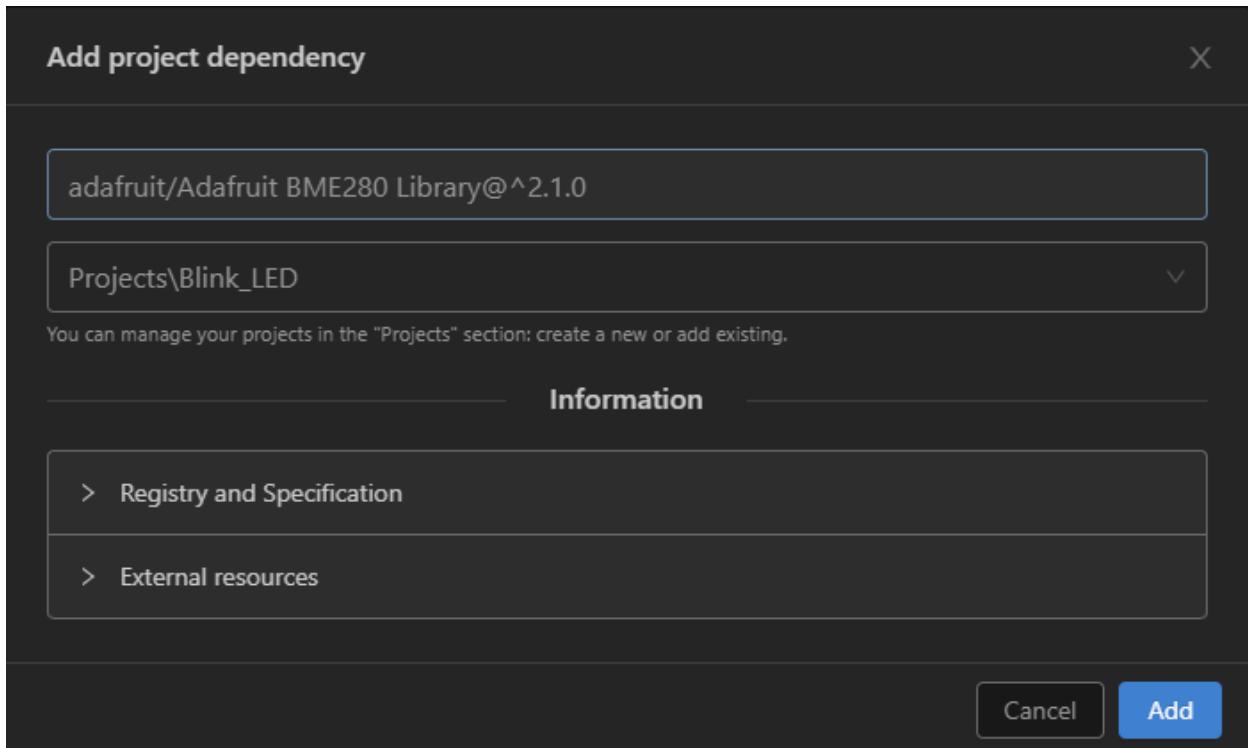
Search for the library you want to install—for example, **Adafruit_BME280**.



Select the library you want to include in your project. Then, click **Add to Project**.



Select the project where you want to use the library.



This will add the library identifier using the `lib_deps` directive on the `platformio.ini` file. If you open your project's `platformio.ini` file, it should look as shown in the following image.

```

[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = adafruit/Adafruit BME280 Library@^2.1.0

```

Alternatively, on the library window, you can select the **Installation** tab and scroll until you see the library's identifier. You can choose any of those identifiers

depending on the options you want to use. The library's identifiers are highlighted in red in the following image.

Adafruit BME280 Library by Adafruit

Arduino library for BME280 sensors. Arduino library for BME280 humidity and pressure sensors.

Installation

2.1.0 released about a month ago | Add to Project | More info

Examples Installation Headers Changelog

Library Dependencies platformio.ini

The PlatformIO Registry is fully compatible with [Semantic Versioning](#) and its "version" scheme <major>.<minor>.<patch>. You can declare library dependencies in "platformio.ini" configuration file using [lib_deps](#) option.

```
; platformio.ini - project configuration file

[env:my_build_env]
platform = infineonxmc
framework = arduino
lib_deps =
    # RECOMMENDED
    # Accept new functionality in a backwards compatible manner and patches
    adafruit/Adafruit_BME280_Library @ ^2.1.0
    # Accept only backwards compatible bug fixes
    # (any version with the same major and minor versions, and an equal or greater patch version)
    adafruit/Adafruit_BME280_Library @ ~2.1.0
    # The exact version
    adafruit/Adafruit_BME280_Library @ 2.1.0
```

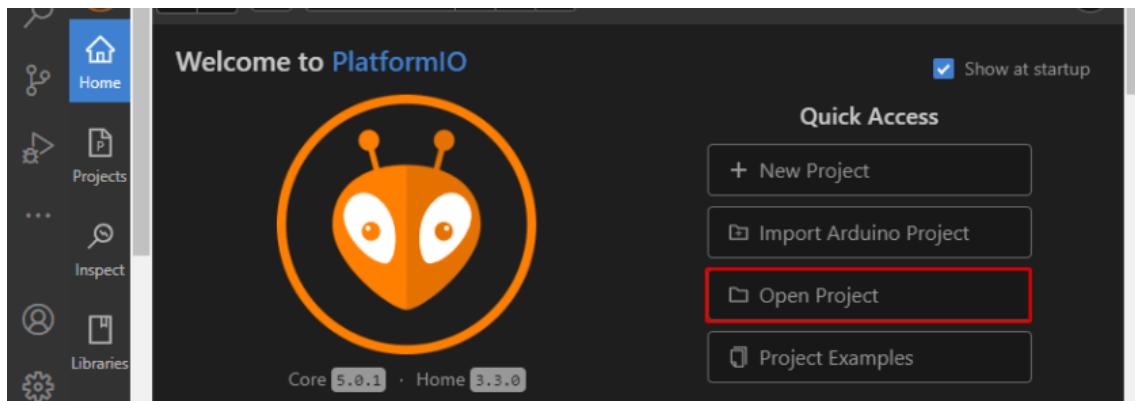
Then, go to the *platformio.ini* file of your project and paste the library identifier into that file, like this:

```
lib_deps = adafruit/Adafruit_BME280_Library@^2.1.0
```

If you need multiple libraries, you can separate their name by a comma or put them on different lines.

Open a Project Folder

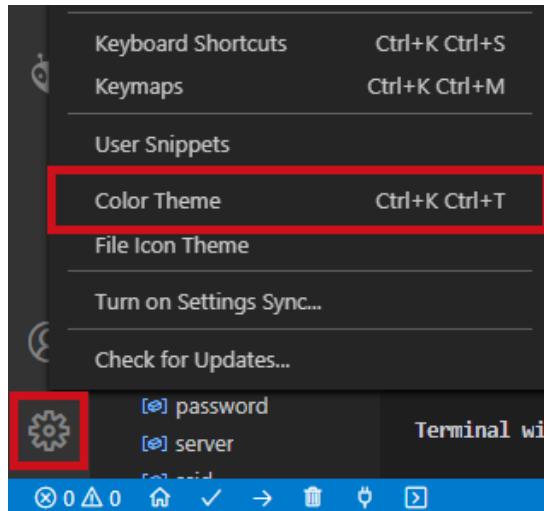
To open an existing project folder on PlatformIO, open VS Code. Then, go to PlatformIO Home and click **Open Project**. Navigate through the files and select your project folder.



PlatformIO will open all the files within the project folder.

VS Code Color Themes

VS Code lets you choose between different color themes. Go to the **Manage** icon and select **Color Theme**.



You can then select from several different light and dark themes.

Shortcuts' List

For a complete list of VS Code shortcuts for Windows, Mac OS X or Linux, use the following link as a reference:

- [VS Code Keyboard Shortcuts Reference](#).

Wrapping Up

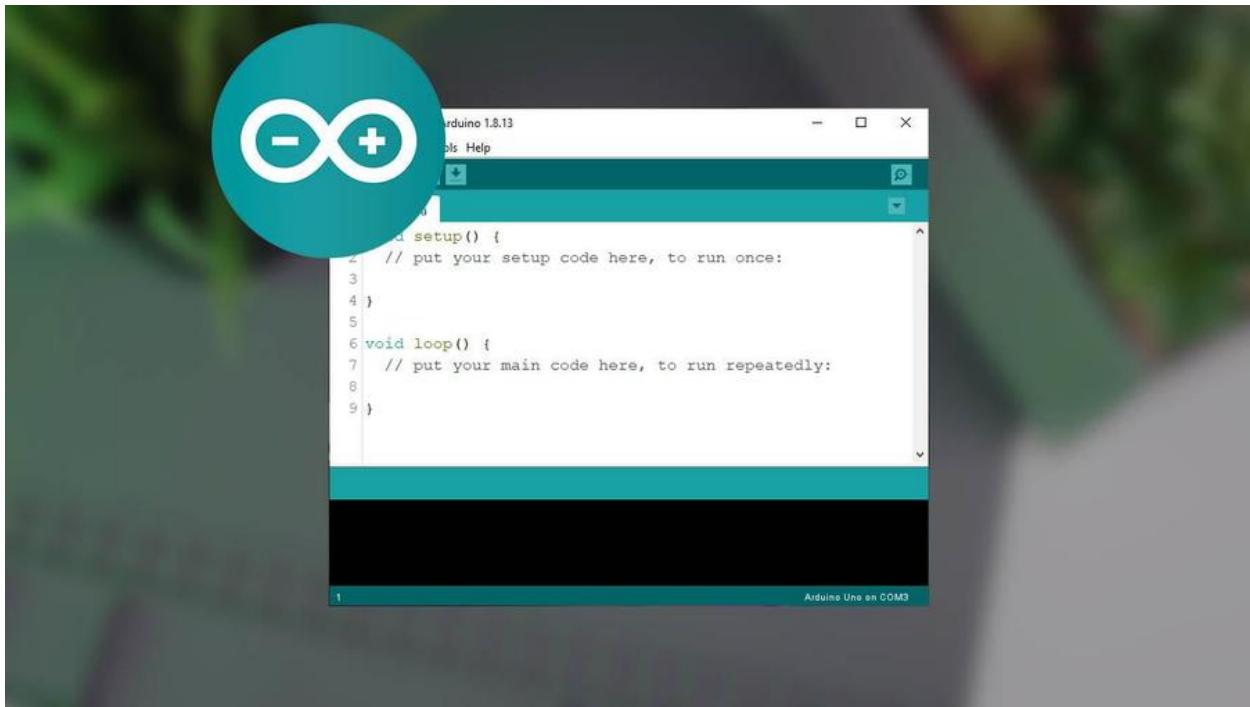
In this Unit, you installed VS Code on your computer. We'll use VS Code to write the HTML, CSS, and JavaScript files to build web pages for the web server projects.

You also prepared VS Code to work with the ESP32 and ESP8266 boards. VS Code with the PlatformIO extension is an excellent alternative to the classical Arduino IDE, especially when you're working on more advanced sketches for larger applications or when you need to work with multiple files

Here are some of the advantages of using VS Code with PlatformIO over Arduino IDE:

- It detects the COM port your board is connected to automatically;
- VS Code IntelliSense: Auto-Complete. IntelliSense code completion tries to guess what you want to write, displaying the different possibilities and provides insight into the parameters that a function may expect;
- Error Highlights: VS Code + PIO underlines errors in your code before compiling;
- Multiple open tabs: you can have several code tabs open at once;
- You can hide certain parts of the code;
- Advanced code navigation;
- And much more.

Installing Arduino IDE



The Arduino IDE works great for small applications, but for more advanced projects with more than 200 lines of code or when you need different files and other advanced features like auto-completion and error checking, VS Code with the PlatformIO IDE extension may be a better alternative.

Throughout this eBook, we'll program the ESP32 and ESP8266 boards using VS Code with the PlatformIO extension, and that's what we recommend you using.

However, if you find an overwhelming task getting used to this new text editor, you can still use Arduino IDE. Additionally, sometimes, it can be handier to use Arduino IDE for some more minor tasks. So, having Arduino IDE installed on your computer can be helpful.

If you don't want to install Arduino IDE, you can skip this section.

Requirements

You need JAVA installed on your computer. If you don't, go to the following website to download and install the latest version:

- <http://java.com/download>

Downloading Arduino IDE

To download the Arduino IDE, visit the following URL:

- <https://www.arduino.cc/en/Main/Software>

Select your operating system and download the software. For Windows, we recommend downloading the "**Windows ZIP file**".

The screenshot shows the Arduino IDE download page. On the left, there's a teal circular icon with a white 'A' and a plus sign. Next to it, the text "Arduino IDE 1.8.13" is displayed. Below this, a paragraph explains what the IDE does: "The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board." It also links to the "Getting Started" page for installation instructions. On the right, a teal sidebar titled "DOWNLOAD OPTIONS" lists download links for different platforms: Windows (Win 7 and newer, ZIP file), Windows app (Get button), Linux (32 bits, 64 bits, ARM 32 bits, ARM 64 bits), and Mac OS X (10.10 or newer). Release notes and checksums are also provided.

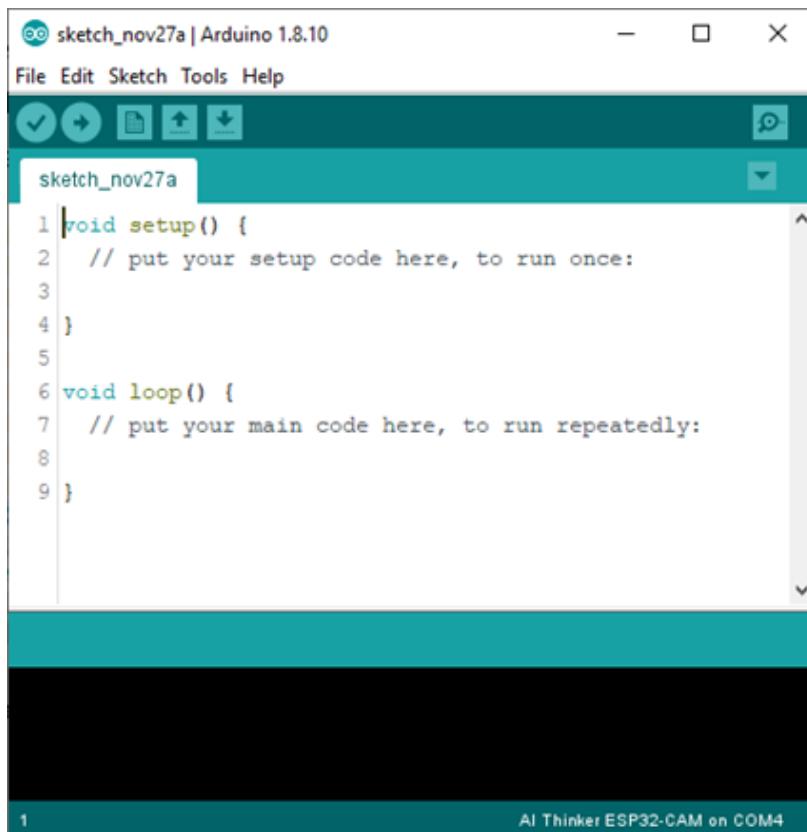
Note: if you have a previous version of the Arduino IDE installed, we recommend upgrading to the most recent version. At the time of writing this Unit, the recent version is Arduino 1.8.13 and that's the one we'll use throughout this eBook.

Installing Arduino IDE

Grab the folder you've just downloaded and unzip it. Run the file highlighted below.

examples	11/28/2019 3:16 PM	File folder
hardware	11/28/2019 3:16 PM	File folder
java	11/28/2019 3:17 PM	File folder
lib	11/28/2019 3:17 PM	File folder
libraries	11/28/2019 3:17 PM	File folder
reference	11/28/2019 3:17 PM	File folder
tools	11/28/2019 3:18 PM	File folder
tools-builder	11/28/2019 3:18 PM	File folder
arduino.exe	11/28/2019 3:16 PM	Application 395 KB
arduino.l4j.ini	11/28/2019 3:16 PM	Configuration sett... 1 KB
arduino_debug.exe	11/28/2019 3:16 PM	Application 393 KB
arduino_debug.l4j.ini	11/28/2019 3:16 PM	Configuration sett... 1 KB

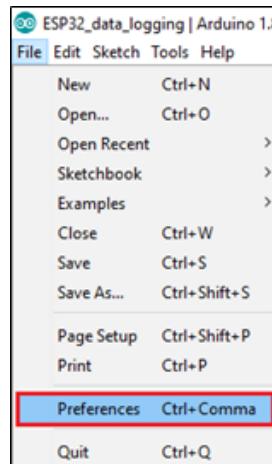
The Arduino IDE window should open.



Installing the ESP32/ESP8266 add-ons for Arduino IDE

To program the ESP32 and ESP8266 boards using Arduino IDE, you need to install the ESP32 and ESP8266 add-on. Follow the next steps to install them.

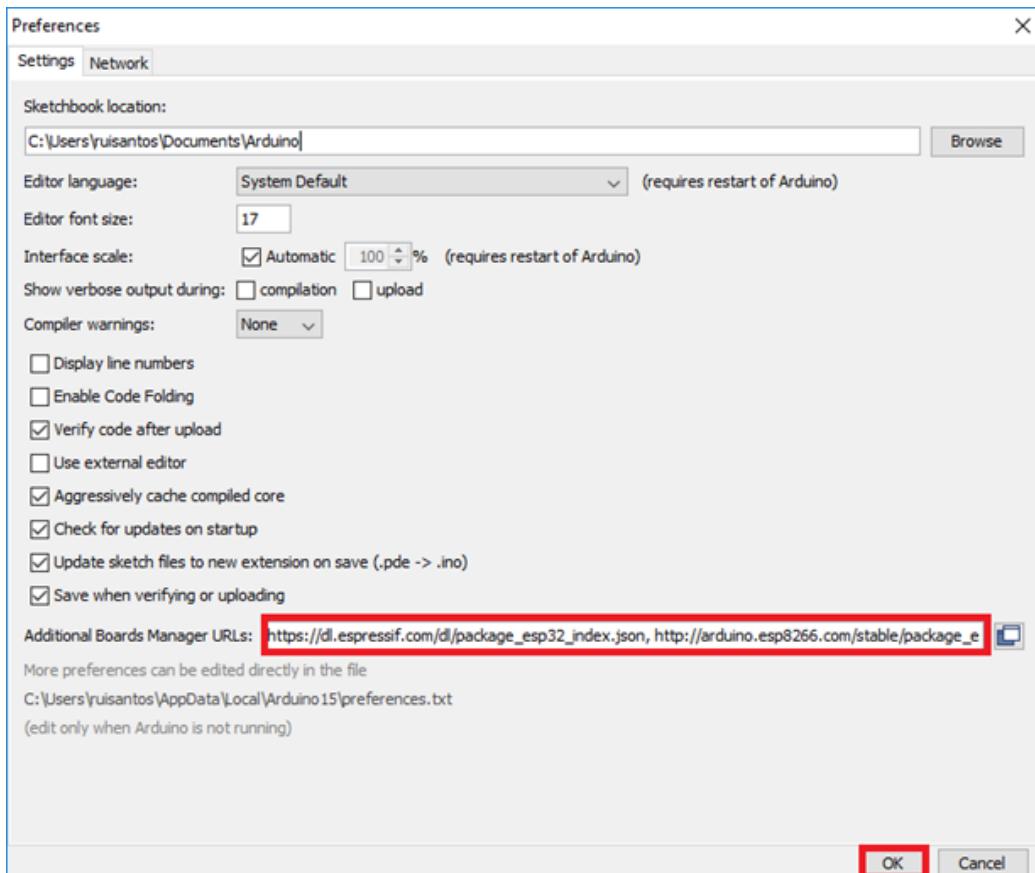
Open the **Preferences** window in the Arduino IDE. Go to **File > Preferences**:



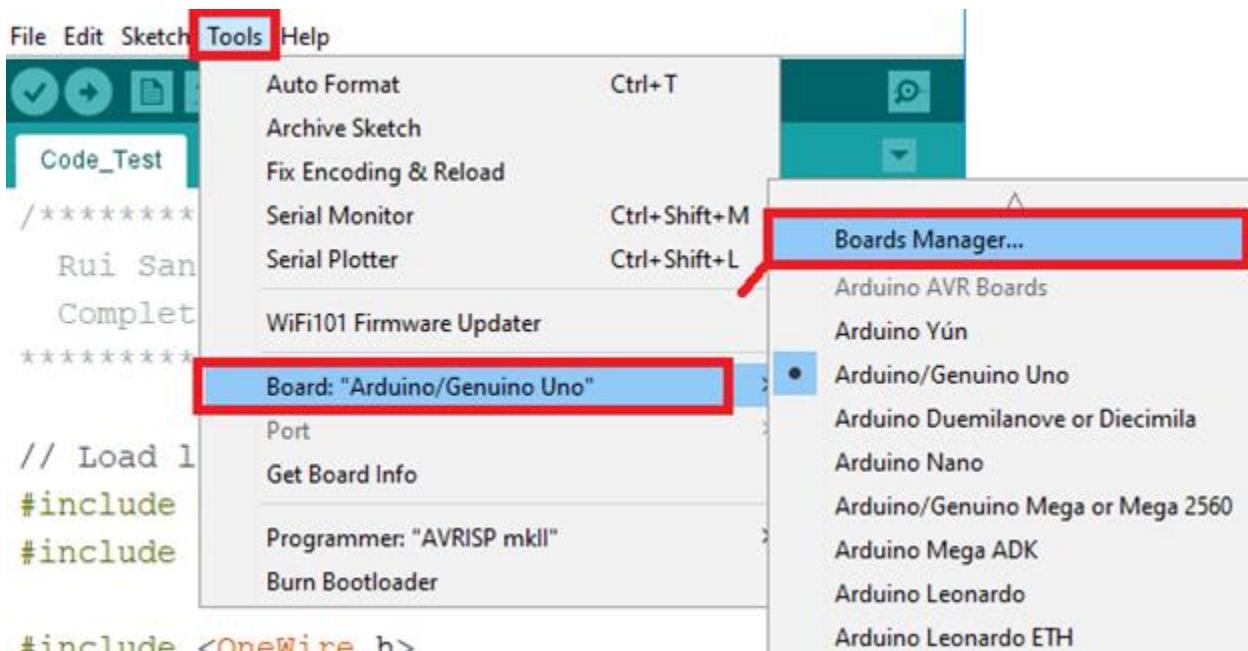
Enter the following into the "Additional Board Manager URLs" field.

**https://dl.espressif.com/dl/package_esp32_index.json,
https://arduino.esp8266.com/stable/package_esp8266com_index.json**

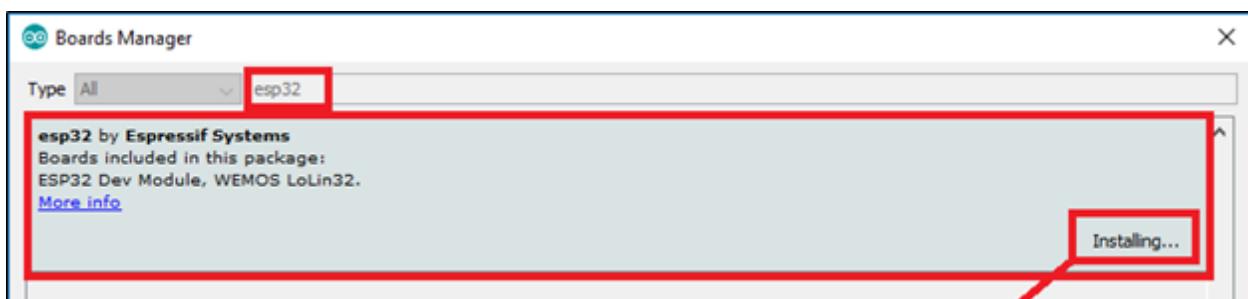
See figure below. Then, click the "**OK**" button.



Open the **Boards Manager**. Go to **Tools > Board > Boards Manager...**



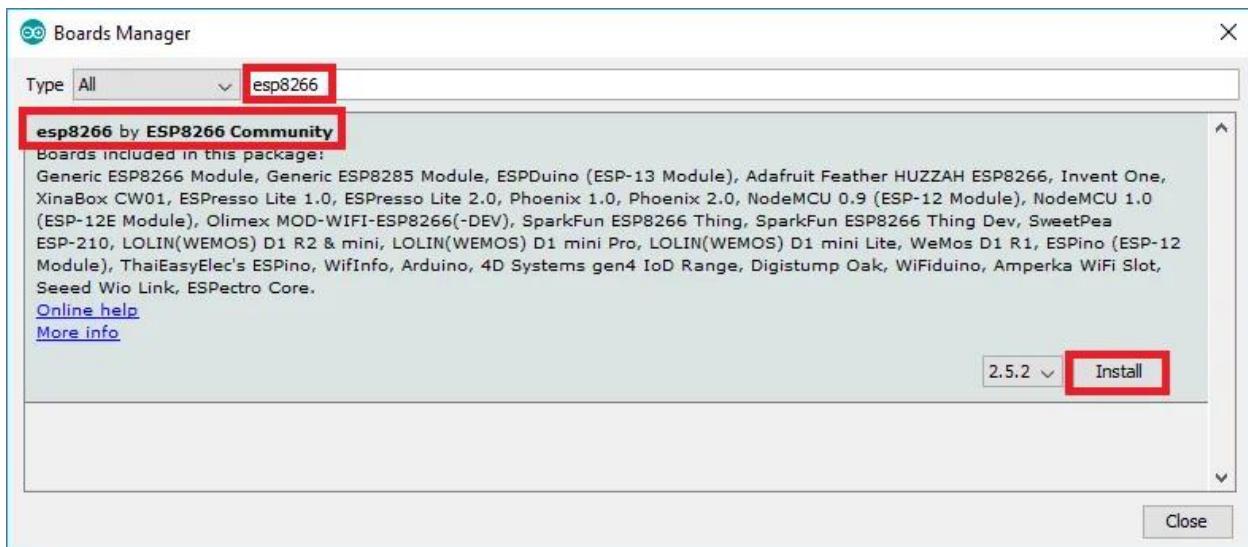
Search for ESP32 and install the "**ESP32 by Espressif Systems**":



That's it. It will be installed after a few seconds.

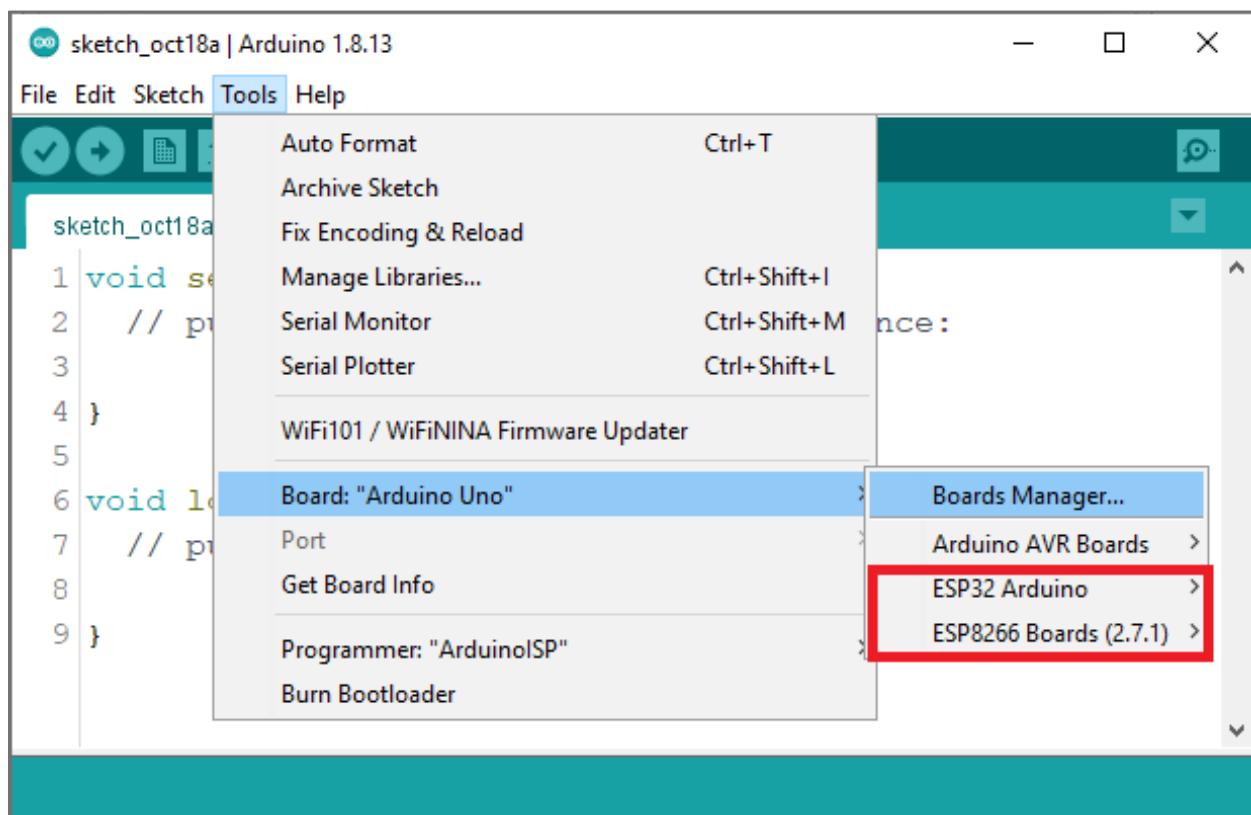


Still on the Boards Manager menu, search for "esp8266" and press the install button for the "**ESP8266 by ESP8266 Community**".



After this, restart your Arduino IDE.

Then, go to **Tools** > **Board** and check that you have ESP32 and ESP8266 boards available.



Update the ESP32/ESP8266 Boards Add-On

Once in a while, it's a good idea to check if you have the latest version of the ESP32 or ESP8266 Boards add-on installed.

You just need to go to **Tools > Board > Boards Manager**, search for ESP32 or ESP8266, and check the version that you have installed. If there is a more recent version available, select that version to install it.

SPIFFS Plugin

If you want to follow this eBook using Arduino IDE, you also need to install the SPIFFS or LittleFS plugin.

Throughout this eBook, we'll save the HTML, CSS, and other files needed to build the web servers in the ESP32 or ESP8266 Serial Peripheral Interface Flash Filesystem (SPIFFS and LittleFS). SPIFFS/LittleFS is a lightweight filesystem created for microcontrollers with a flash chip connected by an SPI bus. It lets you access the flash memory like you would do in a standard file system on your computer, but it's simpler and more limited. You can read, write, close, and delete files. Using SPIFFS/LittleFS with the ESP32 or ESP8266 boards is especially useful to:

- Create configuration files with settings;
- Save data permanently;
- Create files to save small amounts of data instead of using a microSD card;
- Save HTML, CSS, and JavaScript files to build a web server;
- Save images, figures, and icons;
- And much more.

Installing the Arduino ESP32 Filesystem Uploader

This section shows how to install the ESP32 Filesystem Uploader Plugin in Arduino IDE. If you're using an ESP8266, proceed to the next section.

You can create, save and write files to the ESP32 filesystem by writing the code yourself on the Arduino IDE. This is not very useful because you'd have to type your files' content in the Arduino sketch.

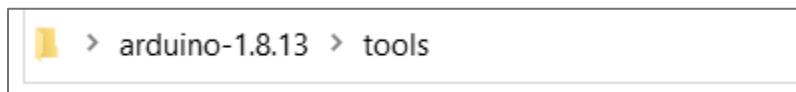
Fortunately, there is a plugin for the Arduino IDE that allows you to upload files directly to the ESP32 filesystem from a folder on your computer. This makes it easy and simple to work with files. Let's install it.

Follow the next steps to install the filesystem uploader:

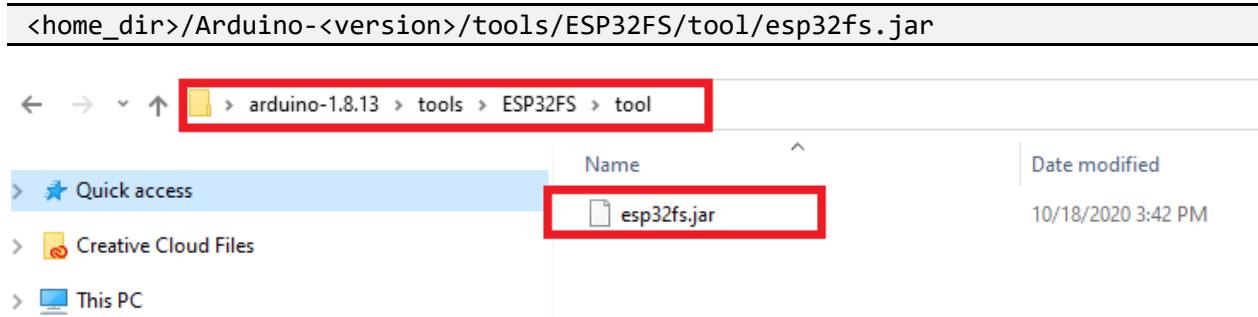
- 1) Go to the [releases page](#) and click the ESP32FS-1.0.zip file to download.



- 2) Go to the Arduino IDE directory, and open the *Tools* folder.



- 3) Unzip the downloaded *.zip* folder to the *tools* folder. You'll get a folder called *ESP32FS-1.0* or similar. Inside that folder you have the *ESP32FS* folder. Cut the *ESP32FS* folder and paste it into the *tools* folder. Delete the *ESP32FS-1.0* folder. Then, you should have the following exact folder structure:

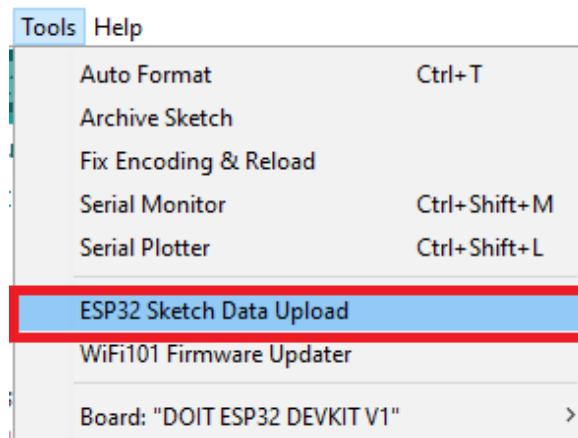


Important: the folder structure should be exactly as shown in the previous figure. Otherwise, it won't work.

On Mac OS X, create the *tools* directory in `~/Documents/Arduino/` and unpack the files there

- 4) Finally, restart your Arduino IDE.

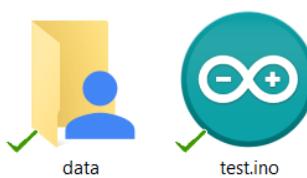
To check if the plugin was successfully installed, open your Arduino IDE. Select your ESP32 board, go to **Tools** and check that you have the option "**ESP32 Sketch Data Upload**".



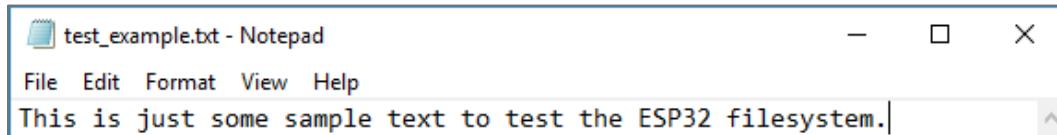
Uploading Files using the Filesystem Uploader (ESP32)

To upload files to the ESP32 filesystem, follow the next instructions.

- 1) Create an Arduino sketch and save it. For demonstration purposes, you can save an empty sketch.
- 2) Then, open the sketch folder. You can go to **Sketch > Show Sketch Folder**.
The folder where your sketch is saved should open.
- 3) Inside that folder, create a new folder called *data*.

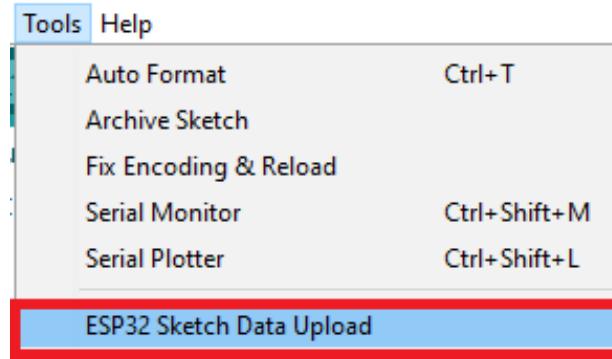


- 4) Inside the *data* folder is where you should put the files you want to save into the ESP32 filesystem. As an example, create a *.txt* file with some text called *test_example*.

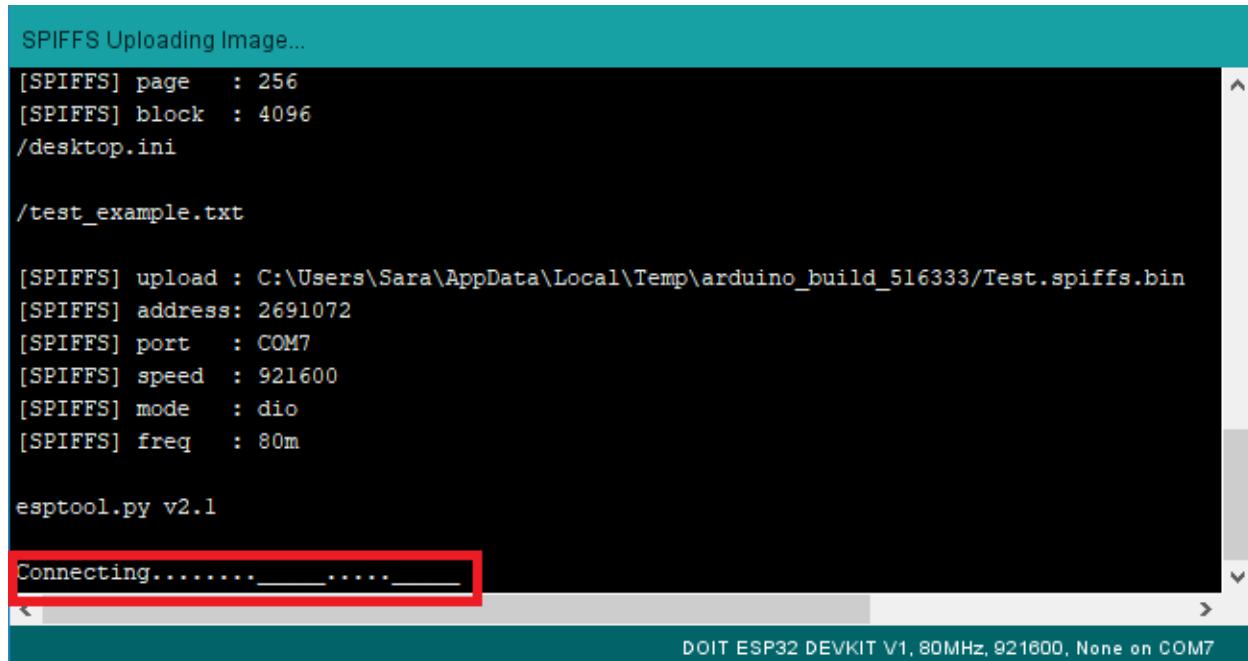


- 5) Then, to upload the files in the Arduino IDE, you just need to go to **Tools > ESP32 Sketch Data Upload**.

Important: ensure the Serial Monitor is closed. Otherwise, it will fail.



Note: in some ESP32 development boards, you need to press the ESP32 on-board BOOT button when you see the "Connecting_____." message.



```
SPIFFS Uploading Image...
[SPIFFS] page   : 256
[SPIFFS] block  : 4096
/desktop.ini

/test_example.txt

[SPIFFS] upload : C:\Users\Sara\AppData\Local\Temp\arduino_build_516333/Test.spiiffs.bin
[SPIFFS] address: 2691072
[SPIFFS] port   : COM7
[SPIFFS] speed  : 921600
[SPIFFS] mode   : dio
[SPIFFS] freq   : 80m

esptool.py v2.1

Connecting....._____.
```

DOIT ESP32 DEVKIT V1, 80MHz, 921600, None on COM7

After a few seconds, you should get the message "**SPIFFS Image Uploaded**". The files were successfully uploaded to the ESP32 filesystem.



```
SPIFFS Image Uploaded
Hash of data verified.

Leaving...
Hard resetting...
```

DOIT ESP32 DEVKIT V1, 80MHz, 921600, None on COM7

Testing the Uploader (ESP32)

Now, let's check if the file was actually saved into the ESP32 filesystem. Upload the following code to your ESP32 board.

```

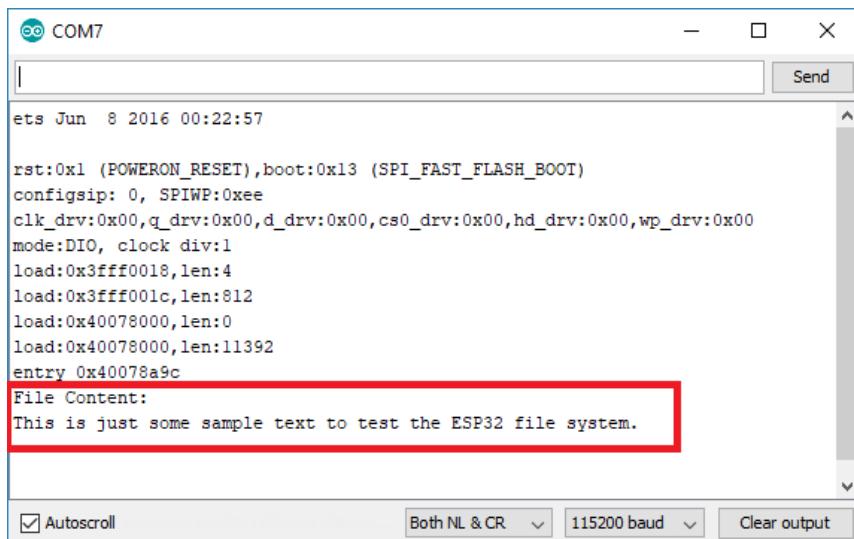
#include "SPIFFS.h"
void setup() {
    Serial.begin(115200);
    if(!SPIFFS.begin(true)){
        Serial.println("An Error has occurred while mounting SPIFFS");
        return;
    }
    File file = SPIFFS.open("/test_example.txt");
    if(!file){
        Serial.println("Failed to open file for reading");
        return;
    }
    Serial.println("File Content:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}
void loop() {
}

```

You can download the code by clicking on the following link:

- https://github.com/RuiSantosdotme/build-web-servers-dl/raw/main/Test_SPIFFS_ESP32.zip

After uploading, open the Serial Monitor at a baud rate of 115200. Press the ESP32 on-board EN (or RST) button. It should print the content of your .txt file on the Serial Monitor.



You've successfully uploaded files to the ESP32 filesystem using the plugin.

Installing the Arduino ESP8266 Filesystem Uploader

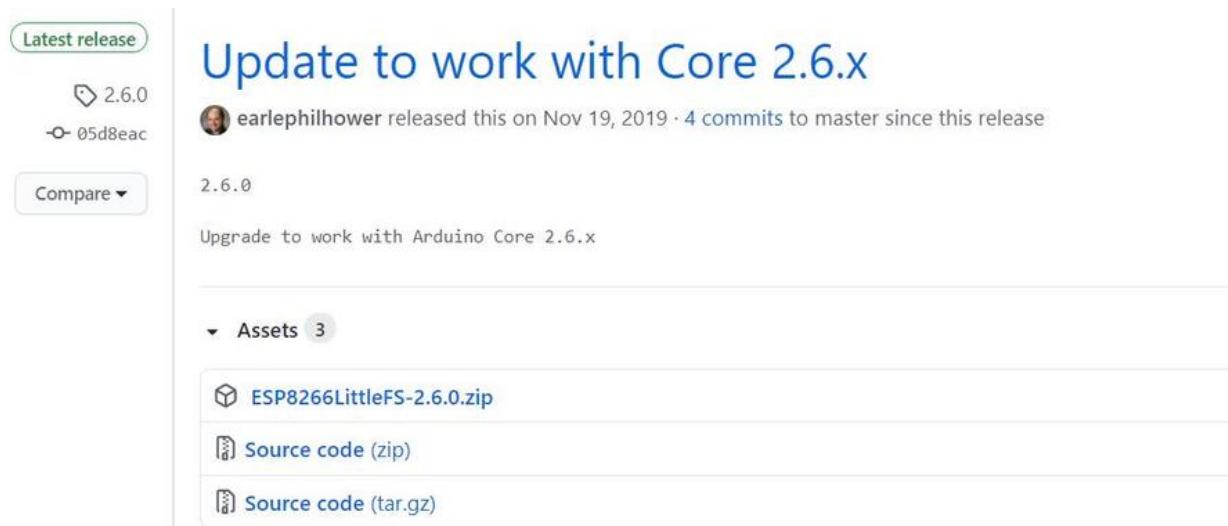
You can create, save and write files to the ESP8266 filesystem by writing the code yourself on the Arduino IDE. This is not very useful because you'd have to type your files' content in the Arduino sketch.

Fortunately, there is a plugin for the Arduino IDE that allows you to upload files directly to the ESP8266 filesystem from a folder on your computer. This makes it easy and straightforward to work with files.

SPIFFS is currently deprecated and may be removed in future releases of the ESP8266 core. It is recommended to use *LittleFS* instead. LittleFS is under active development, supports directories, and is faster for most operations. The methods used for SPIFFS are compatible with LittleFS, so we can simply use the expression `LittleFS` instead of `SPIFFS` in our code.

For file uploading, you need to install the LittleFS Filesystem Upload Plugin. Follow the next steps to install the filesystem uploader:

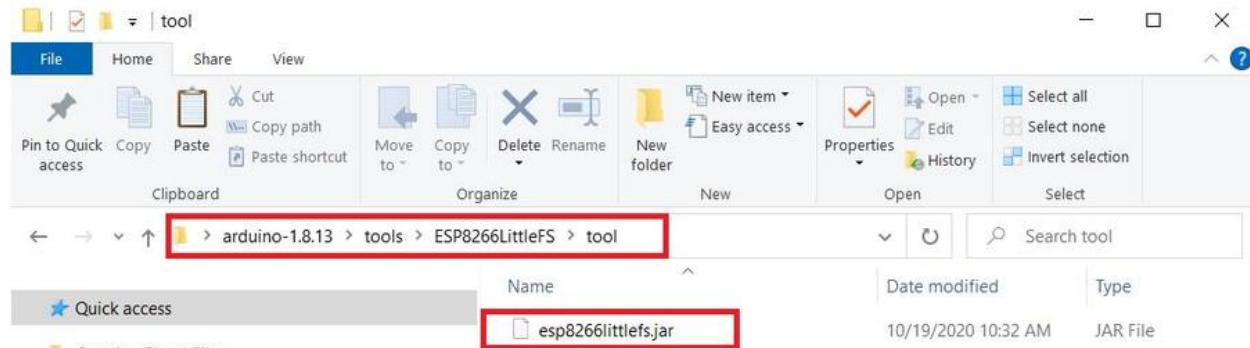
- 1) Go to the [releases page](#) and click the `ESP8266LittleFS-X.zip` file to download.



- 2) Go to the Arduino IDE directory, and open the `Tools` folder.

- 3) Unzip the downloaded .zip folder to the *Tools* folder. You should have a similar folder structure:

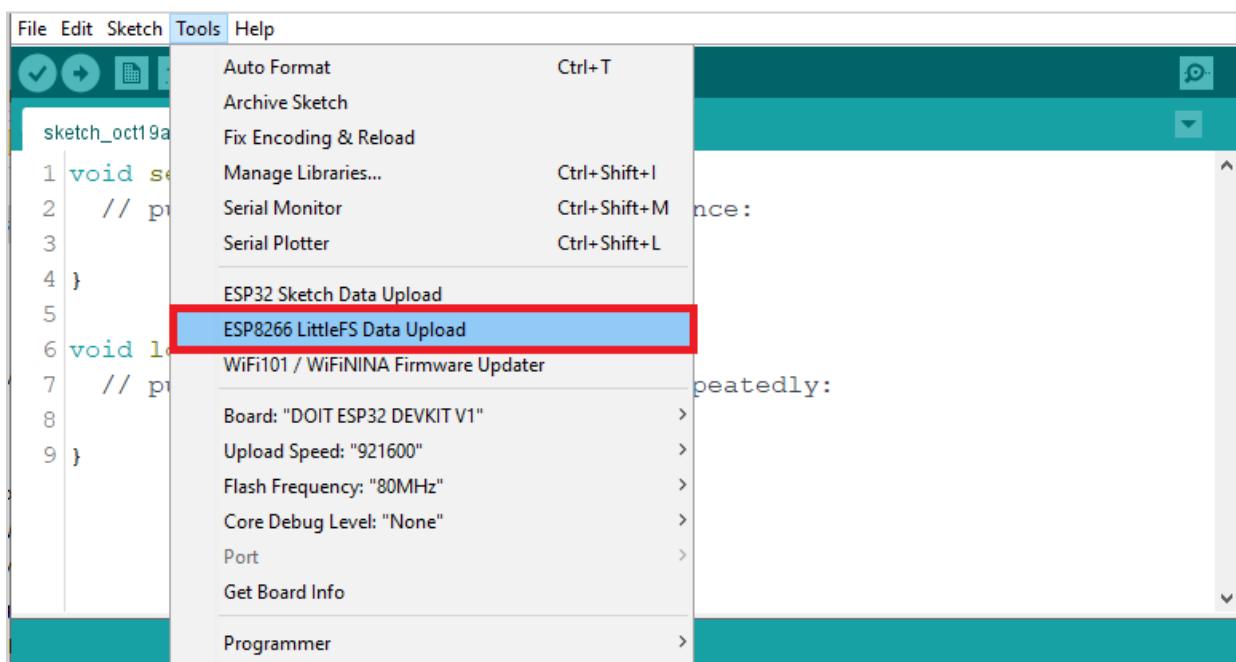
```
<home_dir>/Arduino-<version>/tools/ESP8266FS/tool/esp8266fs.jar
```



On the OS X create the *tools* directory in `~/Documents/Arduino/` and unpack the files there

- 4) Finally, restart your Arduino IDE.

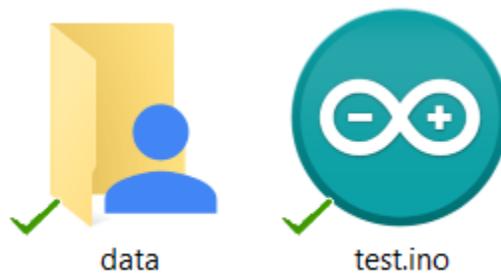
To check if the plugin was successfully installed, open your Arduino IDE and select your ESP8266 board. In the **Tools** menu, check that you have the option "**ESP8266 LittleFS Data Upload**".



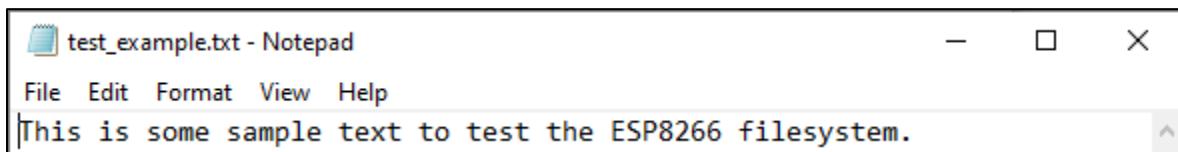
Uploading Files using the Filesystem Uploader (ESP8266)

To upload files to the ESP8266 filesystem, follow the next instructions.

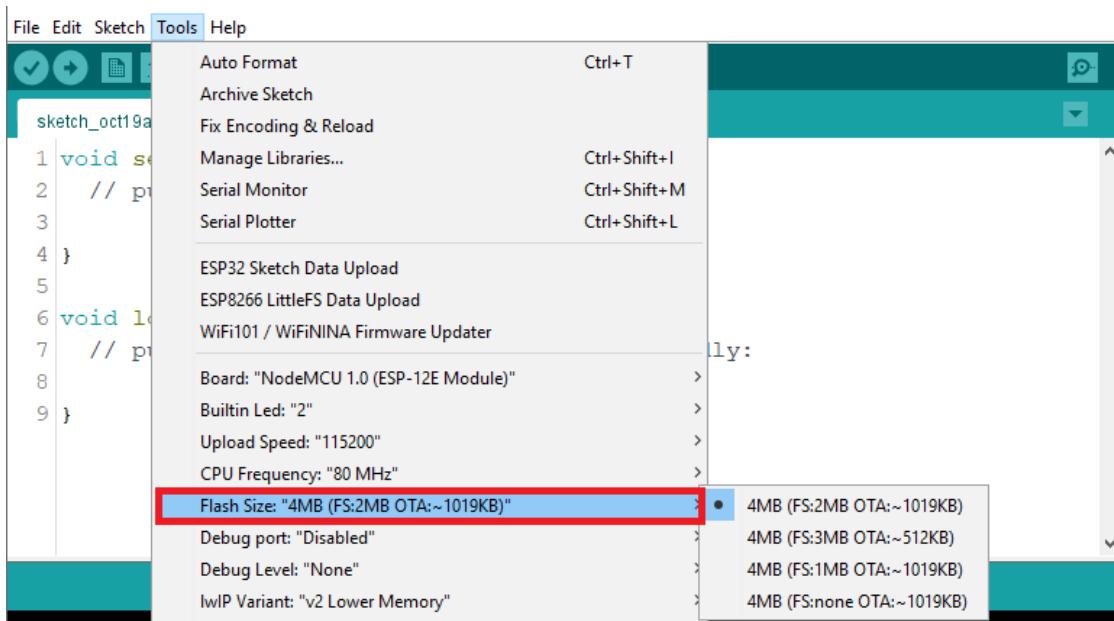
- 1) Create an Arduino sketch and save it. For demonstration purposes, you can save an empty sketch.
- 2) Then, open the sketch folder. You can go to **Sketch > Show Sketch Folder**.
The folder where your sketch is saved should open.
- 3) Inside that folder, create a new folder called *data*.



- 4) Inside the **data** folder is where you should put the files you want to save into the ESP8266 filesystem. As an example, create a **.txt** file with some text called *test_example*.

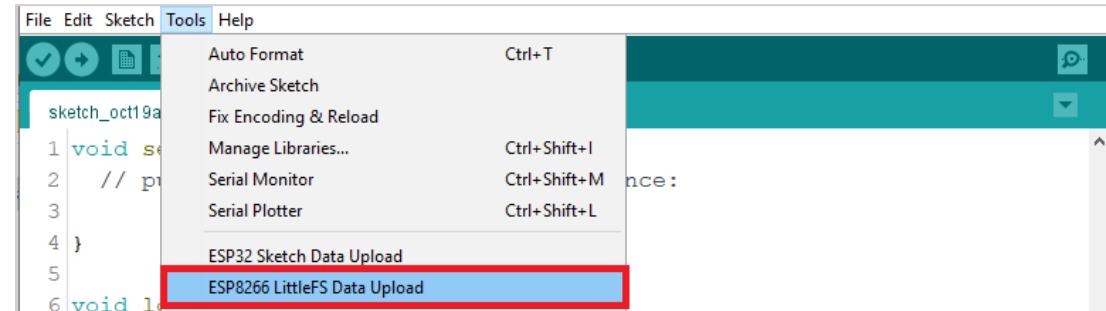


- 5) In the Arduino IDE, in the **Tools** menu, select the desired flash size (this will depend on the size of your files).



- 6) Then, to upload the files in the Arduino IDE, you just need to go to **Tools > ESP8266 LittleFS Data Upload.**

Important: ensure the Serial Monitor is closed. Otherwise, it will fail.



After a few seconds, you should get the message "**LittleFS Image Uploaded**". The files were successfully uploaded to the ESP8266 filesystem.

```
LittleFS Image Uploaded
Wrote 2072576 bytes (2264 compressed) at 0x00200000 in 0.2 seconds (effecti
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

1 NodeMCU 1.0 (ESP-12E Module) on COM3

Testing the Uploader (ESP8266)

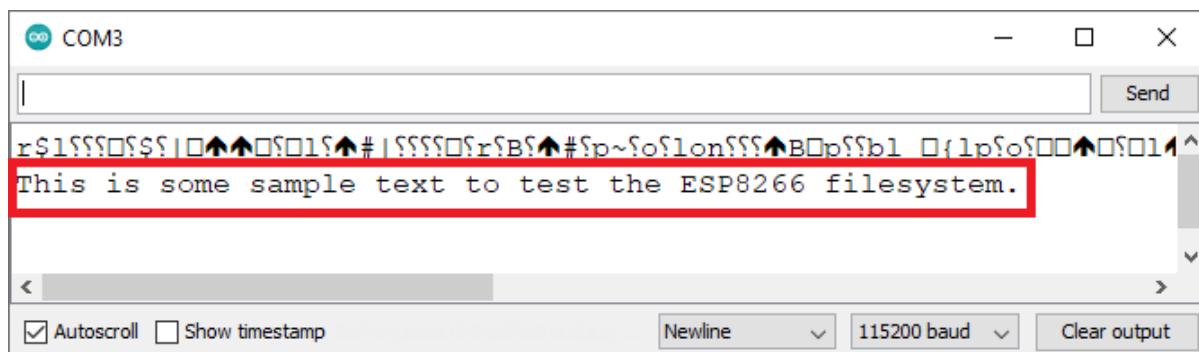
Now, let's check if the file was actually saved into the ESP8266 filesystem. Simply upload the following code to your ESP8266 board.

```
#include "LittleFS.h"
void setup() {
    Serial.begin(115200);
    if(!LittleFS.begin()){
        Serial.println("An Error has occurred while mounting LittleFS");
        return;
    }
    File file = LittleFS.open("/test_example.txt", "r");
    if(!file){
        Serial.println("Failed to open file for reading");
        return;
    }
    Serial.println("File Content:");
    while(file.available()){
        Serial.write(file.read());
    }
    file.close();
}
void loop() {
```

You can download the code by clicking on the following link:

- https://github.com/RuiSantosdotme/build-web-servers-dl/raw/main/Test_LITTLEFS_ESP8266.zip

After uploading, open the Serial Monitor at a baud rate of 115200. Press the ESP8266 on-board "RST" button. It should print the content of your .txt file on the Serial Monitor.



You've successfully uploaded files to the ESP8266 filesystem using the plugin.

Installing Libraries

If you're going to use Arduino IDE, we recommend installing all the required libraries from the start. This way, you won't have to worry about that later on.

Installing the ESPAsyncWebServer Library

To build the web servers, we'll use the [ESPAsyncWebServer library](#). To use this library with the ESP32, you also need to install the [AsyncTCP library](#). If you're using the ESP8266, you also need to install the [ESPAsyncTCP](#). We recommend installing the three libraries so that you're able to program the ESP32 and ESP8266. Follow the next steps to install the libraries.

Click the following links to download the libraries .zip folder

- [ESPAsyncWebServer](#)
- [AsyncTCP](#)
- [ESPAsyncTCP](#)

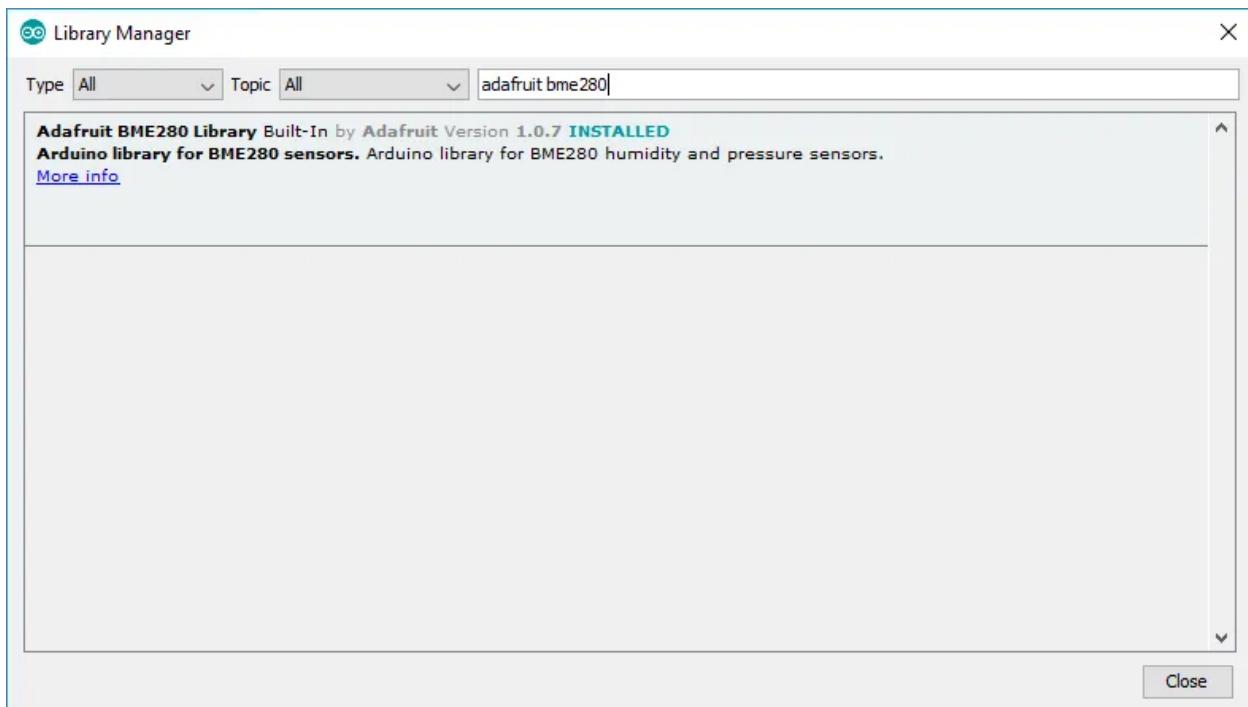
In your Arduino IDE, go to **Sketch > Include Library > Add .zip Library** and select the libraries you've just downloaded.

Installing BME280 Libraries

For the examples related to sensor readings, we'll use the BME280 sensor. You can use any other sensor. However, if you want to follow the same projects we'll build, you need to install the [Adafruit BME280](#) library and the [Adafruit Sensor](#) library.

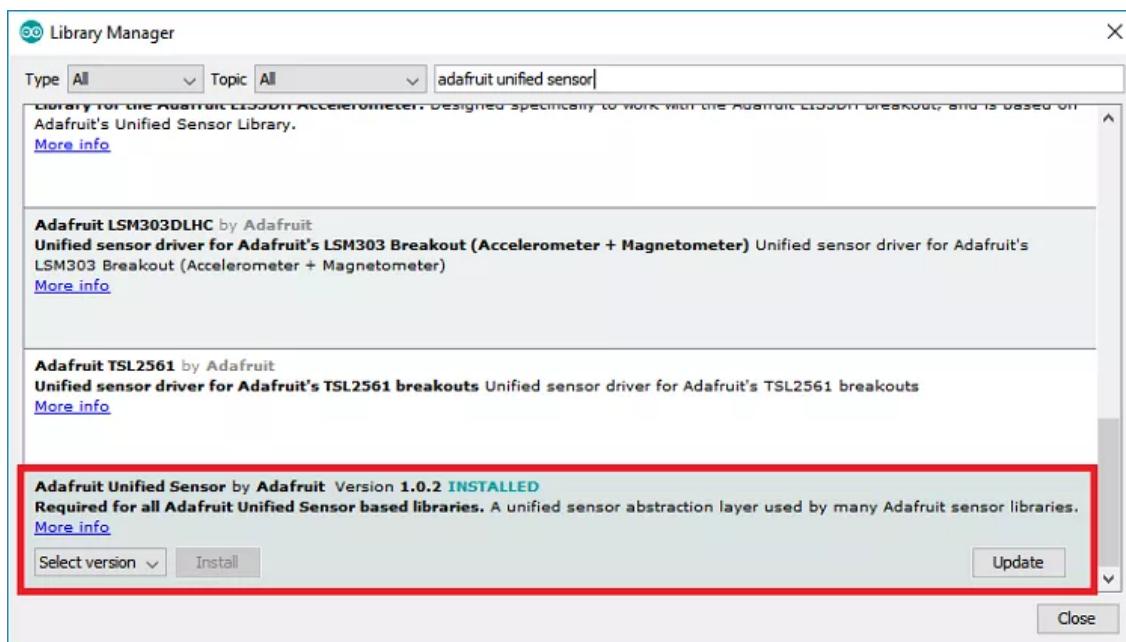
Follow the next steps to install the libraries:

- 1) Open your Arduino IDE and go to **Sketch > Include Library > Manage Libraries**. The Library Manager should open.
- 2) Search for "adafruit bme280" on the Search box and install the library.



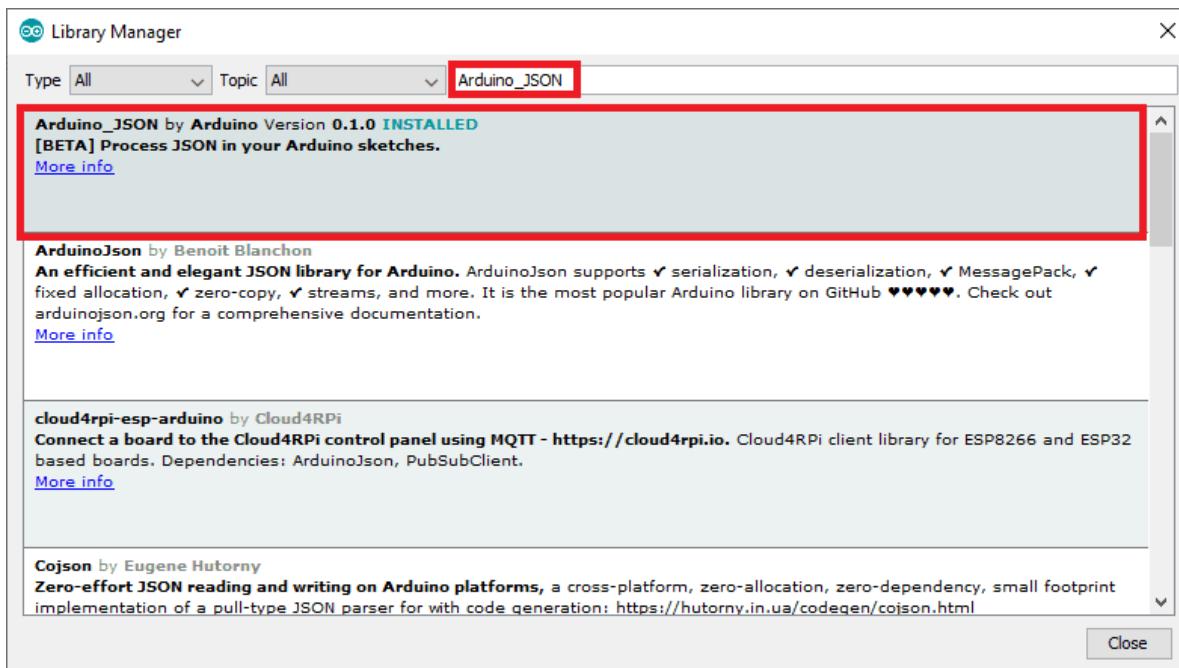
After installing this library, you may be prompted to install the Adafruit Sensor library. Install it. If it doesn't prompt anything about that, follow the next steps to install the library in your Arduino IDE:

- 3) Search for "*Adafruit Unified Sensor*" in the search box. Scroll down to find the library and install it.



Installing the Arduino_JSON Library

To make it easy to handle JSON strings in Arduino IDE, you need to install the Arduino_JSON library. You can install this library in the Arduino IDE Library Manager. Just go to **Sketch > Include Library > Manage Libraries** and search for the library name "Arduino_JSON" as follows:



Install the Arduino_JSON library by Arduino Version 0.1.0.

If you're going to use Arduino IDE to follow this eBook, you have installed everything you need.

MODULE 2

Getting Started with HTML, CSS, and JavaScript

Learn HTML, CSS, and JavaScript basics:

- **HTML** to define the content of web pages
- **CSS** to specify the layout and style of web pages
- **JavaScript** to program the behavior of web pages

Getting Started with HTML



In this section, you'll learn the HTML basics necessary to build simple but functional web pages for your ESP32/ESP8266 projects.

This is not meant to be a complete HTML course or a web development course. You'll learn the basics to build your web pages and become familiar with most HTML terms. By the end of the eBook, if you need additional features for your projects, you'll know how to search for them.

Download Example Files

Throughout this section, we'll present several examples. You can type each example as you go, or you can download the HTML files for all the examples at the following link:

- [HTML Examples](#)

Organizing your files

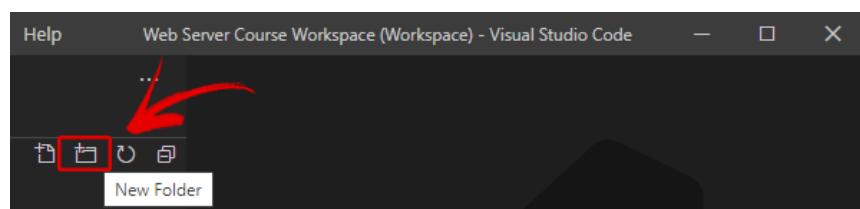
When you're working on your web pages, it's always a good idea to keep everything well organized, especially if you have several files for the same project. It's not a good idea to have your files lying around in your *Desktop* or your *Documents* folder. We recommend that you create a folder for your project on your computer—for example, one called *ESP Web Server Course*.

After creating that folder, open Visual Studio Code (installed in the previous Module). If you still have the Blink LED project opened from the previous sections, you can close it.

Go to **File > Add folder to workspace** and add the folder you've just created.

Now, that folder will be in your workspace. All folders inside that folder will show up in the File Explorer tab. This is useful because you can easily navigate through project files. Go to **File > Save Workspace as...** to save the workspace with whatever name you want. The workspace is simply a file containing the files and folders listed in the File Explorer. Later, if you want to load all files saved on that workspace, you just need to open the workspace file.

In the File Explorer, select the folder you've created and create a new folder for your first project by clicking on the **New Folder** icon. You can call that folder *HTML examples*.



Creating Your First Web Page

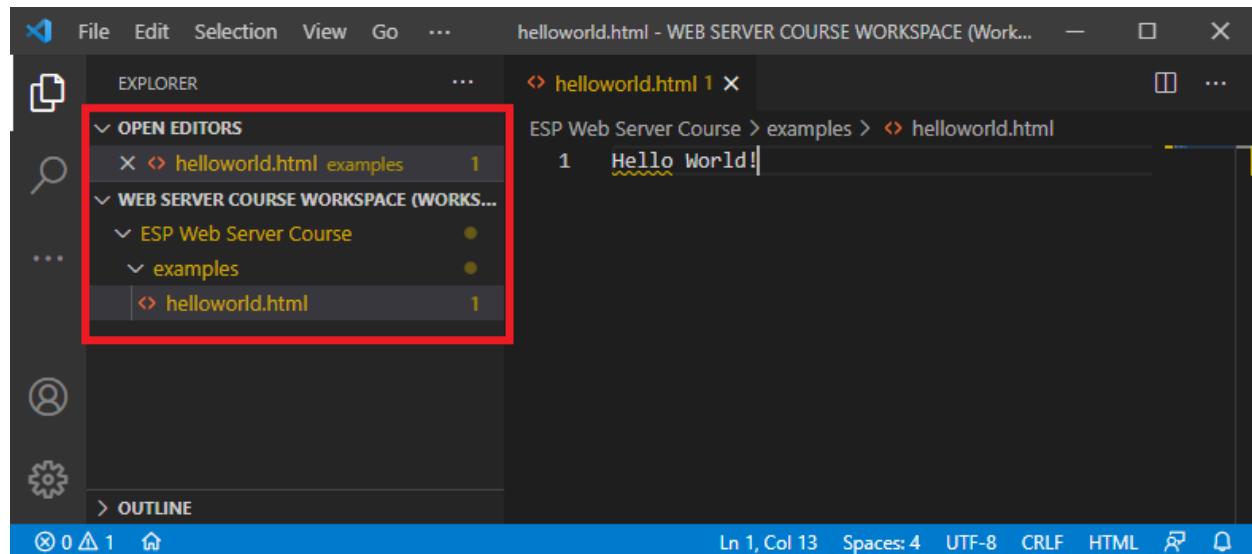
Select that new folder with your mouse and click the **New File** icon. It will create a new file under that folder. Our first example will be a simple web page that displays the "Hello World!" message. So, you can name it *helloworld.html*.

Your new file's name should end with the *.html* extension. Otherwise, your web browser won't recognize the file. Additionally, when using the correct file extension, VS Code knows that it is working with an HTML file and can find errors, help with autocompletion, and other valuable features.

Inside that file, write the following:

```
Hello World!
```

At this moment, your Explorer tab should look like this:

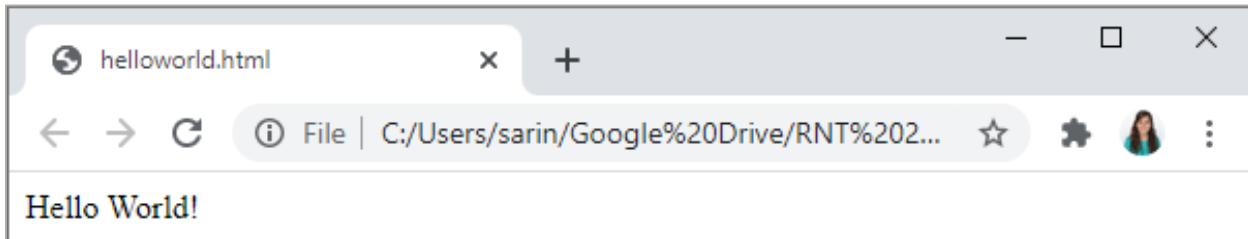


If your files are highlighted with a yellow color, don't worry. It is just VS Code complaining that the HTML file doesn't have the right structure. This is not relevant for now.

Go to **File > Save** or simply press **Ctrl+S** to save the file.

To see the result of this HTML document, you need a browser to "read" it. Throughout this eBook, we'll use the Google Chrome web browser, and that's the one we recommend.

Open Google Chrome. Go to your project folder, drag the *helloworld.html* file into the web browser's address window (also called the URL bar), and then press Enter. And that's it. You've created your first web page.



This isn't very exciting—it's just some text in your browser. Let's learn some HTML so that we can build something more interesting.

Introducing HTML

HTML stands for Hypertext Markup Language and is the predominant markup language used to create web pages. Web browsers were created to read HTML files. The HTML tags tell the web browser how to display the content on the page. We'll see how tags work in the following example.

Writing the First Line

The first line of any HTML document is always the following:

```
<!DOCTYPE html>
```

`!DOCTYPE` isn't an HTML-specific tag. It's simply an instruction that tells your web browser that it's reading an HTML document and which version of HTML you're using (HTML 5).

Structuring an HTML Document

The overall structure of an HTML document looks like this:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

In VS Code, select the *HTML Examples* folder created previously and create a new file with the *.html* extension (for example, *basicstructure.html*).

The HTML that defines the structure of your page goes between the `<html>` and `</html>` tags. Because most web pages require the preceding snippet, you can use that template every time you start writing a new web page.

An HTML document is divided into two main parts: *head* and *body*.

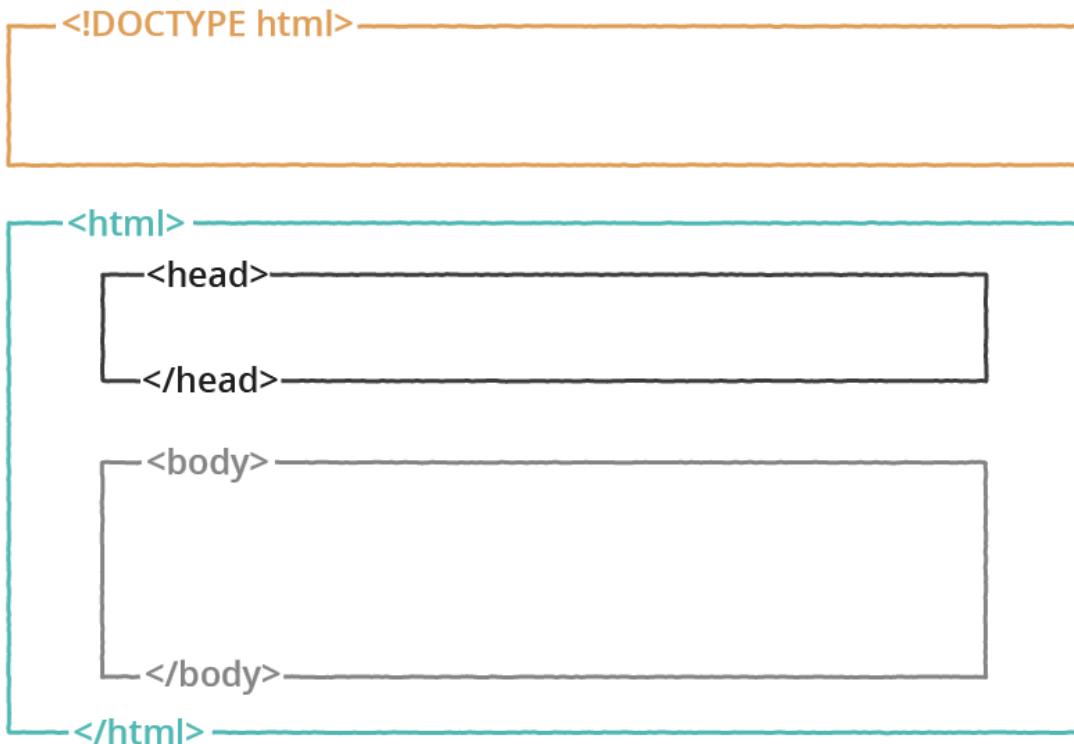
Head

The `<head>` and `</head>` tags mark the start and end of the *head*. The head is where you insert data about the HTML document that is not directly visible to the end-user but adds functionalities to the web page like the title, scripts, styles, and more—this is called *metadata*. For example, you can add the following `<meta>` tag to make your web page responsive in any web browser.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Body

The `<body>` and `</body>` tags mark the start and end of the body. Everything that goes inside those tags is visible page content. The body includes the content of the page like headings, paragraphs, buttons, tables, etc. The following image summarizes the HTML page structure.



Adding a Title

The title goes between the `<title></title>` tags that go within the `<head>` and `</head>` tags. The title is exactly what it sounds like: the title of your document, which shows up in your web browser's title bar.

The title element also provides a title for the page when you add it to your favorites in your browser. It also displays a title for the page in search-engine results (this doesn't apply to our projects because your web server won't be available for search-engine results).

Here's how you can add the `<title>` tag to your HTML document.

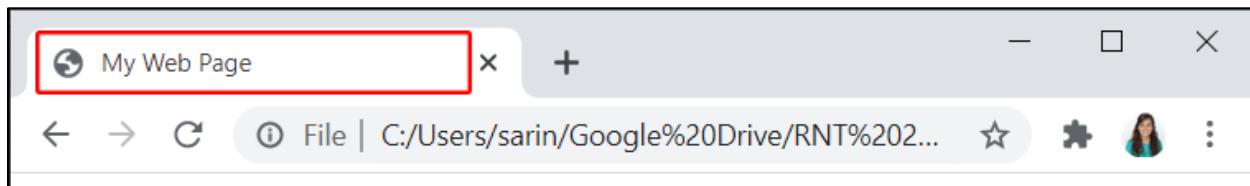
```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
  </body>
</html>
```

Indentation! Indentation! Indentation! Web design uses a lot of tags that go inside tags that go inside tags. Even though using indentation is not strictly necessary, it's extremely easy to get lost if you don't use this technique.

VS Code indents the tags automatically when you're writing an HTML document.

Save your *basicstructure.html* file. Go to your project folder (outside VS Code) and drag the *basicstructure.html* file into the web browser window.

You should see a title in the web browser tab. At the moment, it doesn't display anything else because we didn't add any content to the body of the web page.



Save your file every time you make changes, and then refresh the web page. All changes should be updated immediately.

Tip: there is an extension in VS Code that allows you to preview live updates of your web page as you make changes on the files without the need to refresh the web browser. The Extension is called "**Live Server**". Install it in your VS Code if you want to give it a try.

Favicon

A favicon is a small graphic image (icon) associated with a particular web page. Web browsers display favicons as a visual reminder of the website's identity in the web browser tabs.

When you load an HTML file on Google Chrome or other web browsers, it requests the favicon. The optimal size for creating a favicon is 16x16 pixels. We provide an example of a favicon image in the project folder.

- [You can download the favicon here](#)

Place the favicon in the same folder that your *basicstructure.html* file.

 basicstructure.html	9/18/2020 6:58 PM	Chrome HTML Do...	0 KB
 favicon.png	9/18/2020 11:46 AM	PNG File	1 KB
 helloworld.html	9/18/2020 6:53 PM	Chrome HTML Do...	1 KB

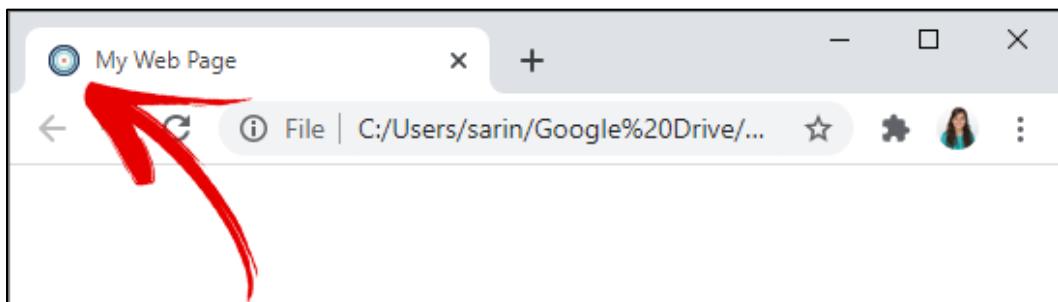
Then, you should reference the favicon using the following meta tag that goes in the head of your file. The image is called *favicon.png*.

```
<link rel="icon" type="image/png" href="favicon.png">
```

Here's how your file looks like:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" type="image/png" href="favicon.png">
  </head>
  <body>
  </body>
</html>
```

Save the file and refresh your web browser. The favicon icon should be displayed next to the title.



If you don't want to display any favicon, we recommend that you add this line in the head of your document. This doesn't change the content of your web page or in how it looks.

```
<link rel="icon" href="data:,">
```

Headings

Now you can start adding visible content to the web page, starting with a heading. Different levels of HTML headings are defined with the `<h1>` through `<h6>` tags (`h1` stands for heading number one), and the heading tags always go inside the `<body>` tags. Type the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
  </head>
  <body>
    <h1>This is heading 1</h1>
    <h2>This is heading 2</h2>
    <h3>This is heading 3</h3>
    <h4>This is heading 4</h4>
    <h5>This is heading 5</h5>
    <h6>This is heading 6</h6>
  </body>
</html>
```

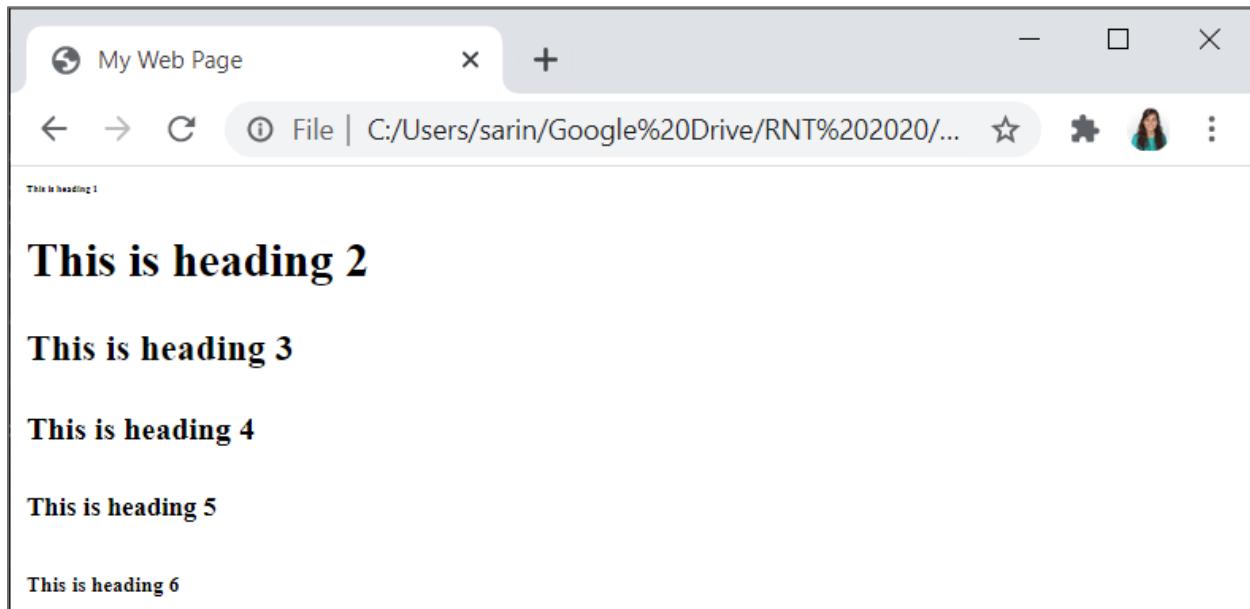
Save the file and refresh your web browser. You'll get different heading levels, as shown below.



Each heading has a default font size, with heading 1 with the biggest font size and heading 6 with the smallest font. However, all of this is customizable with styles. For example, modify heading 1 as follows:

```
<h1 style="font-size: 5px;">This is heading 1</h1>
```

Now, heading 1 font is smaller than the other headings.



Paragraphs

To mark text as a paragraph, use the `<p></p>` tags. Most of your readable content goes inside a paragraph tag. Add a paragraph tag to your HTML file like so:

```
<p>This is my first paragraph.</p>
```

The following example adds several paragraphs to your file.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
  </head>
  <body>
    <h1>This is heading 1</h1>
```

```
<p>This is my first paragraph.</p>
<p>This is another paragraph.</p>
<p>Another paragraph.</p>
</body>
</html>
```

Save the file and open it in your browser.

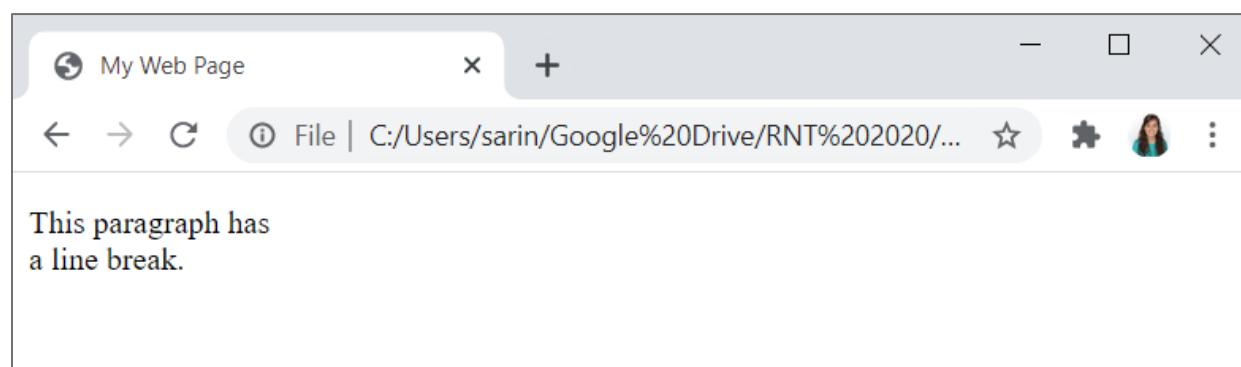
Here's the result: a page with one heading and three paragraphs.



Empty HTML Elements

HTML elements with no content are called empty elements. A commonly used empty element is `
` and it also doesn't have a closing tag. The `
` tag defines a line break. Here's an example:

```
<p>This paragraph has <br>a line break.</p>
```



Formatting Your HTML Document

Many tags are useful for formatting your text or web pages. We'll just take a look at a few of the most used.

Bold

`` and ``: put text inside the `` and `` or `` and `` tags to define **bold** text. Using those tags, the text in your web browser appears in bold, but keep in mind that HTML is used to mark things, not to make them pretty.

With the `` tags, the text is defined without any extra importance. The HTML `` element defines strong text and marks it as important (this is useful for search engine results, which doesn't apply to our case). For example:

```
<b>This text is bold</b>
```

The HTML `` element defines strong text, with added semantic "strong" importance.

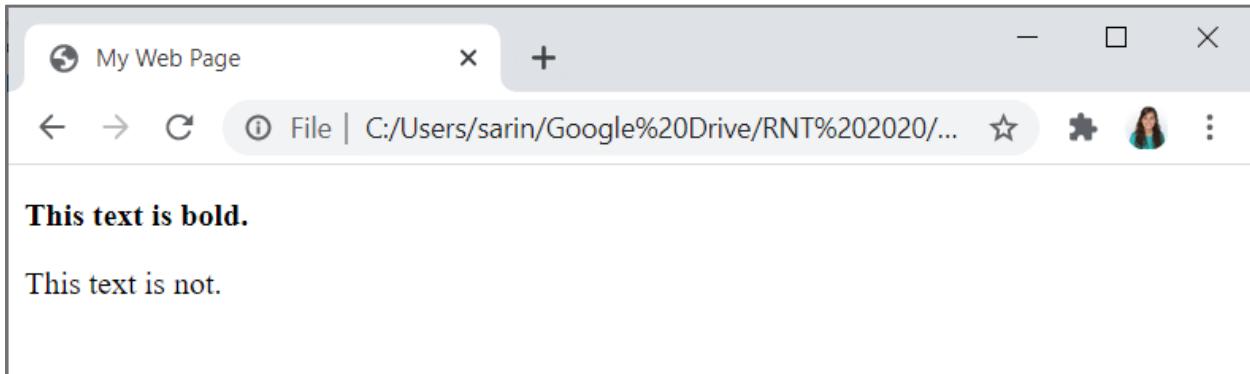
For example:

```
<strong>This text is strong</strong>
```

You can copy the following HTML text to your document to see the results.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <p><b>This text is bold.</b></p>
    <p>This text is not.</p>
  </body>
</html>
```

Save your file and open it in your browser.



Note: if you simply want to make your text bold, you should use CSS.

Italic

`<i>` and ``: put text inside the `<i>` and `</i>` or `` and `` tags to display text in *italic*. The HTML `<i>` element defines italic text, without any extra importance. On the other hand, the HTML `` element defines emphasized text with semantic importance.

Add the following line to your previous example to add a paragraph in italic script.

```
<p><em>This text is italic</em></p>
```

So, your HTML document will be as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <p><b>This text is bold.</b></p>
    <p>This text is not.</p>
    <p><em>This text is italic</em></p>
  </body>
</html>
```

And here's the result: the last paragraph is in italic script.

This screenshot shows a web browser window titled "My Web Page". The address bar indicates the file is located at C:/Users/sarin/Google%20Drive/RNT%202020/... . The page content includes:

- This text is bold.**
- This text is not.
- This text is italic*

Superscript and Subscript

Put text inside the `^{` and `}` tags to define superscript text. Superscript text appears half a character above the normal line, often rendered in a smaller font.

Similarly, the text between the `_{` and `}` tags is subscript text. Subscript text appears half a character below the normal line and it's often rendered in a smaller font. For example:

```
<p>This is <sub>subscripted</sub> text.</p>
<p>This is <sup>superscripted</sup> text.</p>
```

And here's the result:

This screenshot shows a web browser window titled "My Web Page". The address bar indicates the file is located at C:/Users/sarin/Google%20Drive/RNT%202020/... . The page content includes:

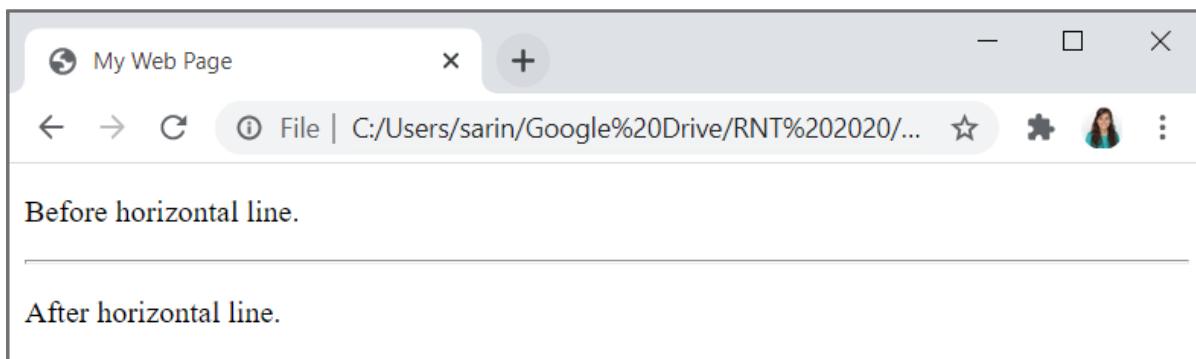
- This is _{subscripted} text.
- This is ^{superscripted} text.

Horizontal Line

`<hr>`: use the `<hr>` tag to add a horizontal line to separate portions of content. Here's an example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
  </head>
  <body>
    <p>Before horizontal line.</p>
    <hr>
    <p>After horizontal line.</p>
  </body>
</html>
```

Your web page should display a horizontal line, as shown below.



You can use lots of other tags to mark your text, but we don't cover them because they're not relevant to the remainder of this eBook.

Comments

`<!--insert comment-->`: This is used to add comments to your HTML text. Comments don't appear on your web page, so you can use them to remember why you wrote some of the HTML tags when you revisit that document sometime later.

```
<!--This is a comment. It will be ignored by the browser.-->
```

Creating Lists

Lists help you organize your content. There are two types of lists: ordered lists and unordered lists.

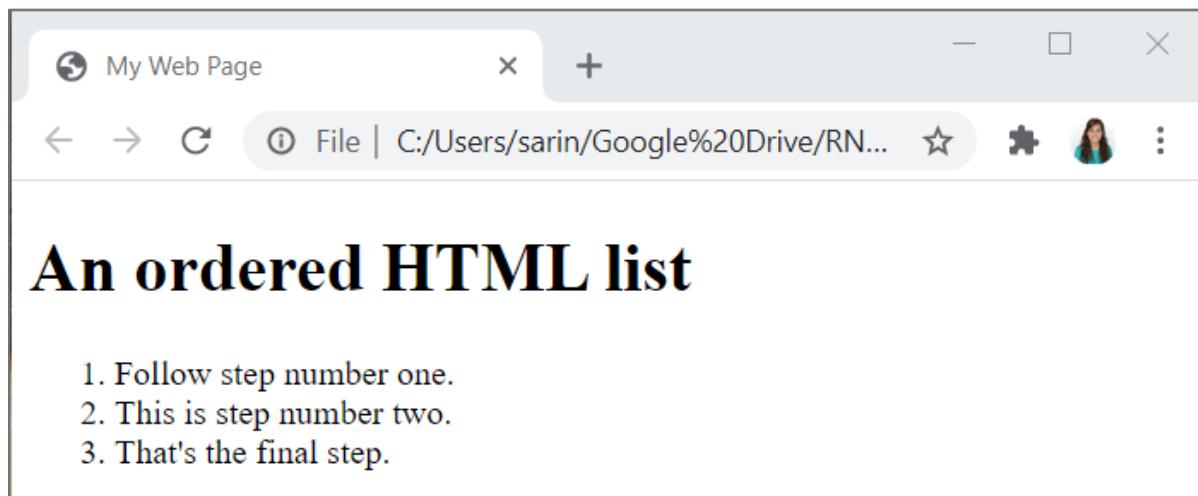
Ordered Lists

You designate an ordered list with the `` and `` tags when the order of your items is important, as in step-by-step instructions, for example. Each item in the list starts with a number or a letter. Each item should go inside the `` tags.

Here's an example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <h1>An ordered HTML list</h1>
    <ol>
      <li>Follow step number one.</li>
      <li>This is step number two.</li>
      <li>That's the final step.</li>
    </ol>
  </body>
</html>
```

Here's the result:

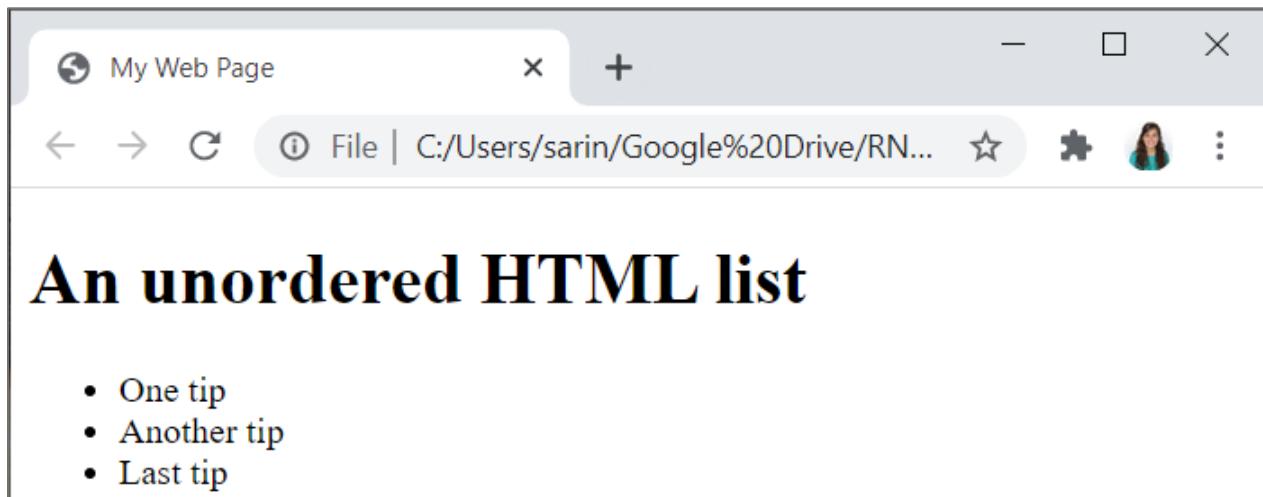


Unordered Lists

You use the `` and `` tags when the items in the list have equal importance and don't have a sequence associated with them. Each item on the list goes between the `` tags as shown below.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <h1>An unordered HTML list</h1>
    <ul>
      <li>One tip</li>
      <li>Another tip</li>
      <li>Last tip</li>
    </ul>
  </body>
</html>
```

Here's the result:



Adding Images

To insert an image, use the `` tag. You need to have an image to work with in your project folder, so copy an image there. The image in the example is called *ESP32.png*.

- [Click here to download the image](#)

Add the following tag to the body of your code, but replace *ESP32.png* with the path to your image file if you're using a different image filename or if it is stored in a different location:

```

```

Like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
  </head>
  <body>
    <h1>ESP32 Board</h1>
    
  </body>
</html>
```

Your web page shows the ESP32 picture or whatever image you've defined.



Did you notice that the `` doesn't have a closing tag? Some HTML tags don't have a closing tag because everything they need to do their task is actually placed inside the opening tag — not between an opening and closing tag.

If the image is a bit bigger than you need, you can change its size by inserting a couple of attributes to manipulate your image's width and height.

Here are some of the attributes that you can use with the `` tag:

- `src` (source): Specifies where your image is located. You can insert an URL or the path to your image location.
- `width`: Changes the width of your image, measured in pixels.
- `height`: Changes the height of your image, measured in pixels.
- `alt`: Stands for alternative text. If a browser can't display your image for some reason, it displays your alternative text instead of the image.

For example, if you want to change the width of your image to 290 pixels and the height to 350 pixels, type the following:

```

```

Buttons

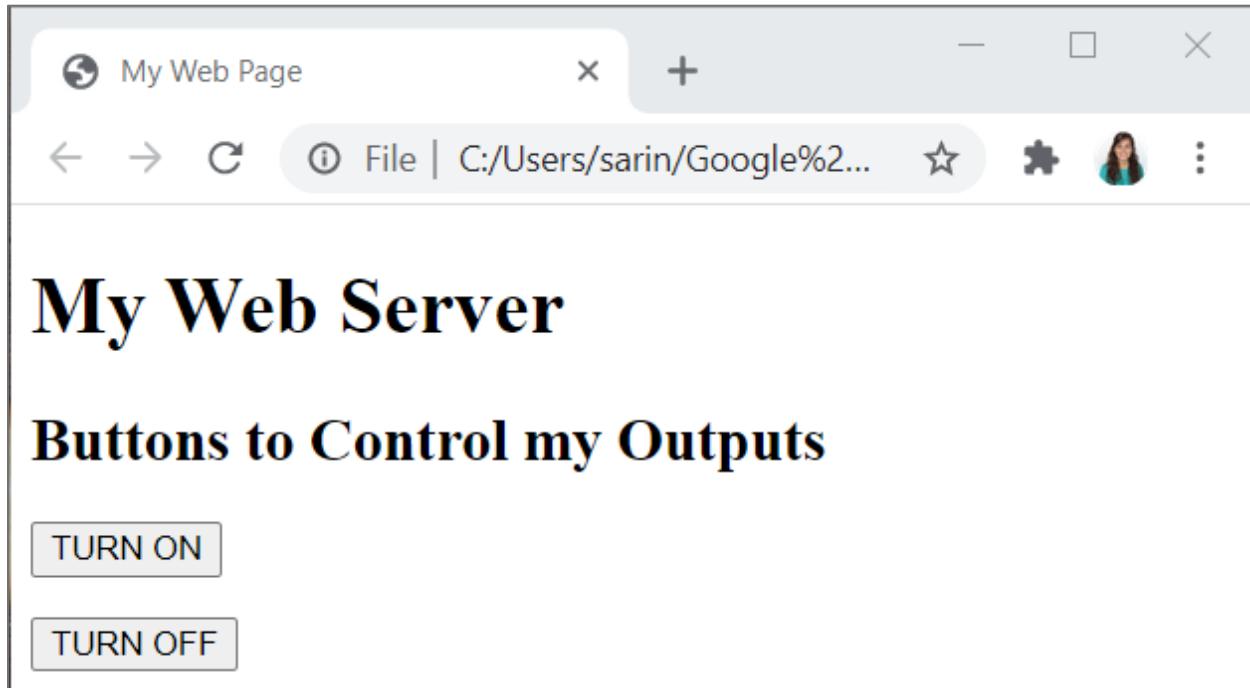
To insert a button in your page you use the HTML `<button>` and `</button>` tags. Between the tags, you should write the text you want to appear inside the button. For example, if you're going to control an ESP output, you can add a "**TURN ON**" text. Take a look at the following example that displays two buttons.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
  </head>
  <body>
    <h1>My Web Server</h1>
```

```
<h2>Buttons to Control my Outputs</h2>
<p><button>TURN ON</button></p>
<p><button>TURN OFF</button></p>
</body>
</html>
```

Note that we've inserted the `<button>` and `</button>` tags between paragraph tags `<p>` and `</p>`.

Now, your web page should have two buttons, as shown in the figure below.



Press the buttons. Nothing happens because those buttons don't have any hyperlink or action (JavaScript) associated with them. You'll learn more about this later in this eBook.

Note: a button is also a type of input. So, alternatively, you can create a button using the `<input>` tag with the proper attribute, as we'll see later on.

Hyperlinks

HTML links are called hyperlinks. You can add hyperlinks to text, images, buttons, or any other HTML element.

Text

To add a hyperlink, use the `<a>` and `` tags in the following format:

```
<a href="url">Link text</a>
```

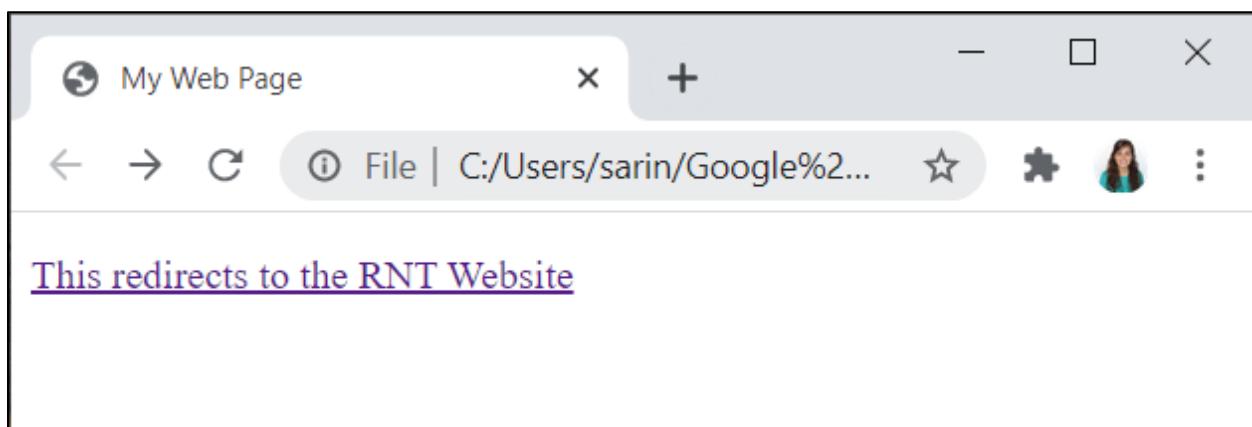
Between the `<a>` and `` tags, you should place the HTML element you want to apply the link to. For example:

```
<p><a href="https://randomnerdtutorials.com/">This redirects to the RNT Website</a></p>
```

This previous tag creates a clickable text that redirects you to the Random Nerd Tutorials website. Add the previous tag in your HTML document, as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <p><a href="https://randomnerdtutorials.com/">This redirects to the RNT Website</a></p>
  </body>
</html>
```

Click on the text, and you'll be redirected to the Random Nerd Tutorials website.



The `href` attribute specifies where the link should go. When you click on the text, you are redirected to the RNT website.

Note: even though most web browsers don't display `http://` at the start of the domain name, you need to type that prefix in your `href=""` attributes; otherwise, the website may not open.

Buttons

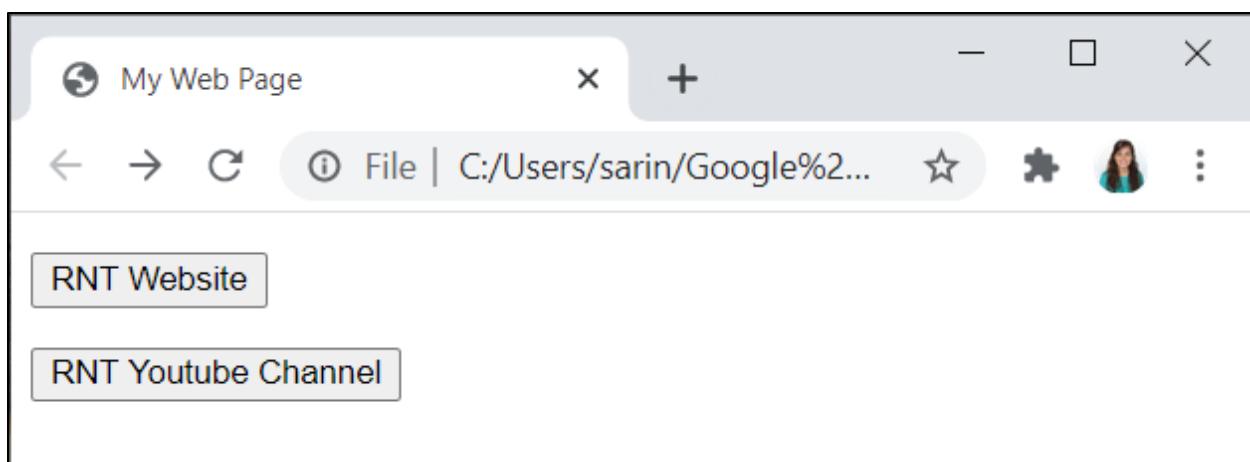
You can also add hyperlinks to buttons. Here's an example:

```
<p><a href="https://randomnerdtutorials.com/"><button>RNT Website</button></a></p>
<p><a href="https://www.youtube.com/user/RandomNerdTutorials"><button>RNT Youtube
Channel</button></a></p>
```

Place these tags in your HTML document like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
  </head>
  <body>
    <p><a href="https://randomnerdtutorials.com/"><button>RNT Website</button></a>
</p>
    <p><a href="https://www.youtube.com/user/RandomNerdTutorials"><button>RNT You
tube Channel</button></a></p>
  </body>
</html>
```

Your web page should display two buttons, as shown below.



Images

Here's another example using a hyperlink in an image:

```
<a href="https://randomnerdtutorials.com/projects-esp32/">
  
</a>
```

Here's the complete example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:, ">
  </head>
  <body>
    <h1>ESP32 Projects</h1>
    <p>Click the image to access ESP32 projects.</p>
    <a href="https://randomnerdtutorials.com/projects-esp32/">
      </a>
    </body>
</html>
```

Here's the result:



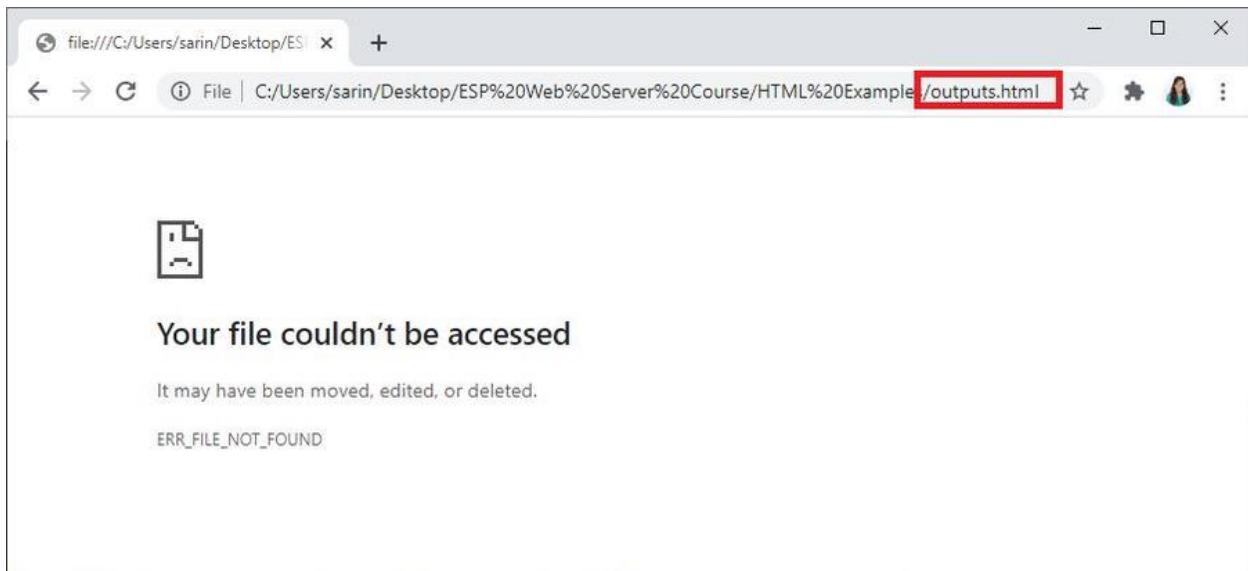
If you click on the ESP32 image, you'll be redirected to the RNT ESP32 projects page.

Multiple Web Pages

In the ESP32 or ESP8266 web servers, you may want to be redirected to another HTML page that shows different settings or controls - like a website with multiple pages. In that case, you'll need to have another HTML file to be redirected to. For example, you may want to be redirected to a web page that shows all your outputs. All the HTML to build that page should be stored in another HTML document. It can be called *outputs.html*, for example. Your hyperlink would look as follows:

```
<p><a href="outputs.html">This redirects to the outputs page.</a></p>
```

If you click the hyperlink, you'll be redirected to a web page *outputs.html* with "Your file couldn't be accessed". That happens because your browser can't find any file with that name because we haven't created it yet.



When using the ESP32 or ESP8266, the boards will receive a request for that specific file and will send different HTML text accordingly. Alternatively, you can also instruct your boards to perform specific actions depending on the request received. For example, you may want to add hyperlinks to your buttons. One redirects to /ON and

another to /OFF. When the ESP32 receives a request on the /ON URL it knows it's time to turn something on, or when it receives a request on the /OFF URL, it's time to turn something off. You'll better understand these concepts when we start building the ESP32/ESP8266 web server projects.

HTML Symbols

Characters that are not present on your keyboard can be replaced by HTML symbol entities. To add those symbols to an HTML page, you can use an HTML entity name.

There are also some special characters that can escape your code or break it like ", %, \, /, so if you need a web page with those characters, you must use HTML symbol entities. Alternatively, you can also use an entity number, a decimal, or hexadecimal reference.

There's a web page that you can visit to find the HTML symbols and corresponding entities: <https://dev.w3.org/html5/html-author/charref>

You can also use the following table with the most common symbols we'll use in the ESP web server projects.

Character	HTML Entity	HEX Code	HTML Code	Description
%	%	%	%	Percentage
&	&	&	&	Ampersand
°	°	°	°	Ordinal indicator
"	"	"	"	Quotation mark

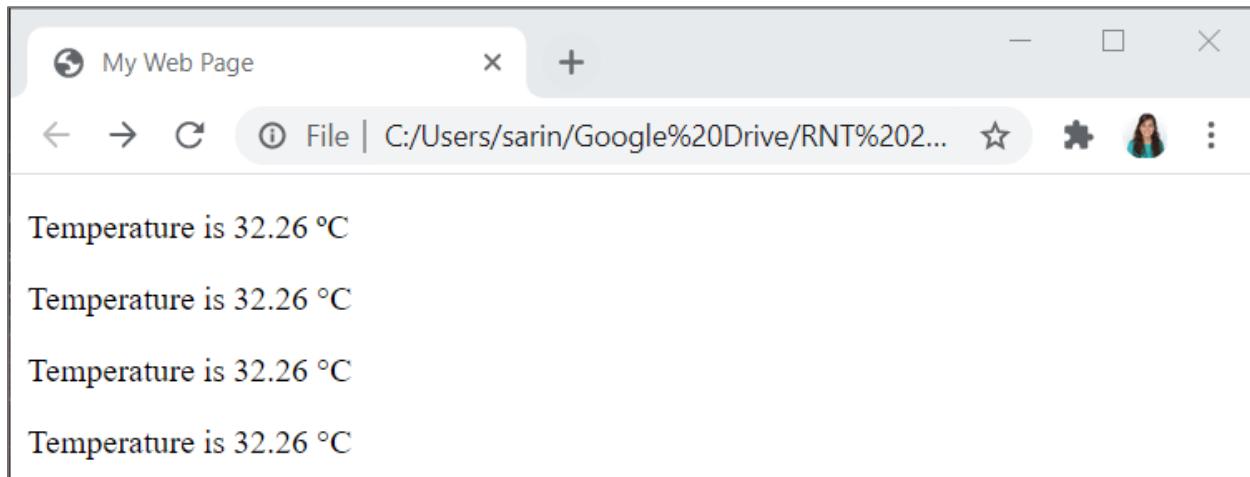
Here's an example using the HTML entity, HEX code and HTML code.

```
<p>Temperature is 32.26 °C</p>
<p>Temperature is 32.26 &deg;C</p>
<p>Temperature is 32.26 &#000xb0;C</p>
<p>Temperature is 32.26 &#176;C</p>
```

Add these paragraphs to your HTML document like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:, ">
  </head>
  <body>
    <p>Temperature is 32.26 °C</p>
    <p>Temperature is 32.26 &deg;C</p>
    <p>Temperature is 32.26 &#000xb0;C</p>
    <p>Temperature is 32.26 &#176;C</p>
  </body>
</html>
```

All paragraphs produce the same result.



HTML Tables

To create a table in HTML, start with the `<table>` and `</table>` tags. This encloses the entire table.

To create a row, use the `<tr>` and `</tr>` tags. The table is defined with a series of rows. Use the `<tr></tr>` pair to enclose each row of data

The table heading is defined using the `<th>` and `</th>` tags, and each table cell is defined using the `<td>` and `</td>` tags.

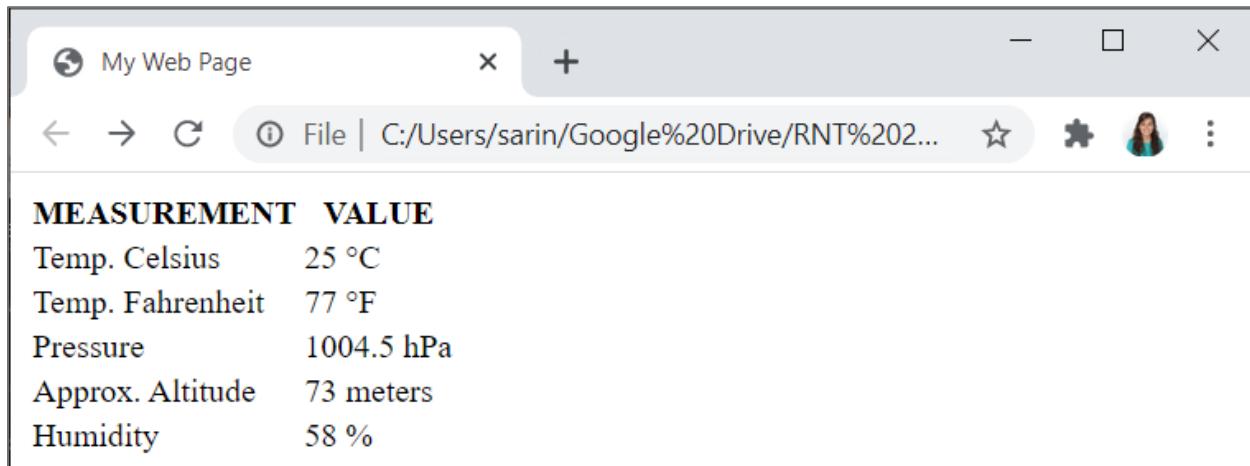
For example, imagine you want to create a table to display temperature, humidity, and pressure readings. Add the following to the body of your HTML document.

```
<table>
  <tr>
    <th>MEASUREMENT</th>
    <th>VALUE</th>
  </tr>
  <tr>
    <td>Temp. Celsius</td>
    <td>25 &deg;C</td>
  </tr>
  <tr>
    <td>Temp. Fahrenheit</td>
    <td>77 &deg;F</td>
  </tr>
  <tr>
    <td>Pressure</td>
    <td>1004.5 hPa</td>
  </tr>
  <tr>
    <td>Approx. Altitude</td>
    <td>73 meters</td>
  </tr>
  <tr>
    <td>Humidity</td>
    <td>58 &percnt;</td>
  </tr>
</table>
```

We create six rows: one for the headings and five to display each of the readings using the `<tr>` and `</tr>` tags. The header of the table contains a cell called MEASUREMENT, and another named VALUE.

Inside each row, create two cells using the `<td>` and `</td>` tags, one with the measurement's name and another to hold the measurement value.

In this example, we've entered the value of the readings manually. The idea is that those readings are sent automatically to your web page by the ESP, as we'll describe later.



The screenshot shows a web browser window titled "My Web Page". The address bar displays the file path "C:/Users/sarin/Google%20Drive/RNT%202...". The main content area contains a table with the following data:

MEASUREMENT	VALUE
Temp. Celsius	25 °C
Temp. Fahrenheit	77 °F
Pressure	1004.5 hPa
Approx. Altitude	73 meters
Humidity	58 %

The previous figure shows how the table looks. It doesn't have any applied styles. You change the way it looks using CSS, as we'll see later on.

HTML Forms

In many of your projects, you may want to get data from the user. For example, checkboxes where the user can choose if a sensor is enabled or not; toggle buttons to control outputs; input fields to define the threshold of a sensor or the value of a variable; sliders to set the brightness of an LED, and so on.

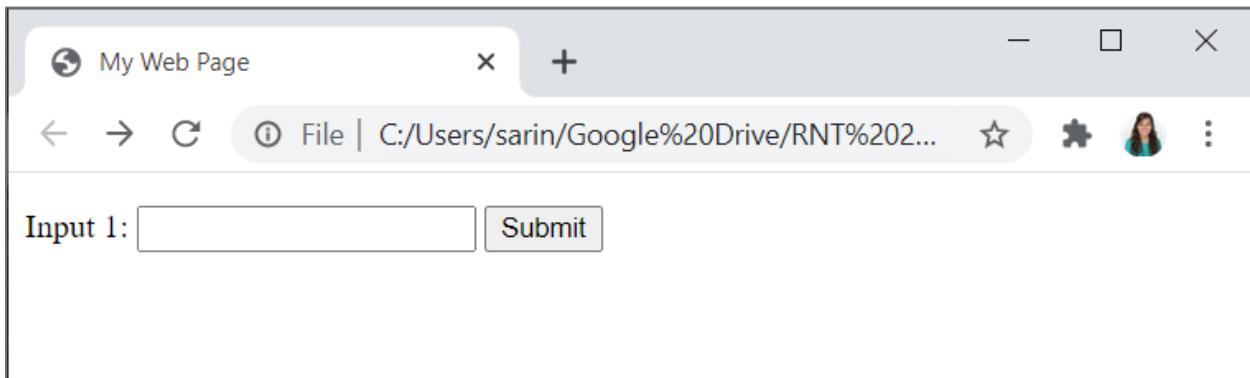
In HTML, the `<form>` tag is used to create an HTML form to collect user input. The user input can then be sent to the server (ESP32 or ESP8266) for processing. Based on the values collected on the form, your ESP board may perform different actions.

Alternatively, the user input can be used on the client side (browser) to update the interface in some way using JavaScript, for example.

Here's an example of the structure of a form:

```
<form action="/get" method = "GET">
  <p>
    <label for="input1">Input 1:</label>
    <input type="text" id="input1" name="input1">
    <input type="submit" value="Submit">
  </p>
</form>
```

Add this to the body of your HTML document. Then, open the page on your web browser. This is what you should get:



The HTML form contains different form elements. All the form elements are enclosed inside this `<form>` tag. It includes controls (checkboxes, buttons, inputs fields, menus, etc.) and labels for those controls.

Additionally, the `<form>` tag must include the `action` attribute that specifies what you want to do when the form is submitted. In our case, we want to send that data to the server (ESP32/ESP8266) when the user clicks the **Submit** button. The `method` attribute specifies the HTTP method (GET or POST) used when submitting the form data.

Note: don't worry if this seems a bit complicated by now. You'll learn more about this in the ESP32/ESP266 web server's section.

The `<label>` tag marks text as a label for a particular element. This tag is not mandatory in your form.

The `<input type="submit" value="Submit">` creates a submit button with the text "**Submit**". When you click this button, the form's data is sent to the server (the ESP32 or ESP8266 boards).

In the previous example, insert whatever you want in the field. For example: "2". When you click on the **Submit** button, it makes a request on a URL like this:

```
get?input1=2
```

It passes the value of the input field in the URL. When you use these forms with the ESP32 or ESP8266, you can get the input field's value from that request.

There are many different input types, not just text. Here's a list:

- `<input type="button">`
- `<input type="checkbox">`
- `<input type="color">`
- `<input type="date">`
- `<input type="datetime-local">`
- `<input type="email">`
- `<input type="file">`
- `<input type="hidden">`
- `<input type="image">`
- `<input type="month">`
- `<input type="number">`
- `<input type="password">`
- `<input type="radio">`
- `<input type="range">`
- `<input type="reset">`
- `<input type="search">`
- `<input type="submit">`
- `<input type="tel">`
- `<input type="text">`
- `<input type="time">`
- `<input type="url">`
- `<input type="week">`

We encourage you to try some of these input types to see what you get. You'll better understand how this works in the [Web Server Projects' Module](#).

HTML Input Types

An input type is a field where the user can enter or select data. To create an input, you use the `<input>` tag. The `<input>` tag doesn't have a closing tag.

The `<input>` tag doesn't need to be enclosed in a `<form>` tag. If you are a web developer, it is a good practice to do that. However, for our web server projects, where we want to keep things as simple as possible, that may not be necessary.

As seen previously, the `type` attribute is used to determine what type of element the input tag creates on the page. There are many types of inputs, as you've seen previously, but these are the most relevant for our web server projects:

- `<input type="text">` creates a field where the user can enter text.
- `<input type="number">` creates a field where the user can enter a number.
- `<input type="button">` creates a button.
- `<input type="checkbox">` creates a checkbox.
- `<input type="range">` creates a range field like a slider.

Each of these input types has further attributes to describe them. For example, the input `value` attribute specifies an initial value for an input field (default value).

Additionally, some input types have specific attributes. For example, the `<input type="number">` and the `<input type="range">` also have the following attributes:

- `max` - specifies the maximum value allowed.
- `min` - specifies the minimum value allowed.
- `step` - specifies the legal number intervals.
- `value` - specifies the default value.

If you want to know all the attributes for your input types, you just need to search for the input type you're interested in, followed by the word "attributes".

HTML Responsive Web Design

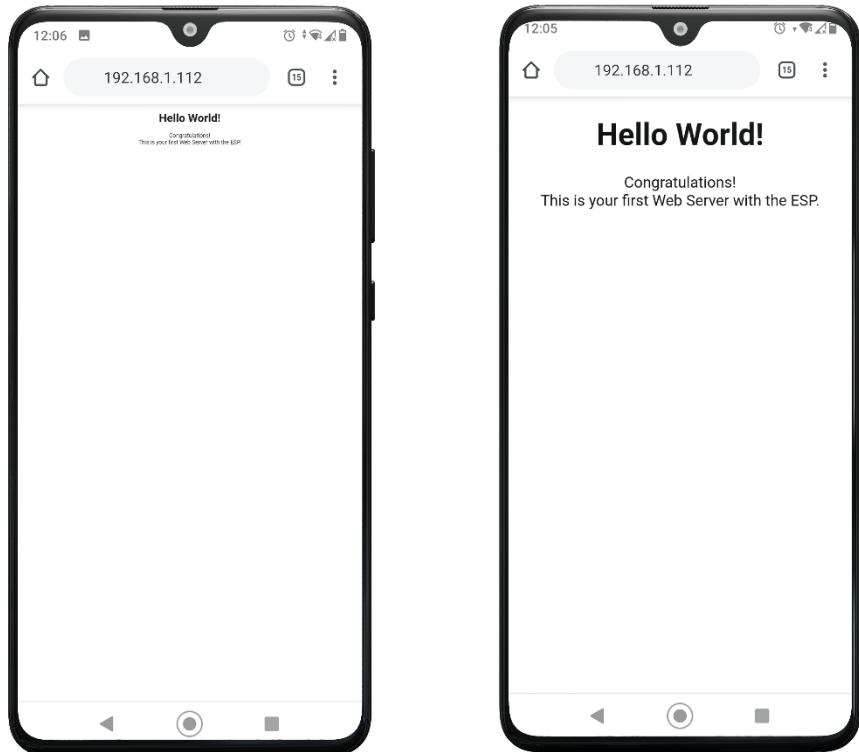
Responsive web design is about creating web pages that render well on various devices and window or screen sizes like your laptop, tablet, and smartphone.

Responsive web design uses HTML and CSS to automatically resize or hide elements to work properly on all devices. To create a responsive web page, add the following `<meta>` tag to the head of your HTML document:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

This will set the viewport of your page, which will give the browser instructions on how to control the page's dimensions and scaling.

For example, at the left, you can see a web page without this meta tag and at the right with the meta tag.



The <div> Tag

The `<div>` tag defines a section in an HTML document. It is a container for HTML elements that can then be styled with CSS or manipulated with JavaScript. For example, we may want to have several paragraphs inside a `<div>` tag with a yellow background. Putting all the paragraphs inside that `<div>` tag will make formatting easier.

Usually, you'll have more than one `<div>` section in your HTML document, so you should specify the `class` or `id` attributes so that you can refer to each of them individually.

You'll better understand how this works when you start learning CSS and JavaScript.

The class Attribute

The `class` attribute specifies one or more class names for an element. Class names are useful to define styles in CSS. For example, to style a group of HTML elements with the same style, we can give them the same class name.

To add the `class` attribute to an HTML element, you use the following syntax:

```
<element class="classname"></element>
```

An HTML element can have more than one class name, that must be separated by a space. In the following example we specify the class name `button` for the ON button. For the OFF button we specify both the `button` and `button2` classes.

```
<p><button class="button">ON</button></p>
<p><button class="button button2">OFF</button></p>
```

The id Attribute

The HTML `id` attribute is used to specify a unique id for an HTML element. The `id` must be unique for each element.

The `id` attribute is used to point to a specific style declaration in a style sheet or it is also used by JavaScript to access and manipulate the element with the specific id.

For example, the `id` of the following button is `mybutton`.

```
<p><button id="mybutton">ON</button></p>
```

Difference Between class and id

What's the main difference between `class` and `id`? A `class` name can be used by multiple HTML elements that you want to share the same style. The `id` name must be used by only one HTML element within the page.

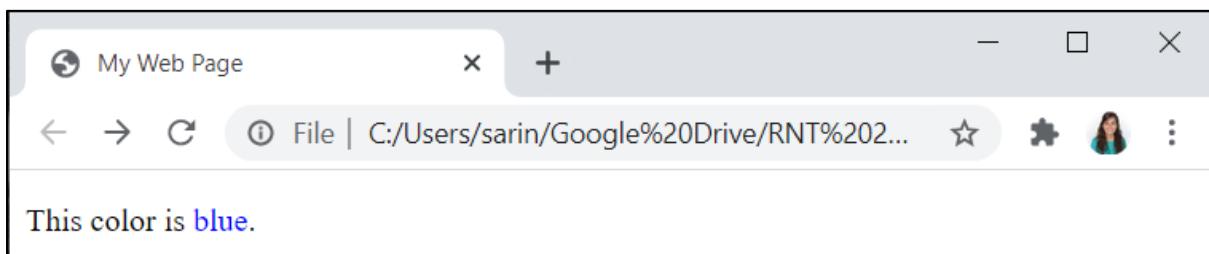
You'll better understand this when you start learning CSS and JavaScript. For now, you just need to understand that the `class` attribute can be shared by several HTML elements and that an `id` is unique for each HTML element.

The `` Tag

The `` tag defines an inline container. It is easily styled by CSS or manipulated with JavaScript using the `class` or `id` attributes.

Imagine that you have a paragraph and want to give a specific style to some parts of that paragraph. You may put those parts within `` tags. For example, the next line sets the “blue” word with the blue color and not the whole paragraph.

```
<p>This color is <span style="color:blue;">blue</span>.</p>
```



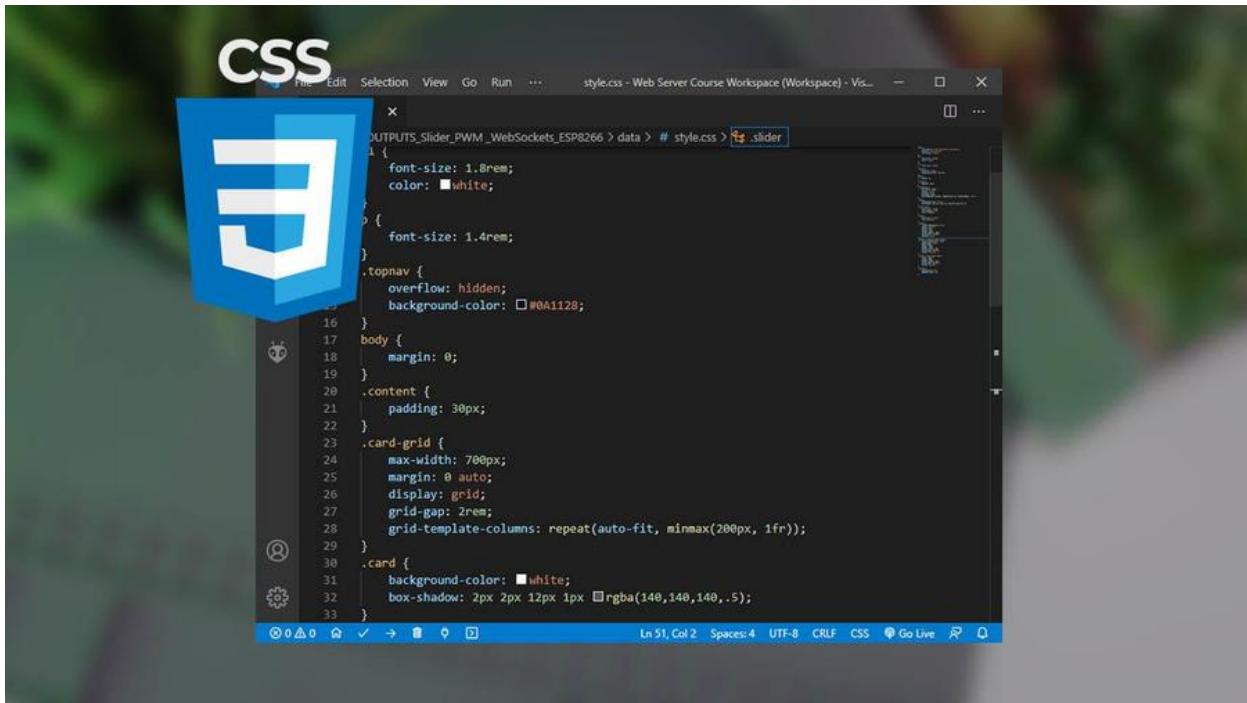
In our examples, we'll use the `` tag often to give a unique `id` to some parts of the text we want to manipulate using JavaScript.

Wrapping Up

In this section, you've learned some HTML basics so that you're able to create your web pages for your web server projects. Now that you understand how HTML works, you can experiment with building a web page by adding some of the HTML tags you already know.

HTML doesn't do much besides adding raw text and structuring your web page elements, so at the moment, the page may not look very pretty. In the "Styling HTML Content with CSS" Unit, you'll find out how to dress it up.

Styling HTML Content with CSS



HTML doesn't do much besides adding raw text to a web page and structuring your elements. If you've been experimenting with the examples provided, your web pages are basic, with no color or customization. The examples merely contain the text marked with HTML. Using CSS, you can add colors and change the layout to make it look prettier.

CSS stands for Cascading Style Sheets, and it is a style sheet language used to describe how the elements in a web page will look when rendered.

You can add CSS directly to the HTML file or in a separate file that you reference in the HTML file. You'll learn both ways.

Download Example Files

You can download the files for all the examples presented throughout this section at the following link:

- [CSS Examples](#)

Knowing the Basics of CSS

Each CSS statement comprises two parts: a *selector* followed by one or more *instructions* within brackets. Each instruction is itself made up of two parts: a *property* and a *value* (with a colon separating the two parts and each instruction ending with a semi-colon). A CSS statement, therefore, has the following style:

```
selector {  
    property: value;  
}
```

The *selector* is the HTML element you want to style. *Property* is the name of the item you want to alter, and *value* is the how you want that property to appear. All instructions end with a semicolon.

For example, the following line will format all `h1` HTML elements with the navy color.

```
h1 {  
    color: navy;  
}
```

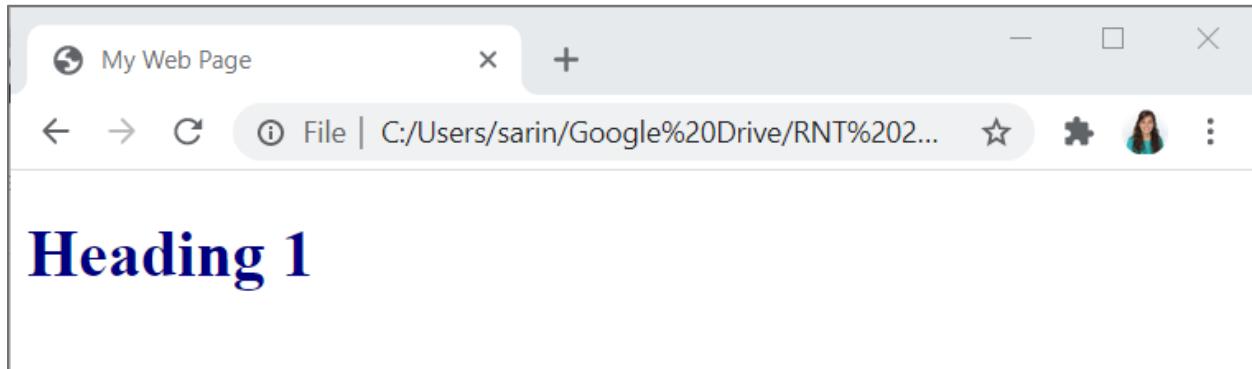
The CSS to style your web page can be stored inside your HTML document enclosed by the `<style></style>` tags inside the `<head></head>` tags, or you can place all your CSS in a separate `.css` file that you reference in the HTML document. It's up to you how you organize your files. We'll show you both ways.

The following example adds the styles for `h1` to the header of your HTML document.

```
<!DOCTYPE html>  
<html>  
    <head>  
        <title>My Web Page</title>  
        <meta name="viewport" content="width=device-width, initial-scale=1.0">  
        <link rel="icon" href="data:,">  
        <style>  
            h1 {  
                color: navy;  
            }  
        </style>  
    </head>  
    <body>  
        <h1>Hello World!</h1>  
    </body>  
</html>
```

```
</style>
</head>
<body>
  <h1>Heading 1</h1>
</body>
</html>
```

Save your file and open it in your web browser. Now, all your elements that use the `<h1>` tag are navy blue.



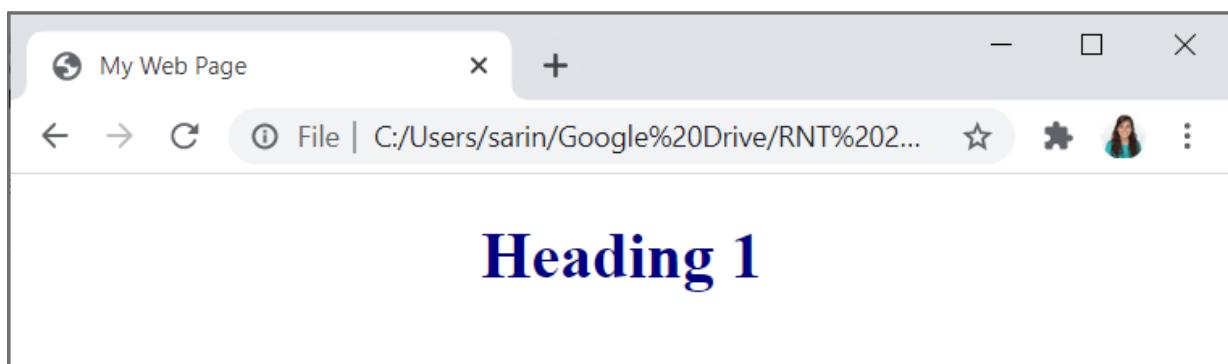
You can have more than one instruction for the same selector, as follows:

```
selector {
  property: value;
  property: value;
}
```

For example, add the following to your styles:

```
h1 {
  color: navy;
  text-align: center;
}
```

Now, your heading 1 HTML elements are in navy blue color and aligned at the center.



CSS ignores extra spaces, so nothing stops you from, say, having all your instructions on the same line. However, we suggest that you organize your CSS as in the preceding example to keep it cleaner and user friendly. It's also the standard way of writing CSS.

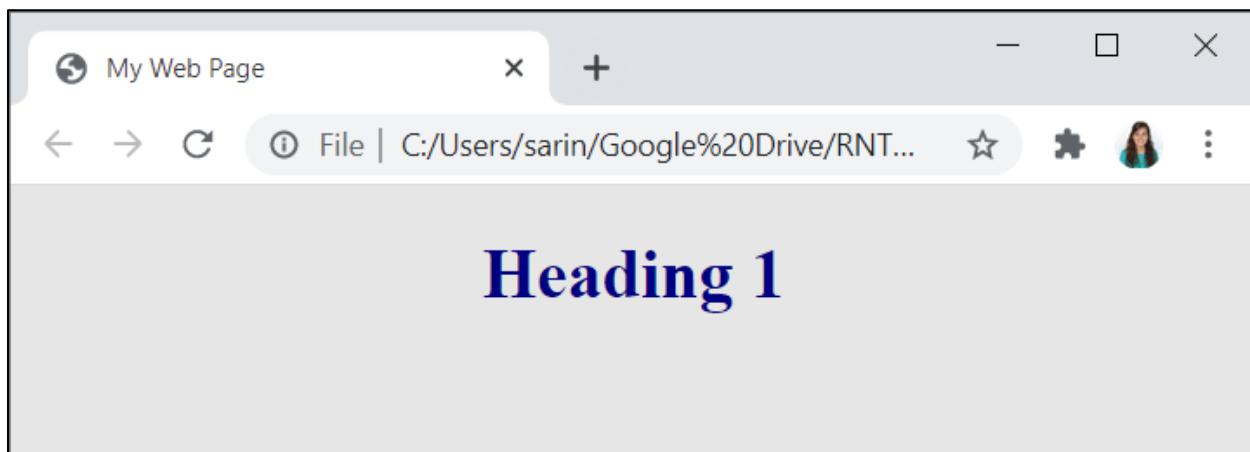
You can select more than one HTML element at the same time. The following example changes the color to navy for all elements that use the `<h1>` or `<h2>` tags:

```
h1, h2 {  
    color: navy;  
}
```

Experimenting With Colors

In the preceding section, you change all your headings to navy blue. You can also change the color of your background with the following instruction:

```
body {  
    background: #E6E6E6;  
}
```

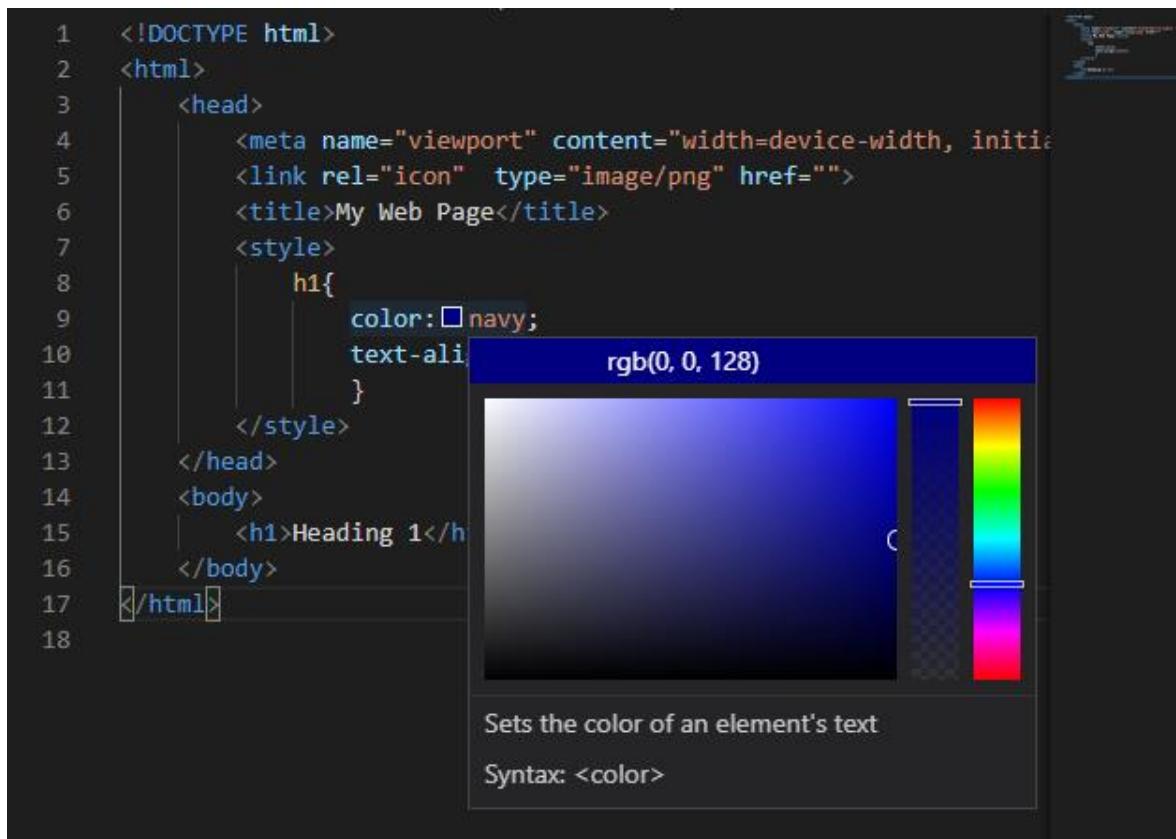


What do those numbers and letters mean? They're hexadecimal numbers, used in this case to specify combinations of red, green, and blue (RGB). Hexadecimal values range from 0 to 9 and from A to F. The lowest value for a hexadecimal number is 0, and the highest is F.

From 0 to F, there are a total of 16 values for each hexadecimal digit. All web browsers support 140 color names, which means that each color name has a hexadecimal color value. Following are some of the most basic values:

- Black: #000000
- Red: #FF0000
- Green: #00FF00
- Blue: #0000FF
- White: #FFFFFF

To find the right color, you don't need to keep trying combinations of numbers and letters. When you insert a color, VS code shows a little square with the color. If you hover your mouse over the name of the color, a color picker shows up where you can choose the desired color. Additionally, you can also select hexadecimal, RGB or other formats.



Changing Text Appearance

When you're working with HTML, you can customize your text to look exactly as you want. HTML is like a word processor with all the options you're already familiar with: font family, font size, text position, and so on. Instead of using a graphical user interface (GUI), you need to write instructions in your stylesheet using CSS. Here are the most relevant text properties you can modify:

- `text-align`: Sets where your text is aligned horizontally. You can set your text `left`, `center`, `right`, or `justified`;
- `text-decoration`: Removes or sets text decorations. You can use any of four values: `none`, `underline`, `overline`, and `line-through`;
- `font-family`: Changes the default font for the text;
- `font-style`: Changes the style of the text to `italic`, `bold`, or `normal`;
- `font-size`: Increases or decreases the sizes of letters;
- `font-weight`: Specifies the weight of a font. The options are `normal`, `bold`, `bolder`, and `lighter`.

VS Code shows the possible options for all these properties, making your life easier when choosing the properties' values. You can also change the indentation, change capitalization, and make plenty of other text customizations. If you want to know how to do something specific, simply search for it online.

Having the following HTML template, play with the text appearance properties to change how the web page looks.

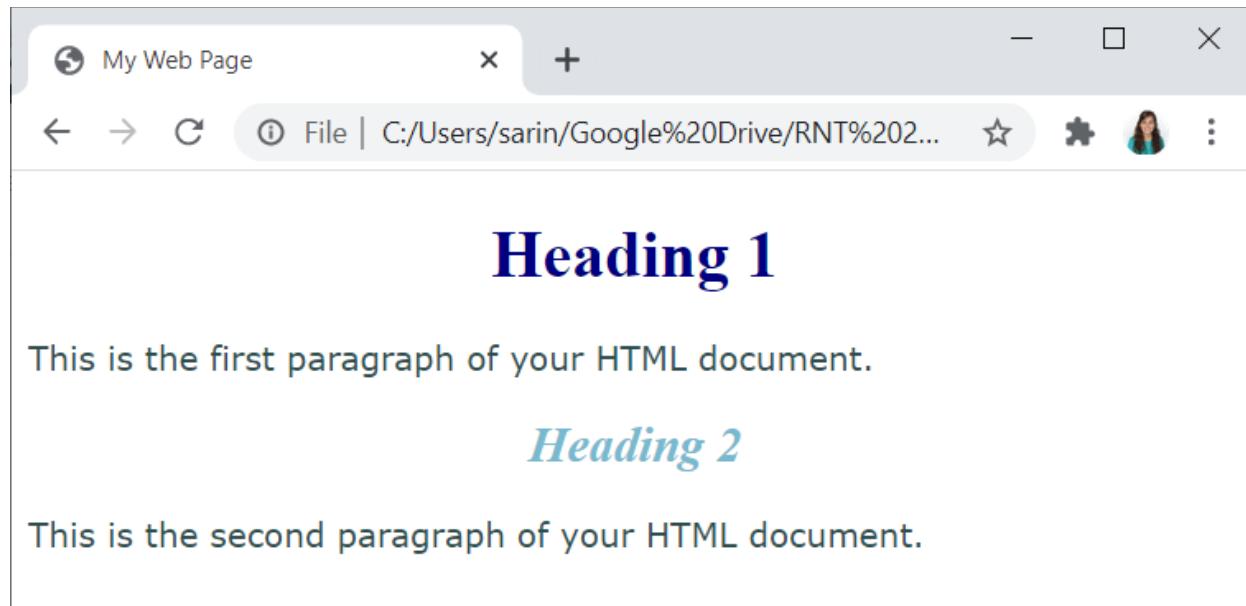
```
<!DOCTYPE html>
<html>
  <head>
    <title>My Web Page</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,>
    <style>
      h1 {
```

```

        color: navy;
        text-align: center;
    }
h2 {
    color: rgb(119, 185, 206);
    text-align: center;
    font-style: italic;
}
p {
    text-align: justify;
    font-family: Verdana, Geneva, Tahoma, sans-serif;
    color: darkslategray;
}
</style>
</head>
<body>
    <h1>Heading 1</h1>
    <p>This is the first paragraph of your HTML document.</p>
    <h2>Heading 2</h2>
    <p>This is the second paragraph of your HTML document.</p>
</body>
</html>

```

Here's the result of the previous example:



Heading 1 elements are set to navy color and aligned at the center.

```

h1 {
    color: navy;
    text-align: center;
}

```

Heading 2 elements are set with a light blue color, aligned at the center in *italics*.

```
h2 {  
    color: rgb(119, 185, 206);  
    text-align: center;  
    font-style: italic;  
}
```

Finally, the paragraph is set with a dark gray color, justified, and with Verdana font.

```
p {  
    text-align: justify;  
    font-family: Verdana, Geneva, Tahoma, sans-serif;  
    color: darkslategray;  
}
```

Usually, when using `font-family`, you should specify more than one font because you want to make sure that the browser supports at least one of the selected fonts. You define a priority in the preceding example. `Geneva` is the font only if the web browser doesn't have `Verdana`, and so on.

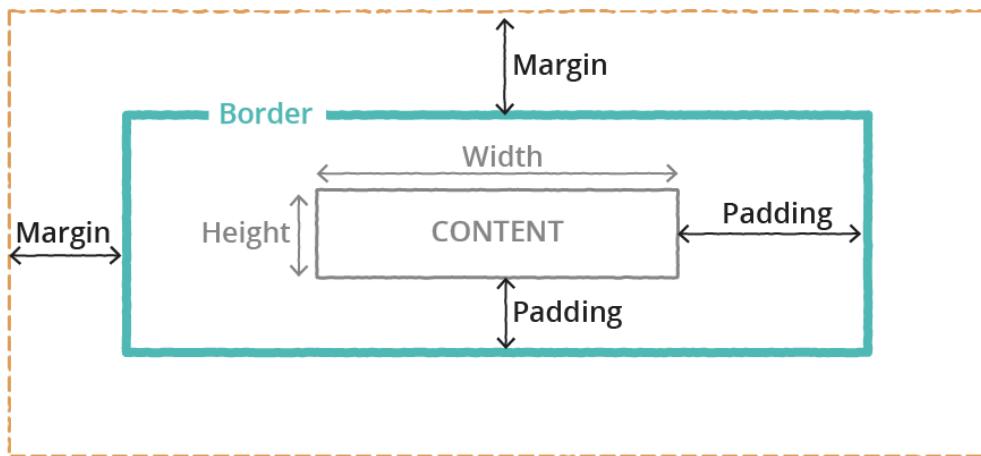
Understanding the Box Model

In discussions of CSS, you frequently hear the term *box model*. This important subject is one that a lot of people don't fully understand.

Because the layout determines the look and feel of your web page, controlling the position of your HTML elements is important, and that's done with CSS. You can think of each HTML element as being a box that holds your content. The CSS box model contains a few properties that help you position your HTML elements where you want them:

- **content**: Sets where your text, hyperlinks, or images appear;
- **width**: Sets the width in pixels;
- **height**: Sets the height in pixels;
- **padding**: Adds a layer of transparent space around your content box;
- **border**: Adds a border around your padding;
- **margin**: Adds a layer of transparent space around your border.

The following figure illustrates all these properties.



The CSS Box Model

To better understand the Box Model, we first need to understand the `<div>` tag. The `<div>` tag defines a section in an HTML document. It is a container for HTML elements that can be styled with CSS or manipulated with JavaScript.

For example, we may want to have several paragraphs or other HTML elements inside a `<div>` tag with a colored background. Putting all the paragraphs inside the `<div>` tag will make that easier. Also, it makes it easier to align elements on your page using CSS.

Usually, you'll have more than one `<div>` section in your HTML document. You should specify the `class` or `id` attributes so that you can refer to each of them collectively or individually.

To see the box model in action, you can create a new HTML document called `box_model.html` with the main structure of an HTML document and insert the following into the `<body>` tags:

```
<div>
  <p>Your content goes here</p>
</div>
```

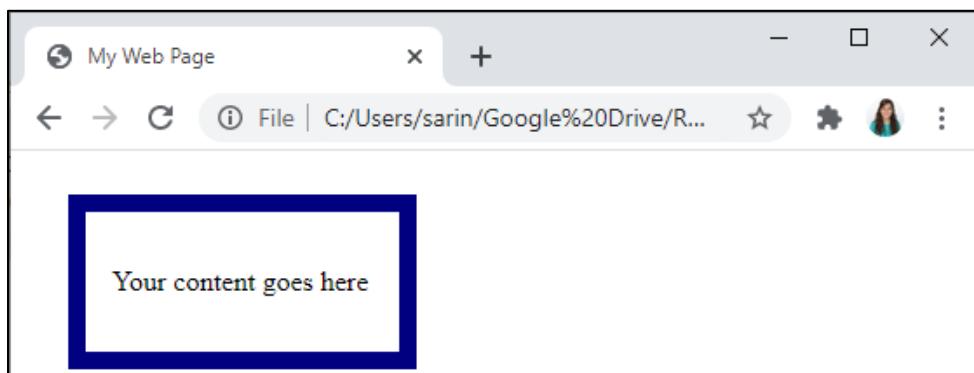
Then add these lines to the styles in your `<head>` section (between the `<style></style>` tags):

```
div {  
    width: 160px;  
    height: 60px;  
    padding: 15px;  
    border: 5px navy;  
    margin: 25px;  
    border-style: solid;  
}
```

Here's the complete HTML document:

```
<!DOCTYPE html>  
<html>  
    <head>  
        <title>My Web Page</title>  
        <meta name="viewport" content="width=device-width, initial-scale=1.0">  
        <link rel="icon" type="image/png" href="">  
        <style>  
            div {  
                width: 160px;  
                height: 60px;  
                padding: 15px;  
                border: 5px navy;  
                margin: 25px;  
                border-style: solid;  
            }  
        </style>  
    </head>  
    <body>  
        <div>  
            <p>Your content goes here!</p>  
        </div>  
    </body>  
</html>
```

You can see the CSS properties applied to your content in the next image.



Let's see what each line of CSS means:

- `div` → refers to the `<div>` HTML element
- `width: 160px;` → the width of your content
- `height: 60px;` → the height of your content
- `padding: 15px;` → padding around your content (space between the content and the border)
- `border: 5px navy;` → thickness and color of the border
- `border-style: solid;` → solid border, otherwise it won't show up
- `margin: 25px;` → margin 25px all around

There are endless ways to style the border of your content: borders with round corners, dotted, dashed, double borders, and much more. We won't cover all of them. We suggest that you search for "*CSS border properties*" and see all the possibilities.

The best way to understand how these previous properties work is by changing their values and see the results - that's what we recommend doing.

Usually, you'll have more than one `<div>` element on your page, and you'll want to style them differently. To do that, we use the `class` attribute. The `class` attribute specifies one or more class names for an element. The `class` attribute is mainly used to point to a class definition in a style sheet.

Give a class name to your `<div>`, for example "content". This is how it would look:

```
<div class="content">
  <p>Your content goes here!</p>
</div>
```

Then, to reference to that class in CSS, you should do it using a dot (.) before the class name (`.content`):

```
.content {
  width: 160px;
```

```
height: 60px;
padding: 15px;
border: 5px navy;
margin: 25px;
border-style: solid;
}
```

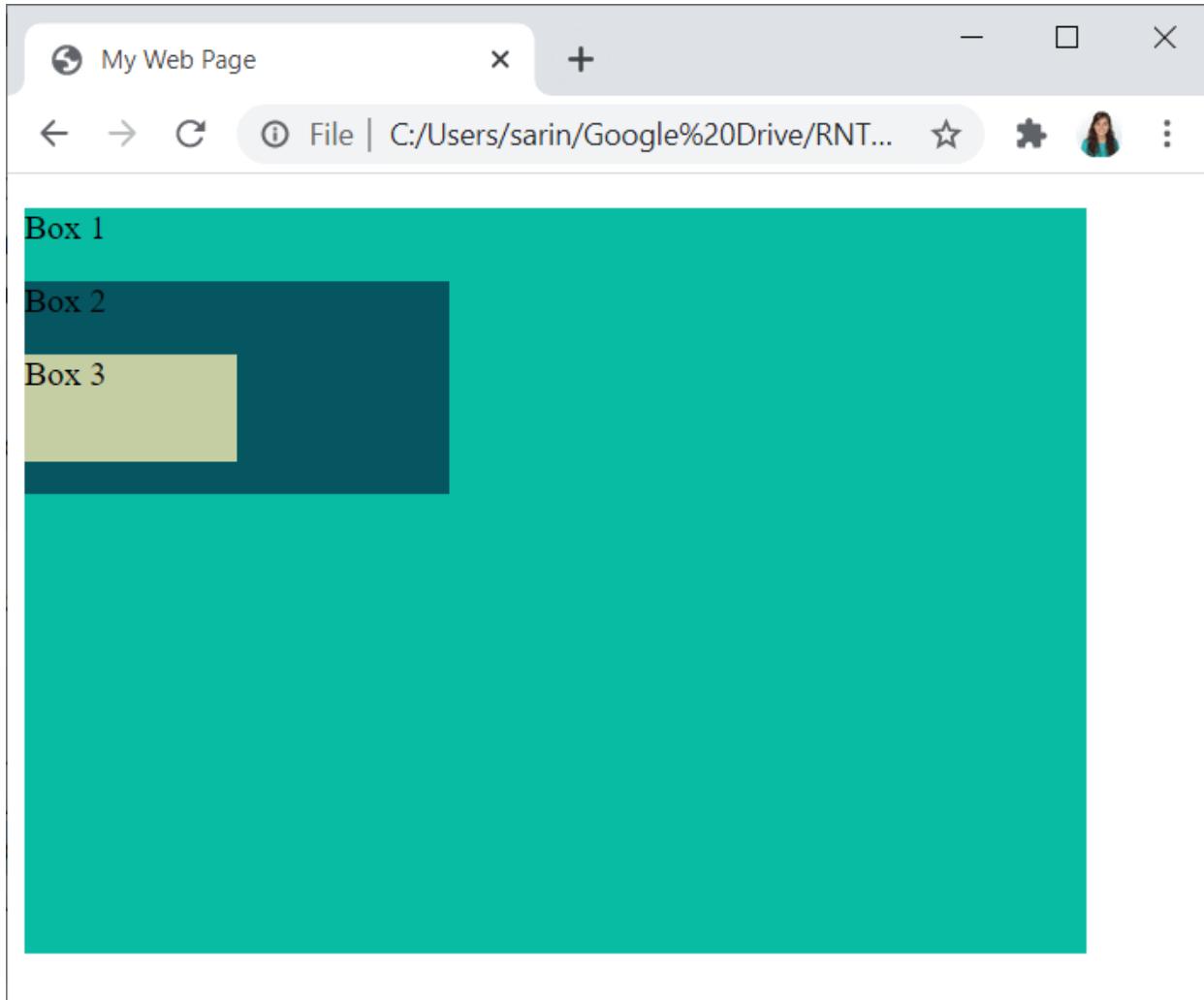
Save your page and you should get the same web page of the previous example.

Nested <div> Tags

You can have `<div>` tags inside `<div>` tags. For example, the following document creates boxes inside of boxes. It is pretty straightforward to understand.

```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP Web Server</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" href="data:,">
    <style>
      .div1 {
        width: 500px;
        height: 350px;
        background-color: rgb(8, 187, 163);
      }
      .div2 {
        width: 200px;
        height: 100px;
        background-color: rgb(6, 86, 97);
      }
      .div3 {
        width: 100px;
        height: 50px;
        background-color: rgba(218, 219, 170, 0.897);
      }
    </style>
  </head>
  <body>
    <div class="div1">
      <p>Box 1</p>
      <div class="div2">
        <p>Box 2</p>
        <div class="div3">
          <p>Box 3</p>
        </div>
      </div>
    </div>
  </body>
</html>
```

Here's what you should get when you open this document in your web browser.



In this example, we create a `<div>` tag with the class name `div1`. Inside `div1`, we create `div2`, and inside `div2`, we create `div3`.

Then, we style each `<div>` with different dimensions and colors using CSS. This results in boxes with different styles and colors.

Embedding a Style Sheet

As mentioned previously, you can include your CSS styles on the HTML document between the `<style></style>` tags, or you can have them in a separate CSS file,

which may be more practical. To do that, first, you need to reference the CSS file in your HTML document.

Open a new file in your text editor and make sure it is saved under the same folder that your HTML file. Name it *stylesheet.css*, and save it. Your filename should have the .css extension; otherwise, it won't work correctly.

To embed a style sheet into your HTML document, type the following between the `<head>` tags of your document:

```
<link rel="stylesheet" type="text/css" href="stylesheet.css">
```

This `<link>` tag tells the HTML file that you're using an external style sheet to format how the page looks.

The `rel` attribute specifies the nature of the external file. In this case, it is a style sheet—the CSS file—that will be used to alter the page's appearance. The `type` attribute is set to "text/css" to indicate that you're using a CSS file for the styles. The `href` attribute indicates the file location; since the file is in the same folder as the HTML file, you just need to reference the filename. Otherwise, you need to reference its file path.

Now you have your style sheet connected to your HTML document. This process saves you a ton of time because you can embed the same style sheet on all your web pages.

If you want to include the stylesheet, the head of your HTML document should look like this:

```
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" href="data:,">
  <link rel="stylesheet" href="stylesheet.css" type="text/css">
</head>
```

In the CSS, you should copy the styles (without the `<style></style>` tags). For example:

```
.div1 {  
    width: 500px;  
    height: 350px;  
    background-color: rgb(8, 187, 163);  
}  
.div2 {  
    width: 200px;  
    height: 100px;  
    background-color: rgb(6, 86, 97);  
}  
.div3 {  
    width: 100px;  
    height: 50px;  
    background-color: rgba(218, 219, 170, 0.897);  
}
```

When using two separate files, make sure you save them both to see all changes.

CSS Grid Layout

CSS Grid Layout is a two-dimensional grid-based layout system. It provides an easy way to position elements on your web page.

If you want to learn more about CSS Grid Layout, we recommend taking a look at the tutorial in the following link that takes an in-depth look at this complex topic but with simple and practical examples:

- <https://css-tricks.com/snippets/css/complete-guide-grid/>

For you to understand the basics of CSS Grid Layout we'll build a simple example with boxes.

CSS Grid Layout Example

Create 6 boxes with the `<div>` tag and put some text inside. To visualize the boxes, set a background color.

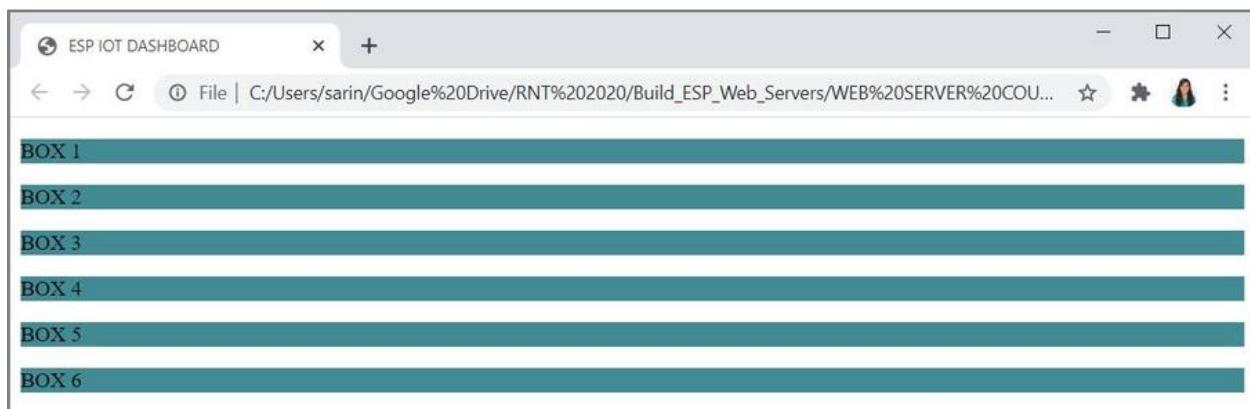
For example, the body of your HTML document can be as follows:

```
<body>
  <div class="box">
    <p>BOX 1</p>
  </div>
  <div class="box">
    <p>BOX 2</p>
  </div>
  <div class="box">
    <p>BOX 3</p>
  </div>
  <div class="box">
    <p>BOX 4</p>
  </div>
  <div class="box">
    <p>BOX 5</p>
  </div>
  <div class="box">
    <p>BOX 6</p>
  </div>
</body>
```

Here's what you should put in your CSS file to set the boxes' background color (you can choose any other color):

```
.box {
  background-color: rgb(66, 138, 148);
}
```

This creates 6 divisions on your web page with some text inside (let's call them boxes).



The boxes are spread across the browser window width, which doesn't look nice at all. Put all the boxes inside a container (a new div) called "mygrid". Adjust the

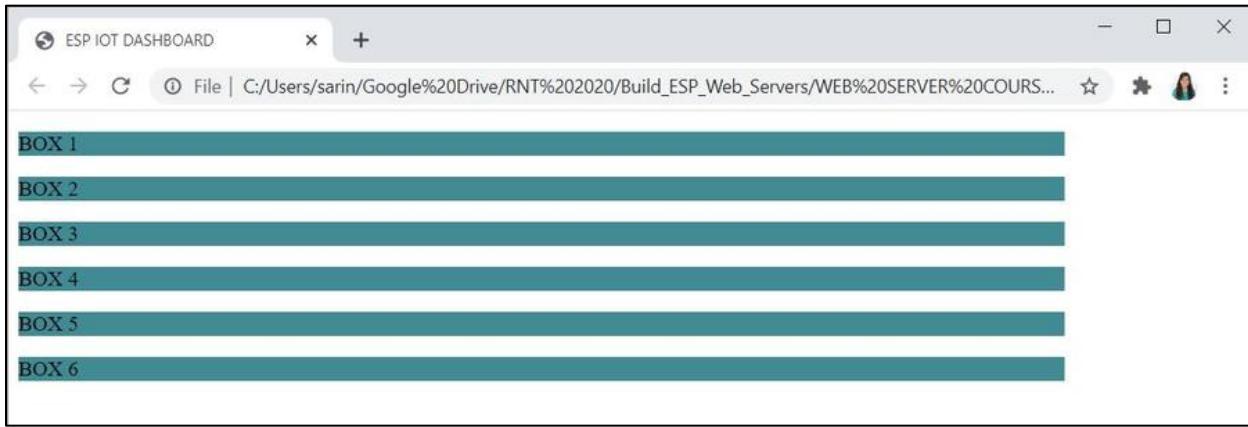
`max-width` of the grid. This prevents the boxes from spreading across the width of your web browser page. Here's the HTML file with the new `<div>` tag:

```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" href="data:,>">
    <link rel="stylesheet" type="text/css" href="stylesheet.css">
  </head>
  <body>
    <div class="mygrid">
      <div class="box">
        <p>BOX 1</p>
      </div>
      <div class="box">
        <p>BOX 2</p>
      </div>
      <div class="box">
        <p>BOX 3</p>
      </div>
      <div class="box">
        <p>BOX 4</p>
      </div>
      <div class="box">
        <p>BOX 5</p>
      </div>
      <div class="box">
        <p>BOX 6</p>
      </div>
    </div>
  </body>
</html>
```

And here's the CSS file to set a maximum width for your content:

```
.box {
  background-color: rgb(66, 138, 148);
}
.mygrid {
  max-width: 800px;
}
```

Save both files and open the HTML document in your browser. Now the boxes only fill 800px of the page.

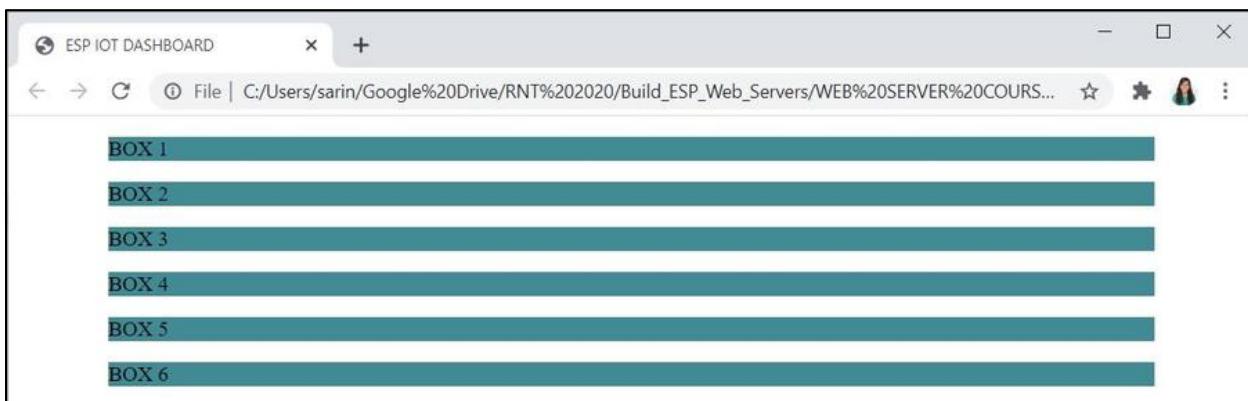


To place everything aligned at the center, set the `margin` of `mygrid` to "`0 auto`" like this:

```
.mygrid {  
  max-width: 800px;  
  margin: 0 auto;  
}
```

Setting the `margin` property to `0 auto` centers the division within its parent container. It is the same as setting the top and bottom margins to zero and the left and right margins to `auto`. The browser automatically distributes the right amount of margin to either side of the `mygrid` div.

So, at the moment, `mygrid` should be centered in the HTML body, as shown below.



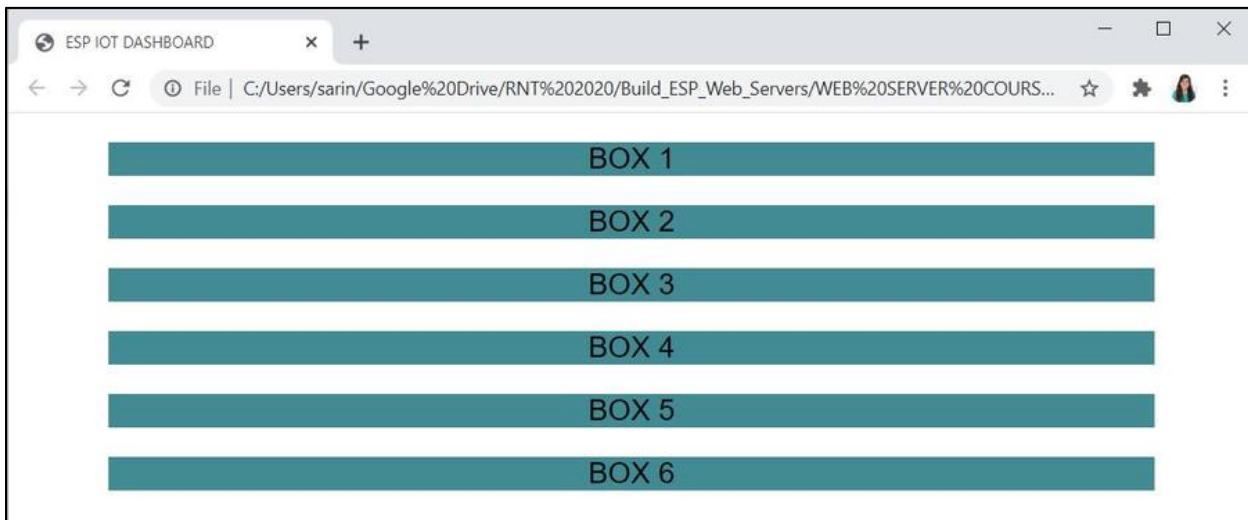
Add the following styles to the `<html>` tag to center all the text on the web page and set a different font type.

```
html {
```

```
font-family: Arial, Helvetica, sans-serif;  
display: inline-block;  
text-align: center;  
}
```

You can also increase the size of the paragraph font.

```
p {  
    font-size: 16px;  
}
```



The `mygrid` div is a container with 6 divs inside. So, it is our *grid container*. Add the `display`, `grid-gap` and `grid-template-columns` properties as shown below.

```
.mygrid {  
    max-width: 800px;  
    margin: 0 auto;  
    display: grid;  
    grid-gap: 2rem;  
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));  
}
```

The `display` property sets the `mygrid` element as a *grid container*. The `grid-gap` defines the space between each grid cell (box).

The `grid-template-columns` property sets the width of the grid's columns.

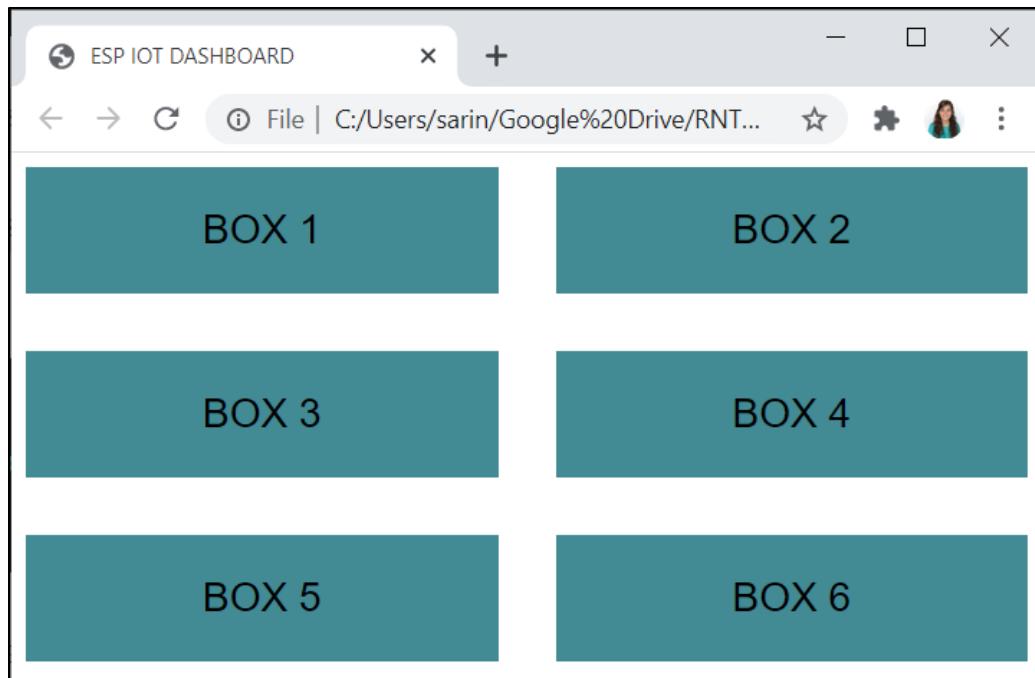
The `repeat` applies the same settings to all columns. The `minmax(200px, 1fr)` means that the width of the columns will adjust to the size of the browser window to

a minimum of 200px each — and since we have also set the `max-width` for `.mygrid` to 800px, we will get 3 and no more than 3 columns across the screen.

The `1fr` allows you to set the size of a track as a fraction of the free space of the grid container. Now, your boxes look like this:

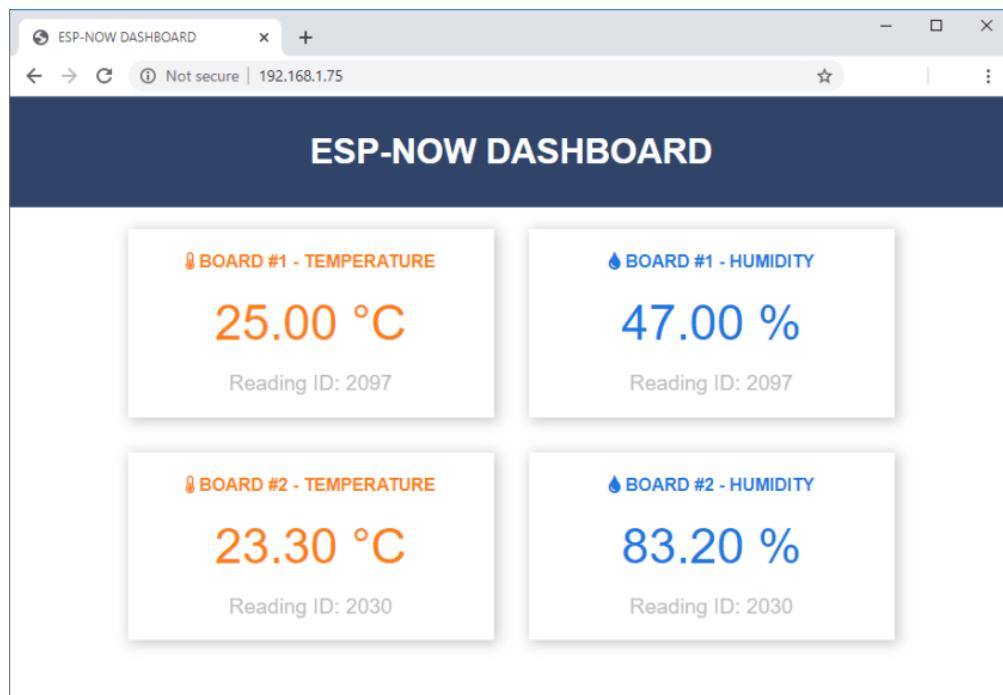


If you decrease the size of the web browser window, the boxes' size will adjust to a minimum of 200px until they break to the next row, as shown below.



In this example, our boxes display text, but you can place whatever you want inside, like buttons, tables, sliders, etc. This provides an easy way to place your web page elements in a beautifully modern and responsive layout.

For example, the cards that show sensor readings in the following web page are built using CSS grid.

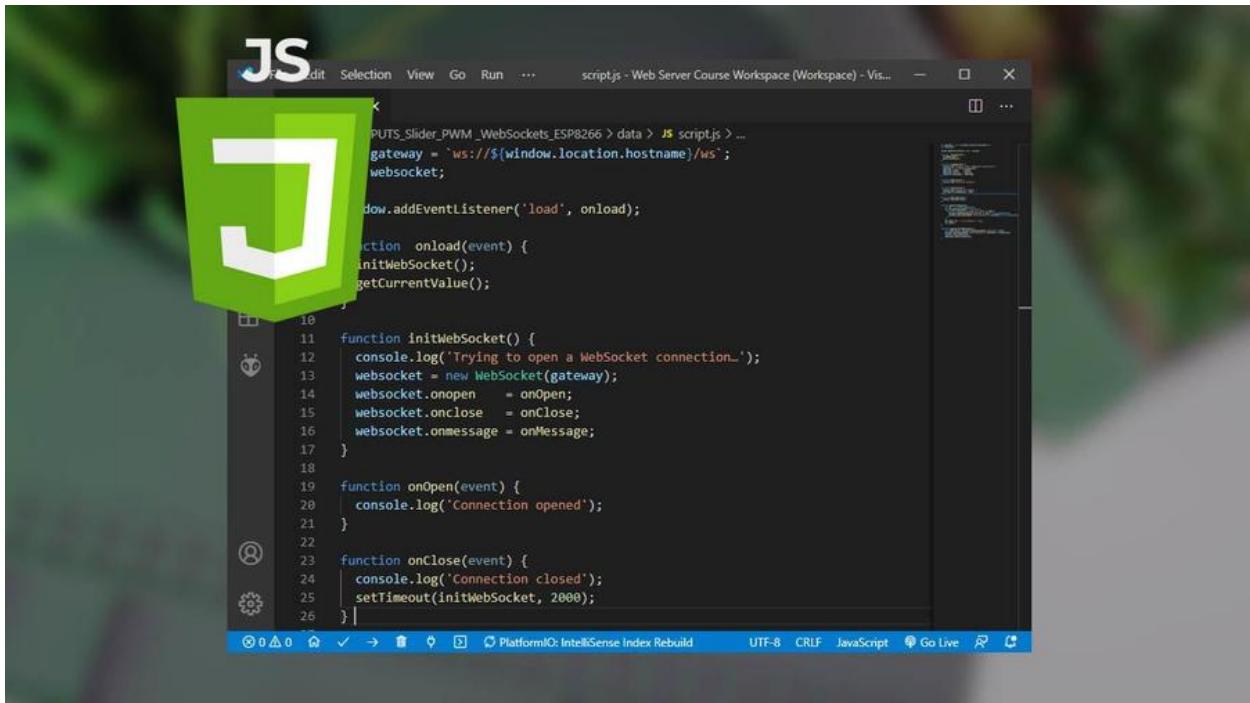


We encourage you to play with the grid settings to see how it behaves. We also recommend taking a look at the tutorial we've [suggested](#) to see what CSS grid allows you to do to place the elements within the HTML page.

Wrapping Up

In this section, you've learned the basics of CSS to format your web page and make it look pretty and organized. It is unlikely that you can memorize all properties and values for each element, so you'll often need to search online for the properties and values you want to set.

Getting Started with JavaScript



JavaScript runs on your browser's computer, tablet or smartphone. If you've ever visited a website that does really cool things through interactive buttons, sliders, gauges, charts, alert messages, or pop-up windows, some sort of JavaScript was certainly working in the back end.

Here's how JavaScript works in your web page: HTML tags create elements, and JavaScript lets you manipulate those elements. For example, with HTML you create a text field, and with JavaScript you can change the content of that text field dynamically.

Throughout this eBook, we'll use JavaScript mainly to make requests to the ESP32 or ESP8266 to get the latest sensor readings and update the corresponding text field with those values. Or to make requests for the state of a GPIO. JavaScript can do much more than that: you can change the color of a text field when its value crosses

a certain threshold, change the color of buttons dynamically, hide HTML elements, display pop-ups and alerts, and so much more.

While using JavaScript to make simple animations is easy, becoming fluent in JavaScript requires time and dedication. Here, we'll show you the basics so that you're able to build functional and dynamic web pages for your ESP32 and ESP8266 projects.

Download Example Files

You can download the files for all the examples presented throughout this section at the following link:

- [JavaScript Examples](#)

The `<script>` Tag

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags. These tags can be placed:

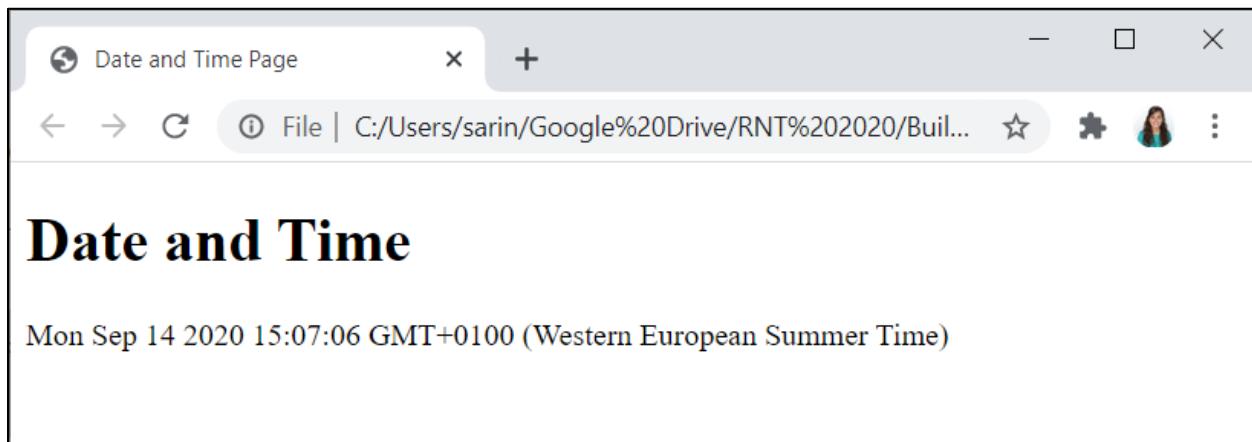
- In the head (`<head></head>`) section
- In the body (`<body></body>`) section
- After the body closing tag `</body>`

Your choice will depend on when you want JavaScript to load. Usually, it is placed inside the HTML head section. However, if you need it to run at a specific time to generate content, you should put it where it should be called, usually in the body section.

Let's take a look at an example. The following code shows the date and time on the browser. We get the date and time using JavaScript. We place the date and time inside a paragraph whose `id` is "dateTime".

```
<!DOCTYPE html>
<html>
  <head>
    <title>Date and Time Page</title>
  </head>
  <body>
    <h1>Date and Time</h1>
    <p id="dateTime"></p>
    <script>
      var todays_date = new Date();
      document.getElementById("dateTime").innerHTML = todays_date;
    </script>
  </body>
</html>
```

In this case, JavaScript is generating content, so it is placed within the `<body></body>` tags. If you've typed it in the head, it won't work. It would search for the element with `"dateTime"` `id`, and it wouldn't find it because it wouldn't have been loaded yet.



The previous figure shows what you get when you open the previous file in your browser. The date and time are updated every time you refresh the web browser.

Remember: The HTML `id` attribute is used to specify a unique `id` for an HTML element. The `id` must be unique for each element.

JavaScript uses the `id` to access and manipulate a specific HTML element (which is the case of the previous example).

Referencing an External JavaScript File

Instead of writing all the JavaScript code inside your HTML document, you can write your JavaScript code on a separate file that you reference on your HTML document. The JavaScript file should end with the `.js` extension.

For example, the following line references a JavaScript file called `script.js`. In this case, the file should be placed in the same folder that the HTML file. If it is placed in a different folder, you must reference the path where it is located.

```
<script src="script.js"></script>
```

To recreate the previous example, your HTML file would look as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Date and Time Page</title>
  </head>
  <body>
    <h1>Date and Time</h1>
    <p id="dateTime"></p>
    <script src="script.js"></script>
  </body>
</html>
```

And you should create a `script.js` file in the same folder with the following content:

```
var todays_date = new Date();
document.getElementById("dateTime").innerHTML = todays_date;
```

Save all your files and open your HTML file in your browser. You should get the same result.

As you can see, you can write the JavaScript in a separate file or directly in your HTML file. Use the method that best works for you.

Create a Variable

Variables are used to store data values. You create a variable in JavaScript as follows:

```
var state = "on";
```

Use the keyword `var` followed by the variable name and then assign its value. In this case, we're creating a variable called `state` and assigning the value `"on"` to it.

You can also declare a variable without a value. This is useful when the value will be calculated or provided later.

```
var state;
```

JavaScript Comments

To add a comment in your JavaScript file use double slashes `//` before your comment.

```
//This is a comment
```

Or for multiline comments as shown below:

```
/*This is a comment  
This line is also a comment  
And this one is also a comment*/
```

Comments can also be used to prevent a line from being executed. This is useful for debugging purposes.

JavaScript Operators

Like in other programming languages, JavaScript uses arithmetic, comparison, and logic operators as described in the following tables.

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Comparison Operators

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not equal
!==	Not equal value or not equal type
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
?	Ternary operator

Logical Operators

Operator	Description
&&	Logical AND
	Logical OR
!	Logical NOT

Changing HTML Content

As said previously, HTML tags create elements and JavaScript lets you manipulate those elements. Much of what you want to do with JavaScript is interact with elements created with HTML.

getElementById and innerHTML

One of the JavaScript HTML methods you'll use often is the `getElementById()`. This is used to "search" for an element with a specific id on your HTML document. Remember that the id must be unique for each HTML element.

In the previous example (date and time), `getElementById("dateTime")` searches for the HTML element with `id="dateTime"` and changes the element content using `innerHTML` to the current date and time.

```
document.getElementById("dateTime").innerHTML = todays_date;
```

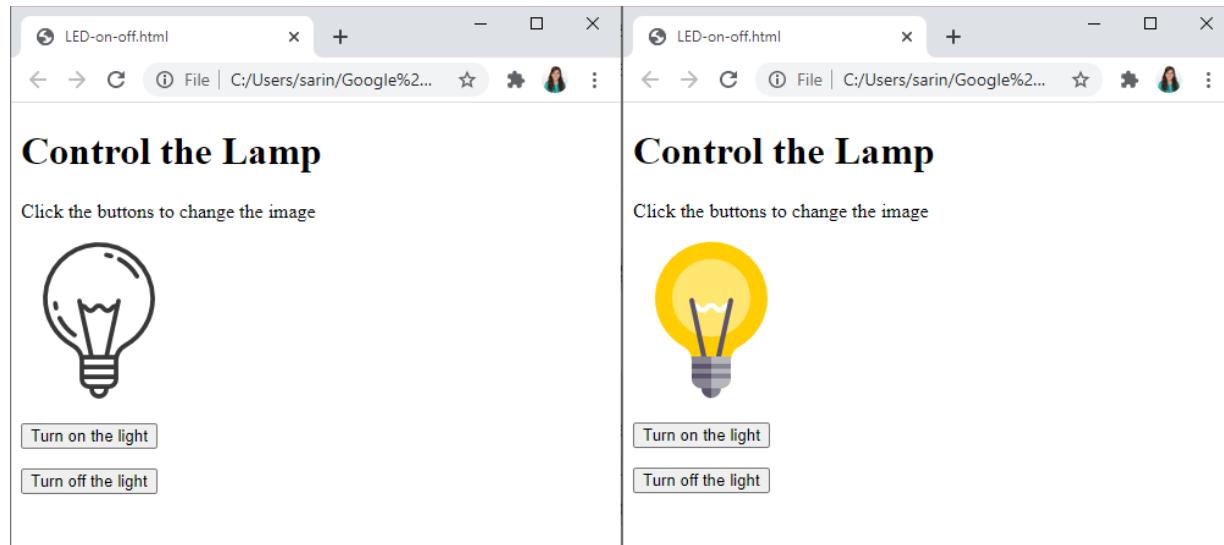
To access an HTML element, JavaScript can use the `document.getElementById(id)` method. The `innerHTML` property defines the HTML content.

Changing HTML Attribute Values

JavaScript can change HTML attribute values. In the following example JavaScript changes the value of the `src` (source) attribute of an `` tag:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Control the Lamp</h1>
    <p>Click the buttons to change the image</p>
    <p>
      
    </p>
    <p>
      <button onclick="document.getElementById('imageLamp').src='light_bulb_
      on.png'">Turn on the light</button>
    </p>
    <p>
      <button onclick="document.getElementById('imageLamp').src='light_bulb_
      off.png'">Turn off the light</button>
    </p>
  </body>
</html>
```

It gets the image element by its id (`imageLamp`) and changes its source by attributing a new value. You must have the images `light_bulb_on.png` and `light_bulb_off.png` in the HTML document folder for this example to work.



Press the on and off buttons to switch between images.

You can download the images using the next two links:

- [Lightbulb OFF](#)
- [Lightbulb ON](#)

Changing HTML Styles

JavaScript can change HTML styles (CSS). Every HTML element that you access via JavaScript has a style object. This object allows you to specify a CSS property and set its value. For example, to change the color of a paragraph with `id='text'`:

```
document.getElementById('text').style.color='blue'
```

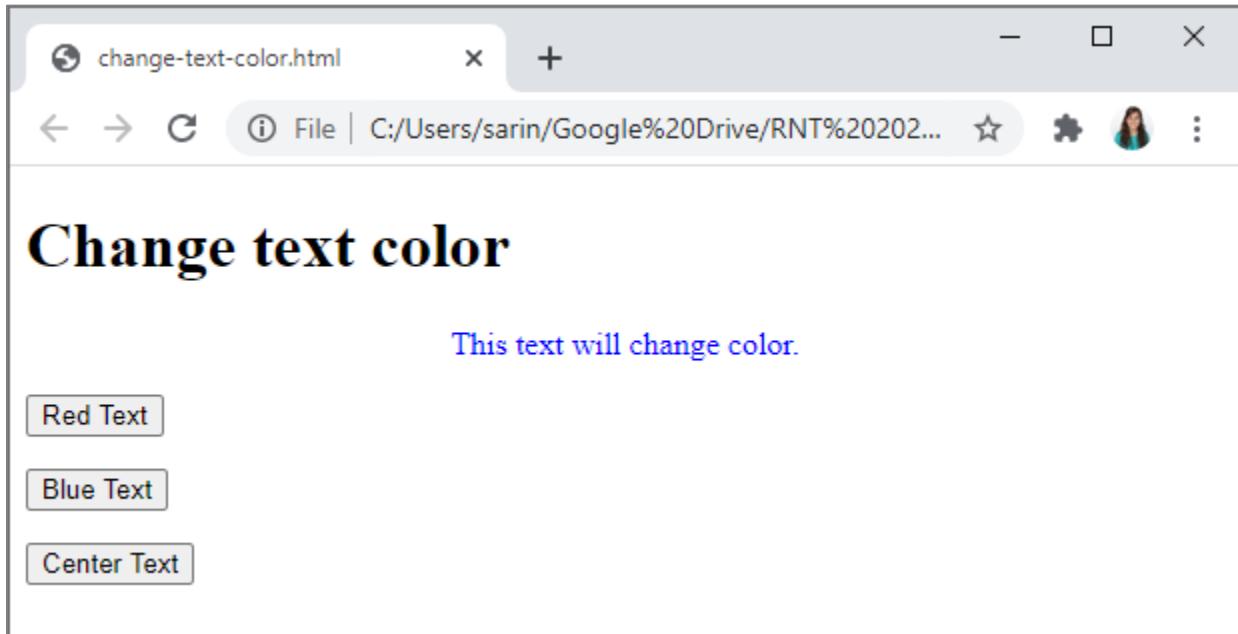
In a similar way, you can align the text to the center:

```
document.getElementById('text').style.textAlign='center'
```

The following example creates three buttons to change the text style.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Change text color</h1>
    <p id="text">This text will change color.</p>
    <p>
      <button onclick="document.getElementById('text').style.color='red'">
        Red Text</button>
    </p>
    <p>
      <button onclick="document.getElementById('text').style.color='blue'">
        Blue Text</button>
    </p>
    <p>
      <button onclick="document.getElementById('text').style.textAlign='center'">
        Center Text</button>
    </p>
  </body>
</html>
```

Save the file and open it in your browser. Click each button to change the text style.



Hiding HTML Elements

In some applications it might be useful to hide some HTML elements. That can be done by changing the display style to "none":

```
document.getElementById("text").style.display="none";
```

Alternatively, you can also show hidden HTML elements by changing the display style like this:

```
document.getElementById("text").style.display="block";
```

Try the following example to see how it works:

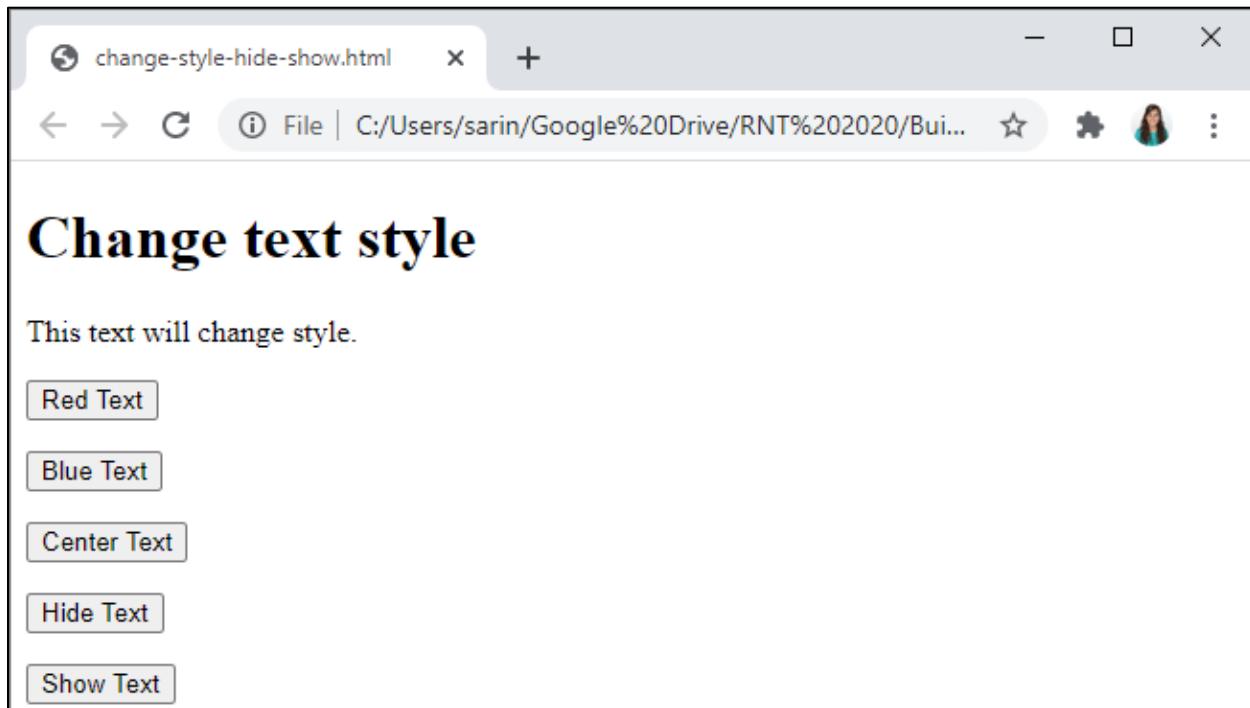
```
<!DOCTYPE html>
<html>
  <body>
    <h1>Change text style</h1>
    <p id="text">This text will change style.</p>
    <p>
      <button onclick="document.getElementById('text').style.color='red'">
        Red Text</button>
    </p>
    <p>
      <button onclick="document.getElementById('text').style.color='blue'">
        Blue Text</button>
    </p>
```

```

<p>
  <button onclick="document.getElementById('text').style.textAlign='center'">
    Center Text</button>
</p>
<p>
  <button onclick="document.getElementById('text').style.display='none'">
    Hide Text</button>
</p>
<p>
  <button onclick="document.getElementById('text').style.display='block'">
    Show Text</button>
</p>
</body>
</html>

```

Click the **Hide Text** and **Show Text** buttons to hide and show the text.



Functions

A JavaScript function is a block of JavaScript code that will be executed when called.

It can be called in the following circumstances:

- When an event occurs (for example, when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self-invoked)

A JavaScript function is defined with the `function` keyword, followed by a name, followed by parentheses `()`. The parentheses may include parameter names separated by commas: `(parameter1, parameter2, ...)`. The code to be executed, by the function, is placed inside curly brackets: `{}`.

For example, the next function returns the product of two numbers:

```
function myFunction(p1, p2) {
    return p1 * p2; // The function returns the product of p1 and p2
}
```

Here's another example: the following HTML document shows the temperature in Celsius degrees. When you click a button, it converts the current value to Fahrenheit.

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Convert Celsius to Fahrenheit</h1>
        <p>Temperature:</p>
        <p id="temperature">25</p>
        <p><button onclick="convertToF()">Convert to Fahrenheit</button></p>
        <script>
            function convertToF() {
                var temperatureC = document.getElementById("temperature").innerText;
                console.log(temperatureC);
                var temperatureF = ((temperatureC)*9/5) + 32;
                document.getElementById("temperature").innerHTML=temperatureF;
            }
        </script>
    </body>
</html>
```

Let's take a closer look on how this example works.

There's a paragraph that displays the temperature value. The `id` of that paragraph is `temperature`.

```
<p id="temperature">25</p>
```

The button calls the `convertToF()` JavaScript function when clicked:

```
<p><button onclick="convertToF()">Convert to Fahrenheit</button></p>
```

Our JavaScript function goes between the `<script></script>` tags.

First, we create a variable called `temperatureC` that saves the temperature displayed on the paragraph with `id="temperature"`.

```
var temperatureC = document.getElementById("temperature").innerText;
```

The following line displays the value of the `temperatureC` variable in the browser console.

```
console.log(temperatureC);
```

To open the browser console, click **Ctrl+Shift+J** in your browser (CMD+ALT+J on Mac)—we'll take a look at this in the next section.

Then, create another variable called `temperatureF` to save the temperature value in Fahrenheit.

```
var temperatureF = ((temperatureC)*9/5) + 32;
```

Finally, replace the temperature value in the paragraph with the `temperatureF` variable.

```
document.getElementById("temperature").innerHTML = temperatureF;
```

After saving the file, open it in your browser. Click on the button to convert the temperature to Fahrenheit degrees.

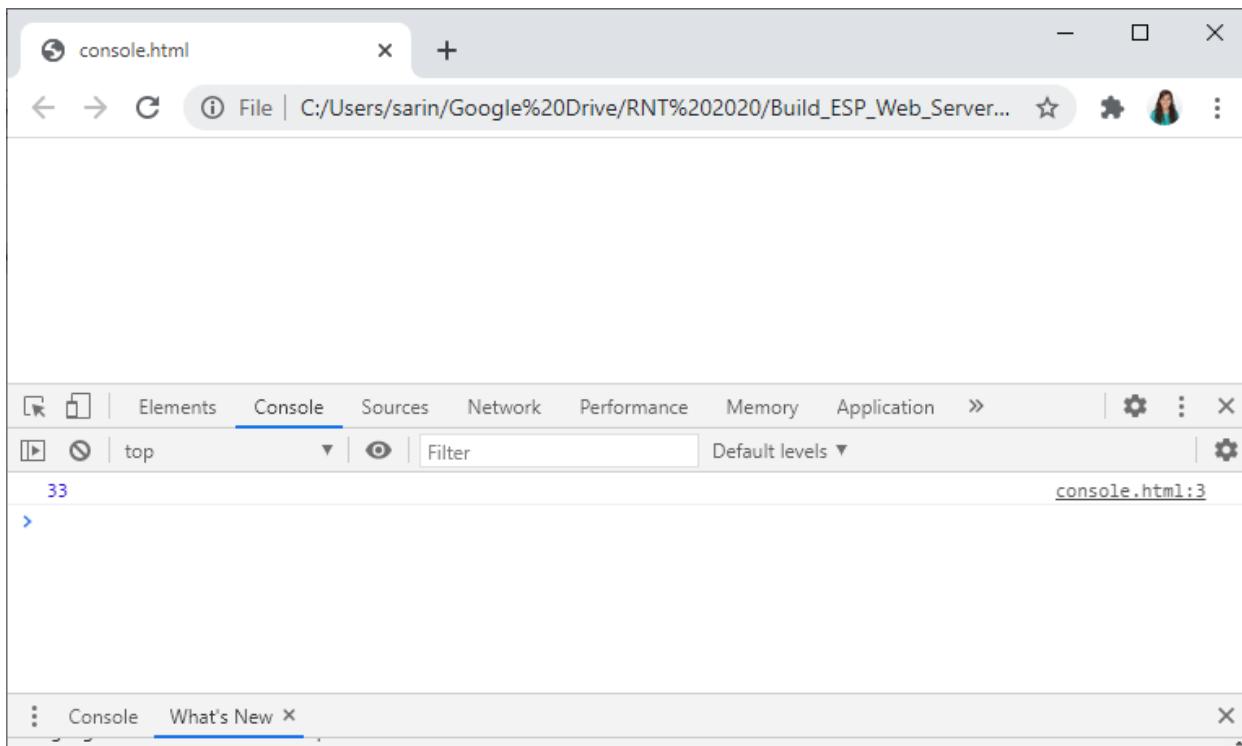


console.log() and window.alert()

For debugging purposes, you can call the `console.log()` method to display data in your web browser console window. Inside `log()`, you pass whatever you want to print. For example:

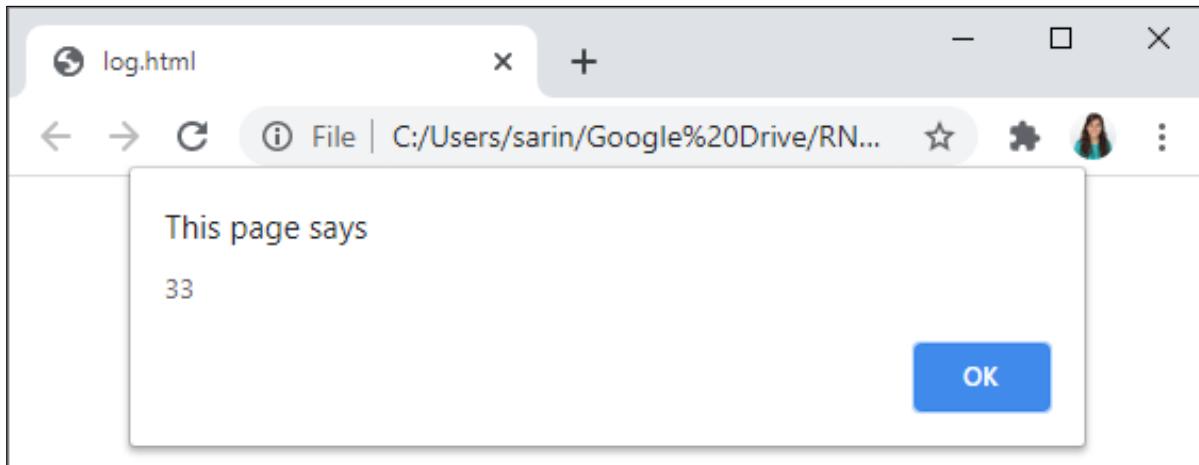
```
<script>
  var a = 33;
  console.log(a);
</script>
```

This prints the value of the variable `a` in the browser console. In your browser, press **Ctrl+Shift+J** (Windows) or **CMD+ALT+J** (Mac) to open the console. You should see 33 printed.



Alternatively, you can also use `window.alert()` in a similar way. It pops up an alert box with whatever data you want.

```
<script>
  var a = 33;
  window.alert(a);
</script>
```



JavaScript Events

Events are things that happen to HTML elements. JavaScript can do something when those events happen. It lets you execute code when events are detected.

Events can be something that the user does, like clicking on a button, or something that the browser does, like reloading a web page or receiving data from the server.

You can add event handlers to your HTML elements. We've done that previously with buttons. For example:

```
<p><button onclick="convertToF()">Convert to Fahrenheit</button></p>
```

This particular example handles what happens when the `onclick` event occurs. In this case it calls the `convertToF()` JavaScript function.

Here's the footprint to add event handlers:

```
<element event='some JavaScript'></element>
```

In the previous example, we had the following:

- element: `button`
- event: `onclick`
- JavaScript: `convertToF()`

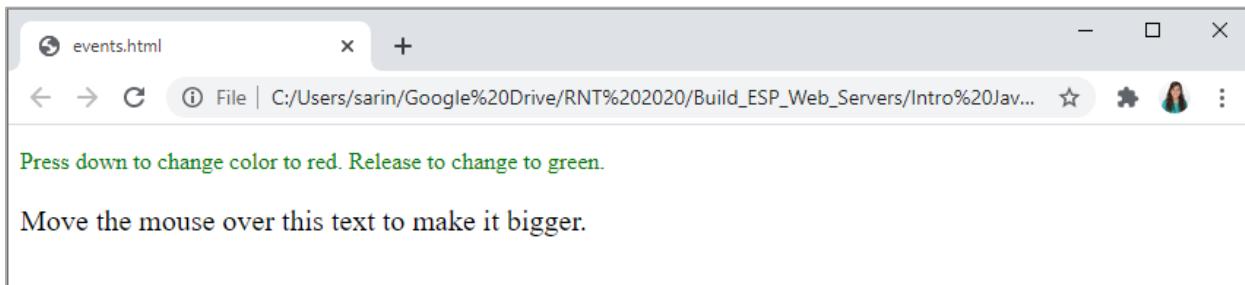
Here's a list of the common events, but there are a lot more:

- `onchange`: an HTML element has been changed (drag a slider, for example)
- `onclick`: the user clicks an HTML element
- `onmouseover`: the user moves the mouse over an HTML element
- `onmousedown`: when you press the mouse over an element
- `onmouseup`: when you release your mouse over an element
- `onmouseout`: the user moves the mouse away from an HTML element
- `onkeydown`: the user pushes a keyboard key
- `onload`: the browser has finished loading the page

Try the following example to see how some of these events work:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="text1" onmousedown="mouseDown()" onmouseup="mouseUp()">
      Press down to change color to red. Release to change to green.</p>
    <p id="text2" onmouseover="mouseOver()" onmouseout="mouseOut()">
      Move the mouse over this text to make it bigger.</p>
    <script>
      function mouseDown() {
        document.getElementById("text1").style.color = "red";
      }
      function mouseUp() {
        document.getElementById("text1").style.color = "green";
      }
      function mouseOver() {
        document.getElementById("text2").style.fontSize = "20px";
      }
      function mouseOut() {
        document.getElementById("text2").style.fontSize = "16px";
      }
    </script>
  </body>
</html>
```

Save the file and open it in your browser. Click on the first paragraph to change its color and move the mouse over the second paragraph to make it bigger.



Wrapping Up

Our aim with this Unit is to show you the basics of JavaScript to better understand the web server Projects later on.

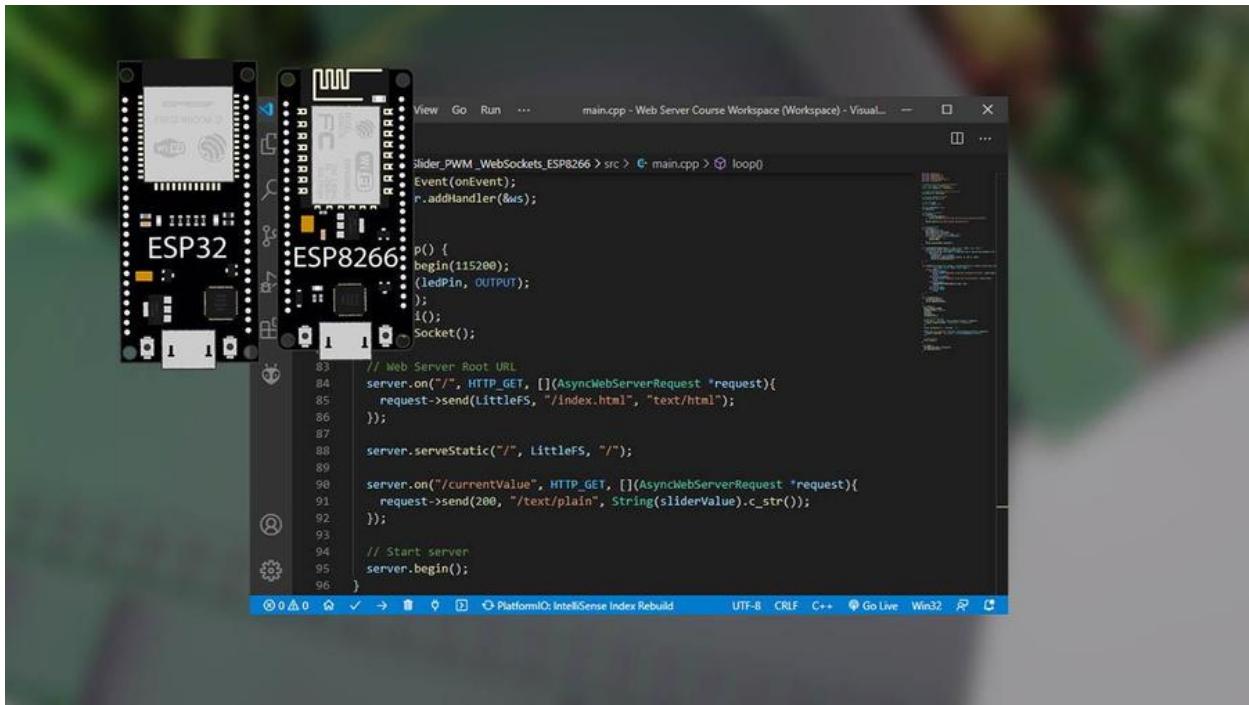
As we've mentioned previously, we'll use JavaScript mainly to make requests to the ESP32 or ESP8266 boards. We'll address this subject later when we start building the web servers. This section was just an introduction to JavaScript to understand what it is used for and how to get started. We've shown examples that are simple yet useful, like changing the style of an HTML element or manipulating its content.

MODULE 3

ESP32 and ESP8266 Web Servers

Learn how to build web server projects with the ESP32 and ESP8266 boards to control outputs and monitor sensors remotely. You'll create different web pages to control outputs and discover different ways to display sensor readings on the web page: text, tables, and charts. You'll also learn different client-server communication protocols: HTTP requests, WebSocket, and Server-Sent Events (SSE).

Introducing Web Servers



In this section you'll learn what a web server is and how an ESP32 or ESP8266 web server works.

Introducing Web Servers

In simple terms, a web server is a “computer” that provides web pages. It stores the website’s files, including all HTML documents and related assets like images, CSS style sheets, fonts, and/or other files. It also brings those files to the user’s web browser device when the user makes a request to the server.

When you access a web page in your browser, you’re actually sending a request via Hypertext Transfer Protocol (HTTP) to a server. This is a process for requesting and returning information on the internet. The server sends back the web page you requested—also through HTTP.

To better understand how all of this works with your ESP32 and ESP8266 boards, let's take a look at some terms that you've probably heard before, but you may not know exactly what they mean.

Request-response

Request-response is a message exchange pattern, in which a requestor (your browser) sends a request message to a replier system (the ESP32 or ESP8266) that receives and processes the request, and returns a message in response.

In most of our projects, the response message will be a web page that displays the latest sensor readings or that provides an interface to control outputs.

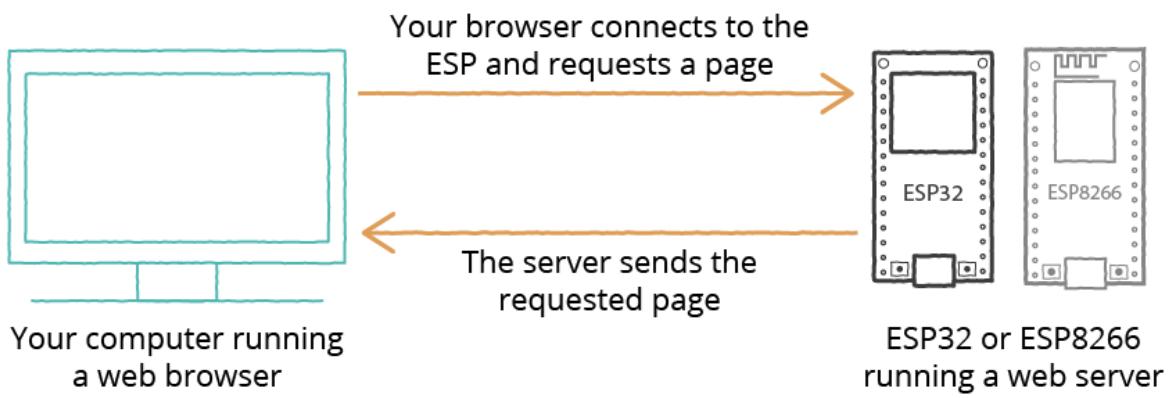


This is a simple, yet powerful messaging pattern, especially in client-server architectures.

Client-Server

When you type an URL in your browser, in the background, you (the client) send a request via Hypertext Transfer Protocol (HTTP) to a server. When the server receives the request, it sends a response also through HTTP, and you will see the web page requested in your browser. Clients and servers communicate over a computer network.

In simple terms, clients make requests to servers. Servers handle the clients' requests.



In our projects, the ESP32 or the ESP8266 is the server, and you (your browser) are the client. Our projects only have one server (the ESP32 or ESP8266 boards), but can have multiple clients: multiple web browsers on the same network but on different devices like computers, smartphones or tablets, or multiple web browser tabs opened on the same device.

IP Address

An IP address is a numerical label assigned to each device connected to a computer network.

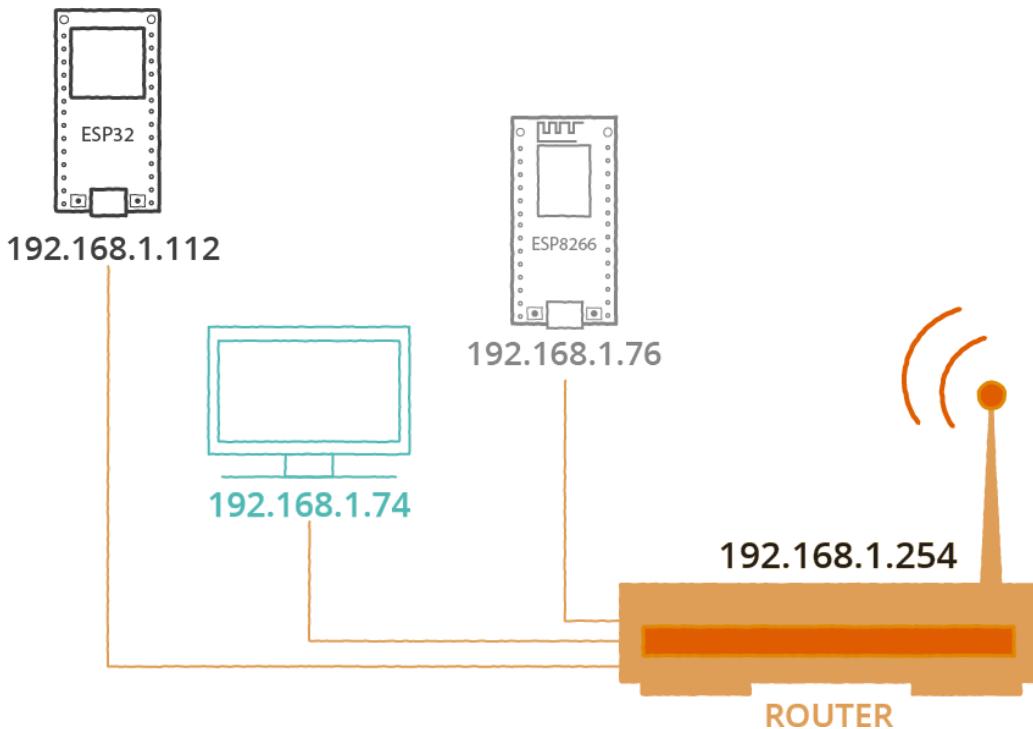
It is a series of four values separated by periods with each value ranging from 0 to 255. Here's an example:

192.168.1.75

At home, your devices are connected to a private network through your router (local network). All devices connected to your router are part of your local network. Inside this network, each device has its own IP address.

Devices that are connected to the same router can access each other via the IP address. Devices outside your local network can't access your local devices using their local IP address.

When you connect your ESP32 or your ESP8266 to your router, they become part of your local network. So, your ESP32 and ESP8266 boards are assigned an IP address.



In your local network, the IP address of the ESP board (and other devices) is assigned by the router using DHCP (Dynamic Host Configuration Protocol). You don't need to worry about what DHCP is, you just need to know that it assigns an IP address and other network configuration parameters to each device on the network.

The router keeps track of every device on the network and maps an IP address to each device every time it joins the network. Two devices on the same network can't have the same IP address.

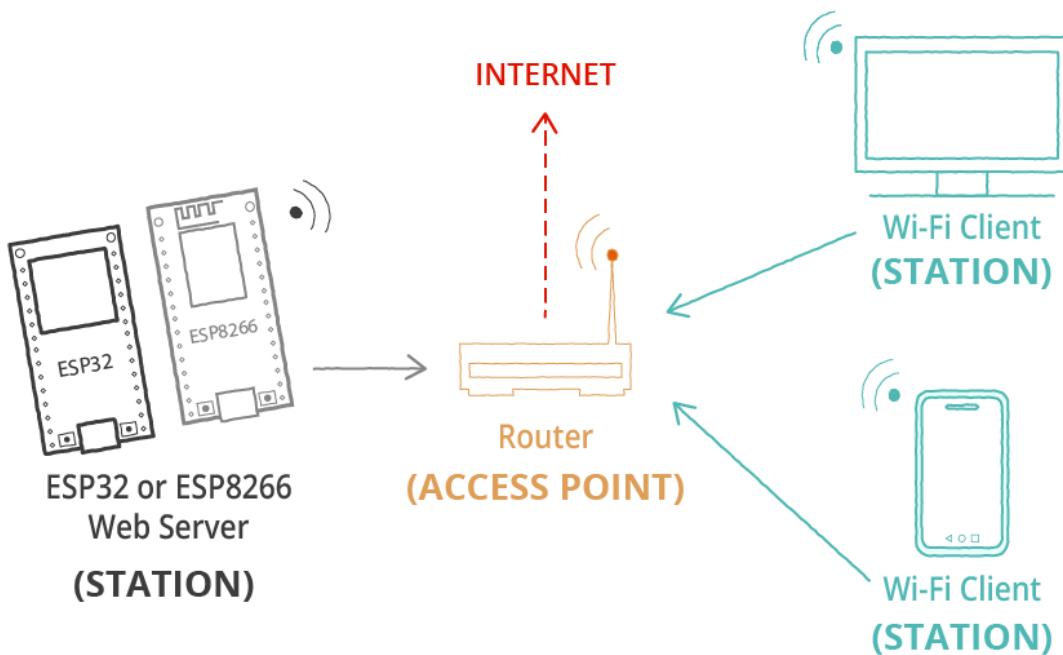
Again, when the ESP32 or ESP8266 is connected to your router, the IP address assigned is a local address. This means you can only access it using other devices that are also connected to the same network. In the previous image, you can access the ESP32 or the ESP8266 boards using the computer (teal color) that is also connected to the same network.

Wi-Fi Station and Wi-Fi Access Point

The ESP32/ESP8266 board can act as Wi-Fi Station, Access Point or both.

Wi-Fi Station

When the ESP32 or ESP8266 is set as a Wi-Fi station, it can connect to other networks (like connecting to your router—local network). In this scenario, the router assigns a unique IP address to your ESP board. You can communicate with the ESP using other devices (stations) that are also connected to the same network by referring to the ESP's local IP address.

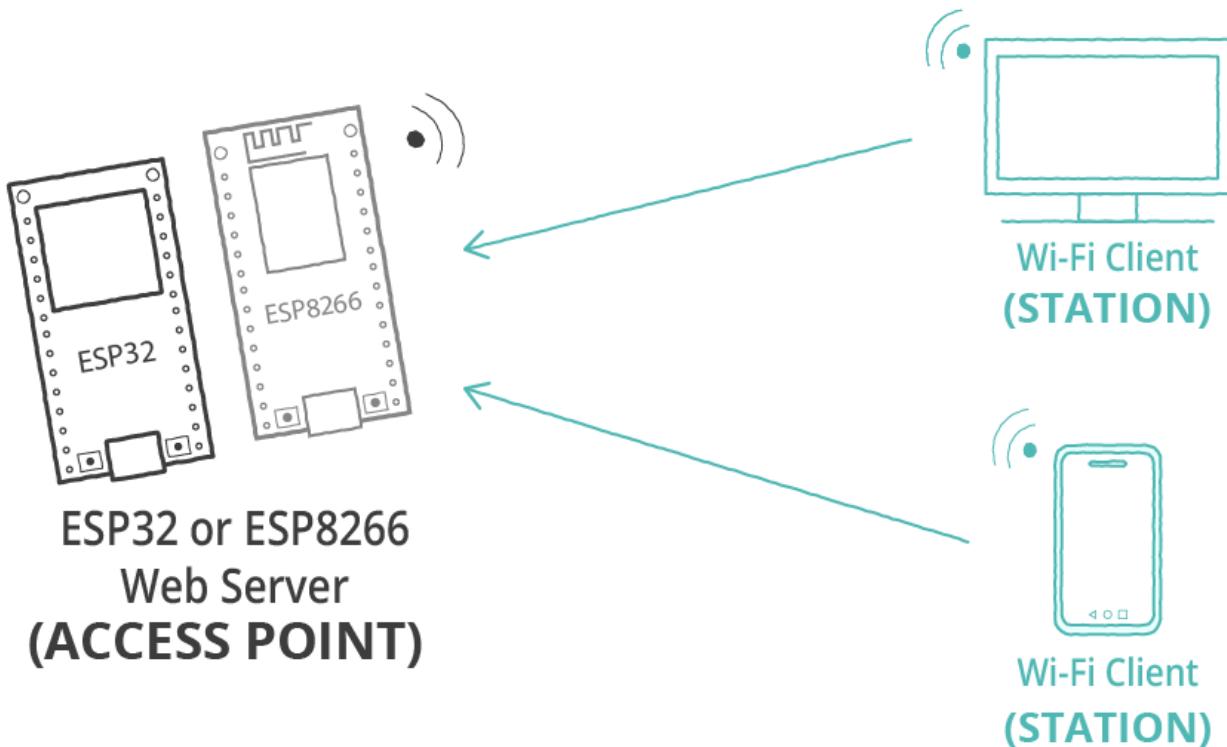


Since the router is also connected to the internet, we can request information from the internet using our ESP boards like data from APIs (weather data, for example), publish data to online platforms, use icons and images from the internet in our web server pages or include JavaScript libraries.

However, in some cases, we may not have a router nearby to connect the ESP. In this scenario, you must set your ESP board as an access point.

Access Point

When your ESP is set up as an Access Point, other devices (such as your smartphone, tablet, or computer) can connect to it without the need for a router; the ESP controls its own Wi-Fi network.



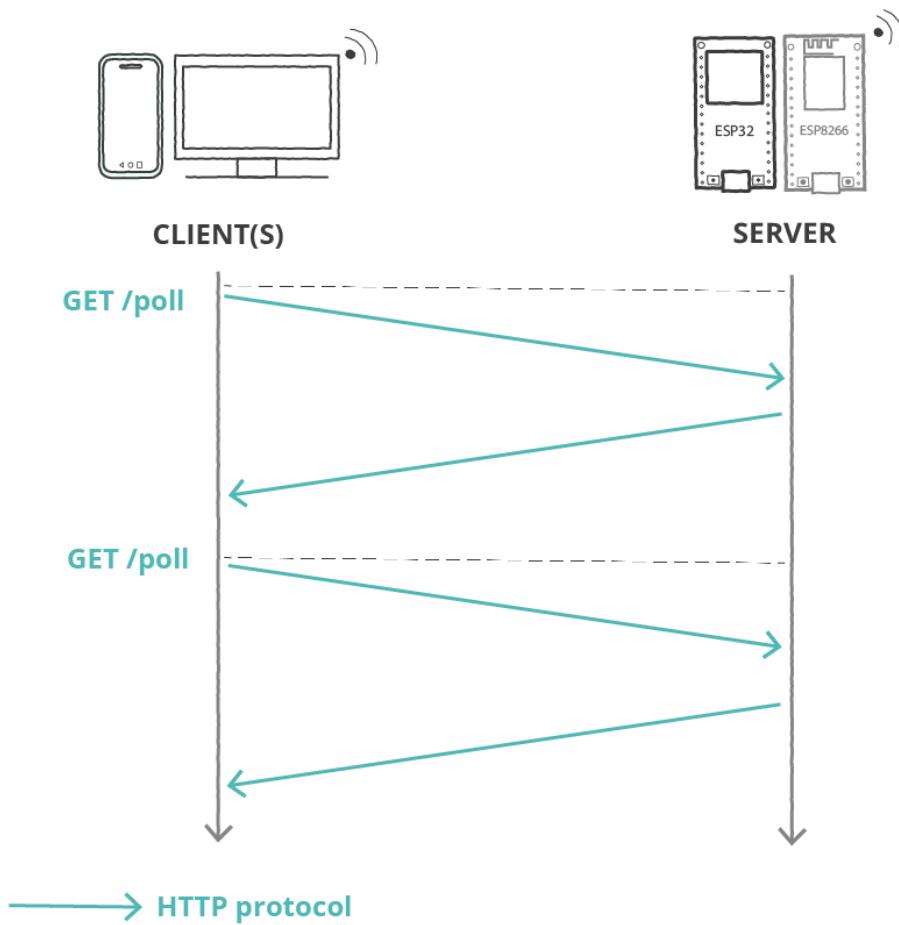
Unlike a router, an ESP Access Point doesn't connect further to a wired network or the Internet, so you can't access external libraries, publish sensor readings to the cloud, or use services like mail. In our examples, we'll set the boards as stations. This is more versatile, in our opinion, because it allows you to insert icons, fonts, and other assets from the web in your web page. However, you can easily modify our examples and set the boards as access points instead.

Client-Server Communication

There are several ways to communicate between the client and the server. We'll take a look at HTTP Polling, Server-Sent Events (SSE), and WebSocket.

HTTP Polling

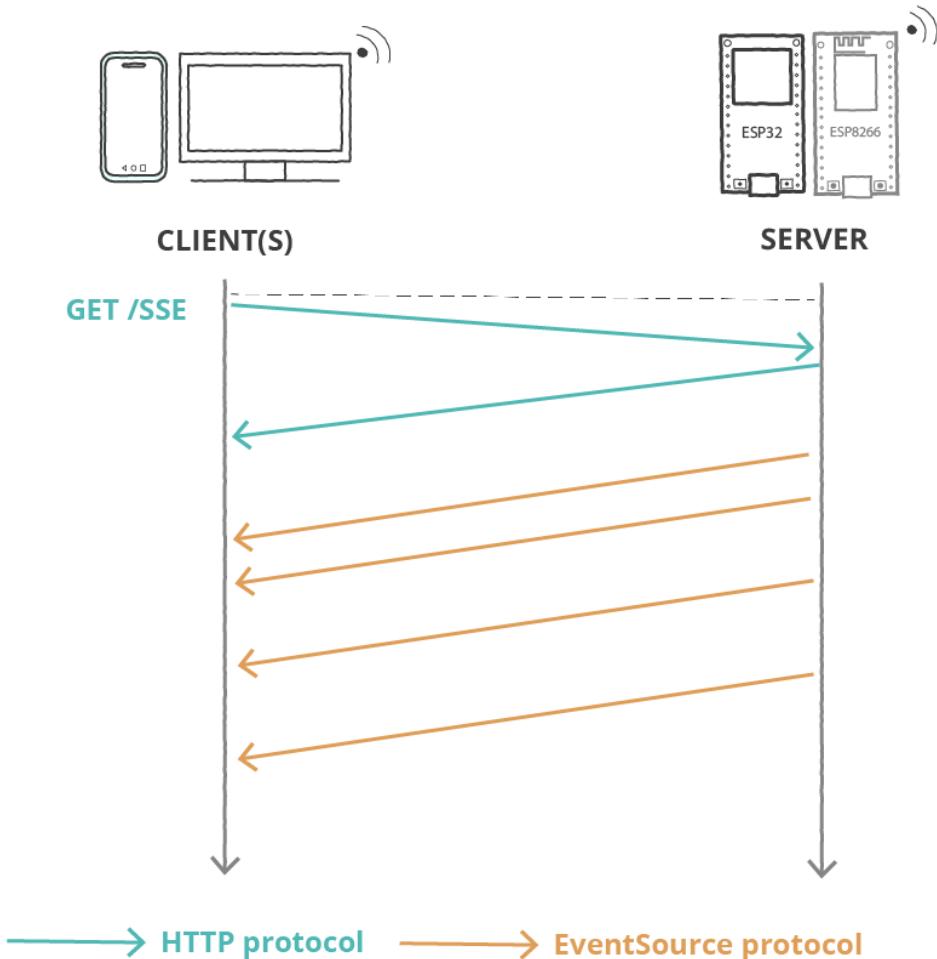
In HTTP polling, the client polls the server requesting new information. When the server receives a request, it sends a response back to the client. The server will only send information when requested by the client.



For example, you in your browser send a request to the ESP32, and it responds with whatever you want, for example, a web page with the latest sensor readings. Using this method, to keep your web page updated, you need to make new requests constantly.

Server-Sent Events

Server-Sent Events (SSE) allows the client to receive automatic updates from a server via an HTTP connection.

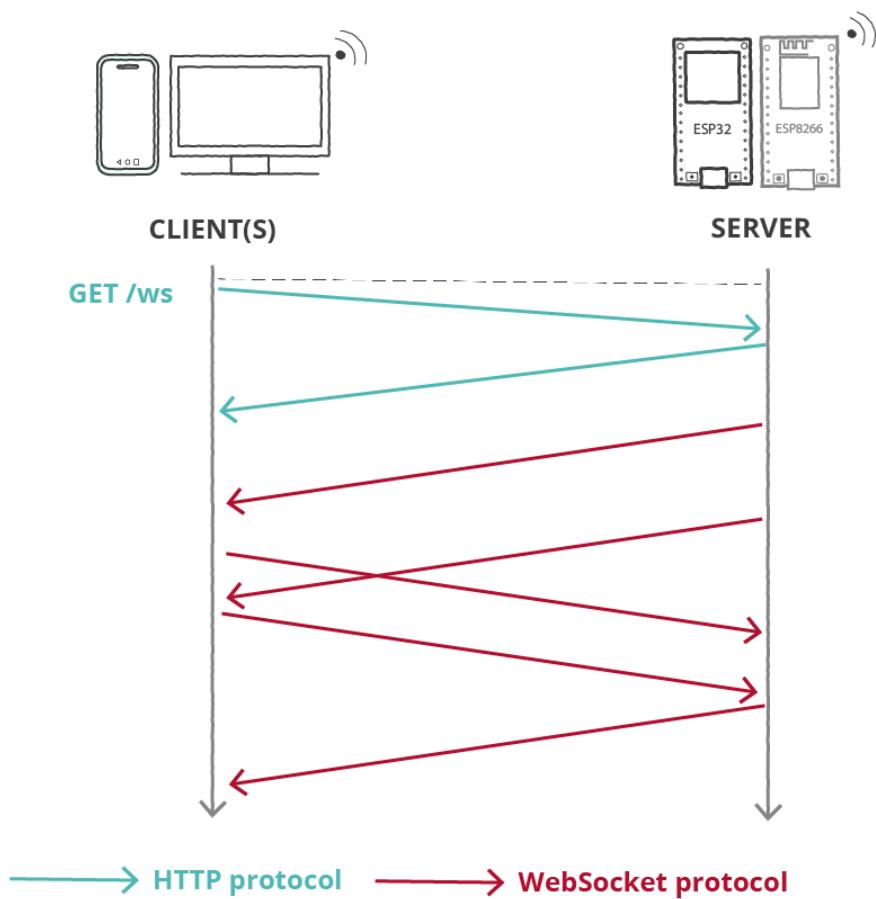


The client initiates the SSE connection, and the server uses the event source protocol to send updates to the client. The client will receive updates from the server, but it can't send any data to the server after the initial handshake.

This is useful to send updated sensor readings to the browser. Whenever a new reading is available, or a change happens in the state of a GPIO, the ESP sends it to the client, and the web page can be updated automatically without the need for further requests.

WebSocket

A WebSocket is a persistent connection between a client and server that allows bidirectional communication between both parties using a TCP connection (Transmission Control Protocol is simply the name of part of the standard way of ensuring computer systems can interact correctly and reliably). This means you can send data from the client to the server and from the server to the client at any given time.



The client establishes a WebSocket connection with the server through a process known as a *WebSocket handshake*. The handshake starts with an HTTP request/response, allowing servers to handle HTTP connections and WebSocket connections on the same port. Once the connection is established, the client and the server can send WebSocket data in full duplex mode.

ESPAsyncWebServer Libraries

Throughout this eBook, we'll use the [ESPAsyncWebServer](#) Library to build web servers with the ESP32 and ESP8266 boards. To use this library, you also need the [AsyncTCP](#) library for the ESP32 and the [ESPAsyncTCP](#) library for the ESP8266.

- [ESPAsyncWebServer](#) (ESP32 and ESP8266)
- [AsyncTCP](#) (ESP32)
- [ESPAsyncTCP](#) (ESP8266)

We'll use these libraries to build the web servers because we've been working with them for some time, and they work pretty well. Additionally, these libraries include many functionalities to handle web servers easily.

What you have learned about HTML, CSS and JavaScript can also be used with other libraries. You just need to use the proper methods to handle requests and serve HTML pages using your ESP32 or ESP8266 boards.

Async Web Server Advantages

Building an asynchronous web server has several advantages, as mentioned in the library GitHub page, such as:

- “Handle more than one connection at the same time”;
- “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”;
- “Speed is OMG (it’s fast)”;
- “Easily extendible to handle any type of content”;
- “Async EventSource (Server-Sent Events) plugin to send events to the browser”;

- “Async WebSocket plugin offering different locations without extra servers or ports”;
- “Simple template processing engine to handle templates”;
- And much more.

Important Things to Remember

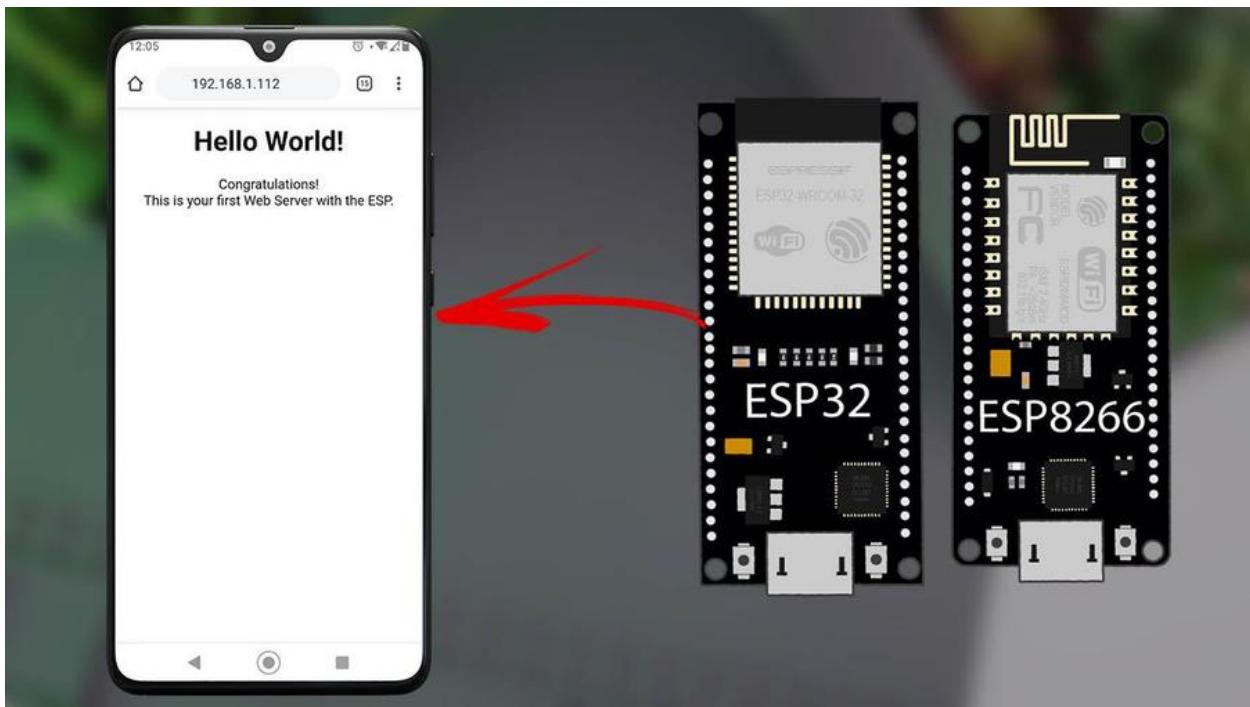
There are some essential things you need to remember when building a web server using these libraries. Here are the bullets mentioned on the library GitHub page:

- “This is a fully asynchronous server and as such **does not run on the loop thread**”;
- “You **cannot use yield or delay** or any function that uses them inside the callbacks”;
- “The server is smart enough to know when to close the connection and free resources”;
- “You cannot send more than one response to a single request”.

We also recommend that you take a look at the [library GitHub page](#) to have a better idea on how everything works.

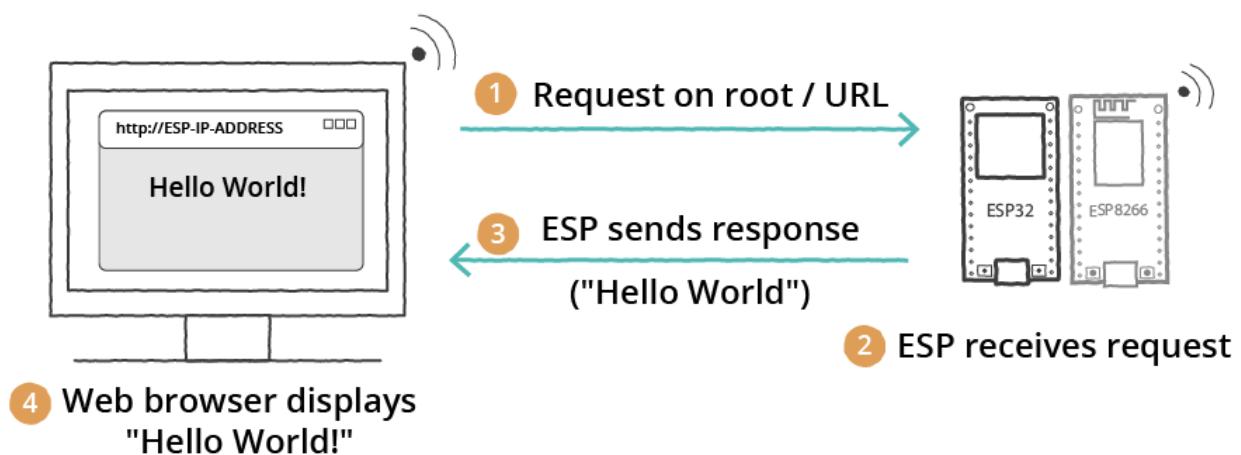
Don't worry if you don't understand some of the concepts, once you start building the web server projects, you'll see how this works with practical applications.

1.1 - Hello World Web Server



In this section, you'll build your first web server with the ESP32 or ESP8266. This is a simple web server that displays a "Hello World" message. This project aims to get you familiar with the workflow to upload code and files to your board, import libraries, and all the steps you need to follow to get your project working.

Project Overview

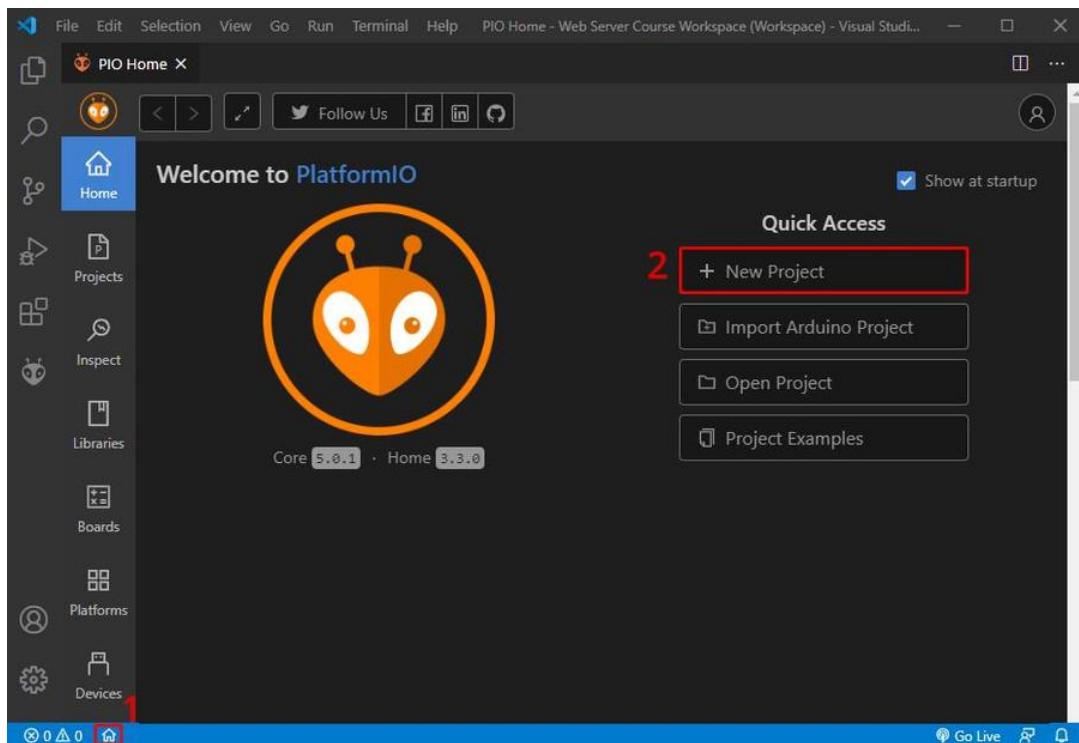


We'll set the ESP32/ESP8266 as a Wi-Fi station that is connected to your router. You can access the web server by typing the ESP's IP address in a browser on your local network. When you access the ESP IP address, you'll see the "Hello World!" web page.

Create a New Project (VS Code)

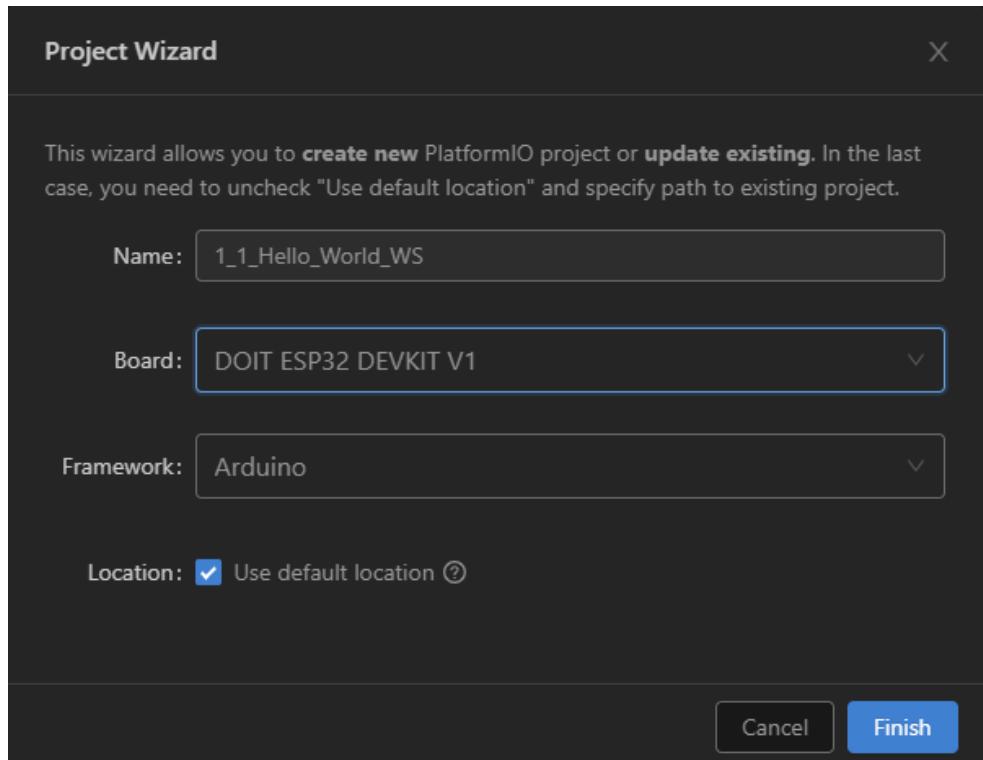
Open VS Code Editor. Make sure you've followed all the steps to install the PlatformIO extension in VS Code. If you didn't, go back to Module 1 and follow all the steps.

In VS Code, open PlatformIO Home. You can click on the Home icon at the bottom toolbar (1). The PlatformIO Home will open. Click on **+ New Project** (2).

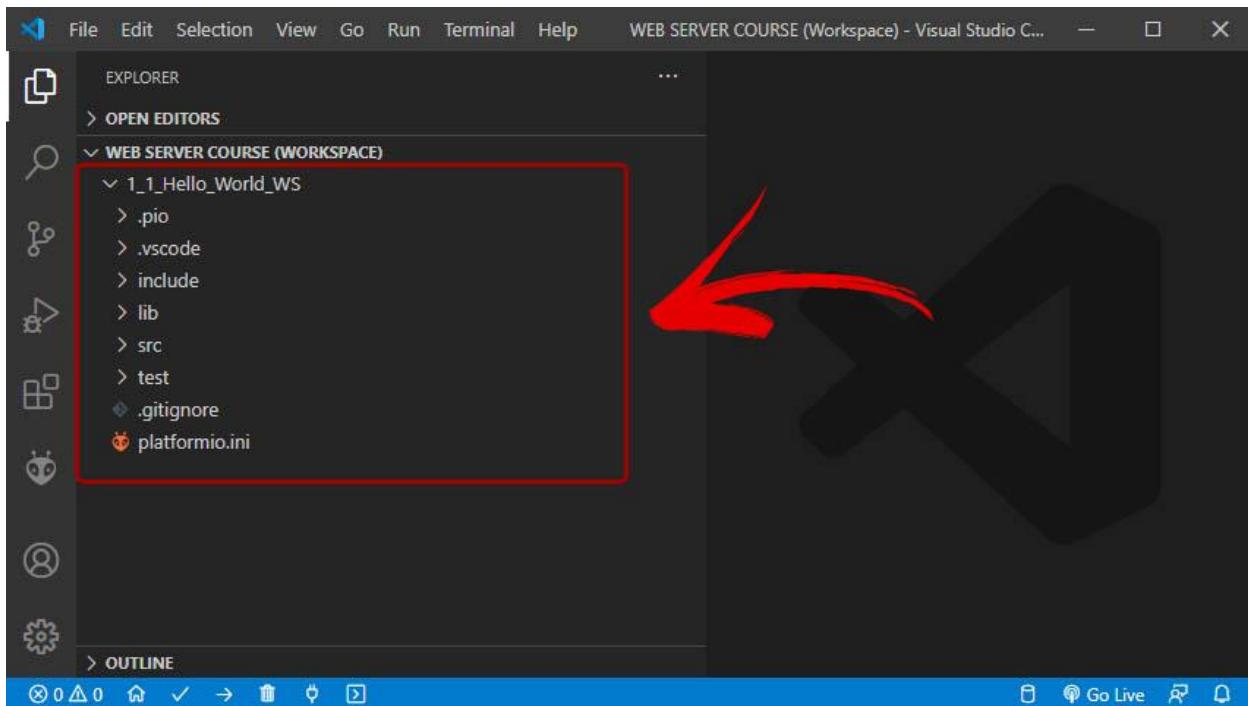


Give your project a name, for example, `1_1_Hello_World_WS`. Select the board you're using. The framework should be Arduino.

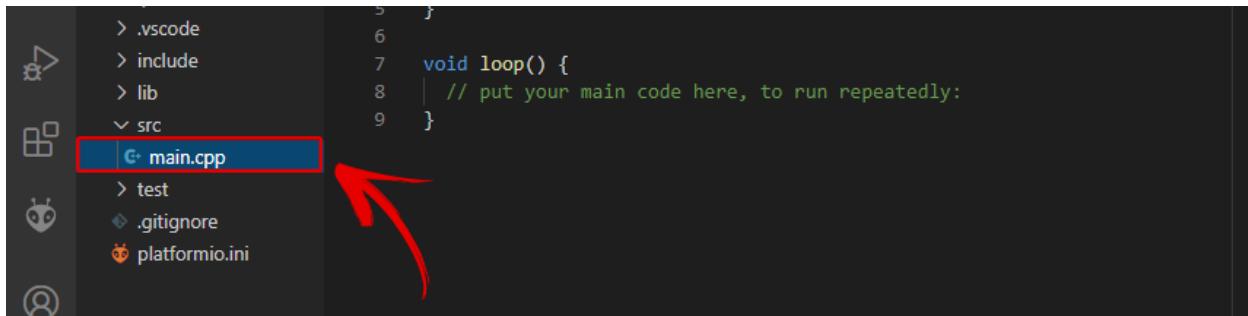
You can save the project in the default location or choose any other folder on your computer. After that, click **Finish**.



VS Code and PlatformIO have a folder structure that is different from the standard *.ino* project. Click the **Explorer** tab to see all the files created under your project folder. It may seem like a lot of files to work with. But, don't worry. Usually, you'll just need to deal with one or two of those files.



The `src` folder is your working folder. Under the `src` folder, there's a `main.cpp` file. That's where you write your code. Click on that file, and the structure of an Arduino program should open with the `setup()` and `loop()` functions.



A screenshot of the PlatformIO IDE interface. On the left is a tree view of the project files: .vscode, include, lib, src (which contains main.cpp), test, .gitignore, and platformio.ini. The `main.cpp` file is selected and highlighted with a blue border. A red arrow points from the text above to this selection. On the right is a code editor window showing the contents of `main.cpp`:

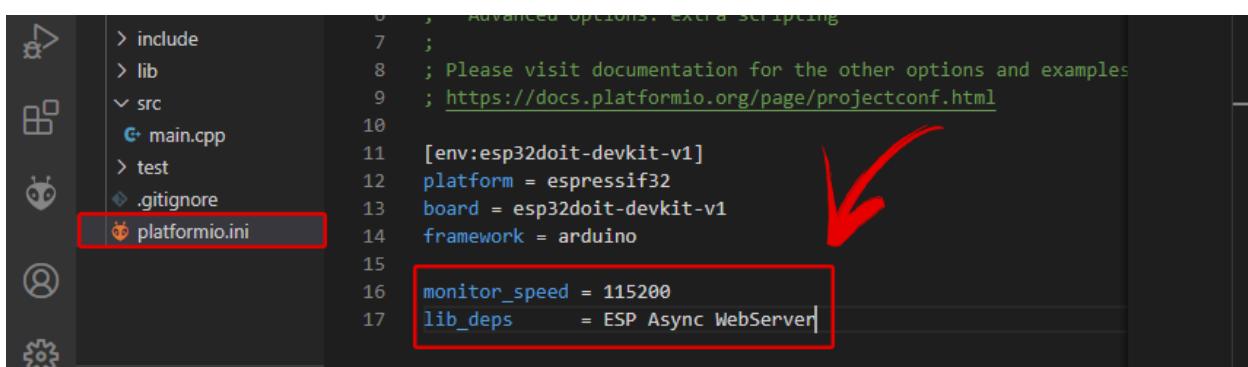
```
5    }
6
7    void loop() {
8        // put your main code here, to run repeatedly:
9    }
```

Important: in PlatformIO IDE, all your Arduino sketches should start with `#include <Arduino.h>`.

Configurations: `platformio.ini` file

To build the web server we'll use the `ESPAsyncWebServer` library. To add this library to your project, open your `platformio.ini` file and add the `lib_deps` directive with the library name as shown below. Additionally, change the `monitor_speed` to 115200.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps      = ESP Async WebServer
```



A screenshot of the PlatformIO IDE interface. On the left is a tree view of the project files: include, lib, src (which contains main.cpp), test, .gitignore, and platformio.ini. The `platformio.ini` file is selected and highlighted with a blue border. A red arrow points from the text above to this selection. On the right is a code editor window showing the contents of `platformio.ini`:

```
6      ; Advanced options, extra scripting
7      ;
8      ; Please visit documentation for the other options and examples
9      ; https://docs.platformio.org/page/projectconf.html
10
11 [env:esp32doit-devkit-v1]
12 platform = espressif32
13 board = esp32doit-devkit-v1
14 framework = arduino
15
16 monitor_speed = 115200
17 lib_deps      = ESP Async WebServer
```

A large red checkmark is placed over the `platformio.ini` file in the tree view, indicating it is the correct file to edit.

Press **Ctrl+S** to save the changes. If you see a little white circle next to the *platformio.ini* tab, it means that the latest changes haven't been saved yet. After saving, the little white circle disappears.

PlatformIO will take care of downloading the source code of the library and integrating it into the project for you and including any dependencies like the AsyncTCP or ESPAsyncTCP libraries.

If you're using an ESP8266, the *platformio.ini* file should be similar but with a different **platform** and **board**.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps    = ESP Async WebServer
```

Web Server Code: main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```

<link rel="icon" href="data:,">
<style>
  html {
    font-family: Arial;
    text-align: center;
  }
  body {
    max-width: 400px;
    margin: 0px auto;
  }
</style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
)rawliteral";

void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

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

  server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(200, "text/html", index_html);
  });

  // Start server
  server.begin();
}

void loop() {
  // put your main code here, to run repeatedly:
}

```

Insert your network credentials in the `ssid` and `password` variables.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Web Server Code: main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
  <style>
    html {
      font-family: Arial;
      text-align: center;
    }
    body {
      max-width: 400px;
      margin:0px auto;
    }
  </style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
)rawliteral";
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
```

```

    }
    Serial.println(WiFi.localIP());
}
void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    initWiFi();

    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(200, "text/html", index_html);
    });
    // Start server
    server.begin();
}
void loop() {
}

```

Insert your network credentials in the `ssid` and `password` variables.

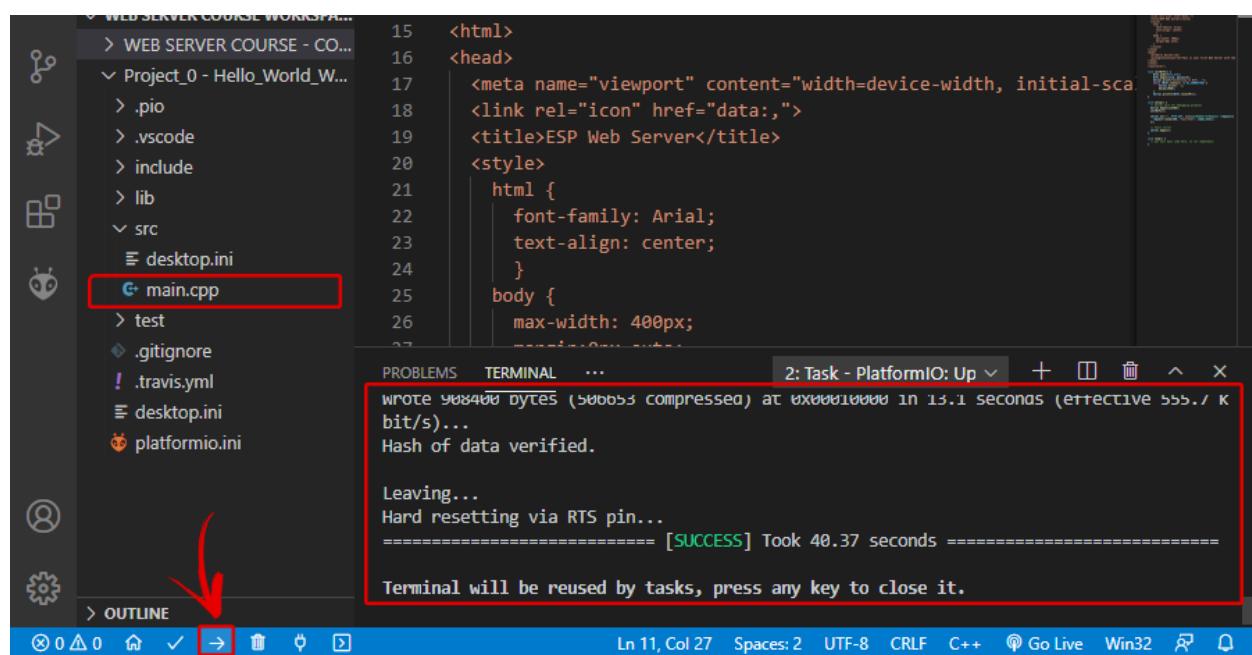
```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Uploading Code

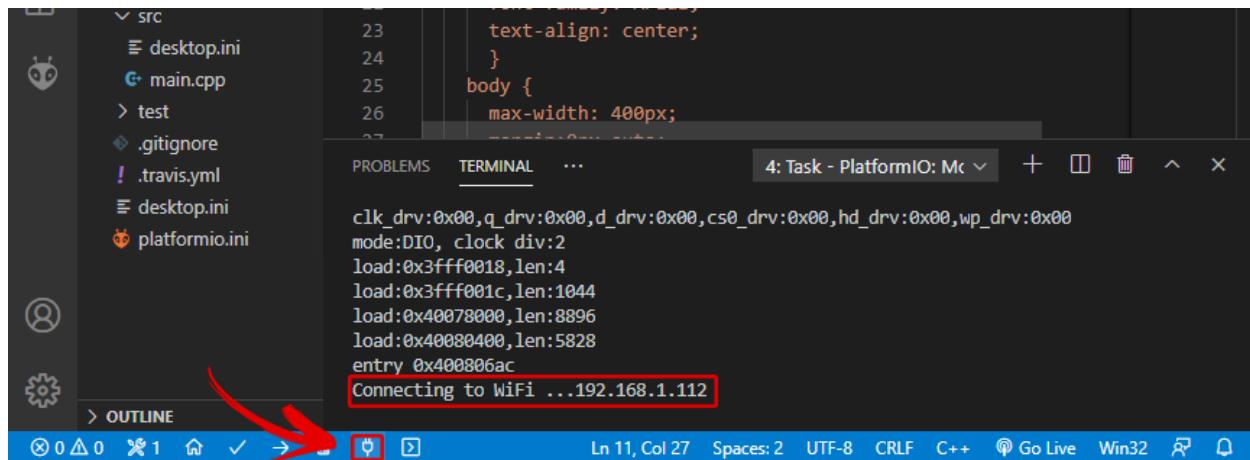
With your ESP32 or ESP8266 connected to your computer, click on the **Upload** icon. The program will compile and upload to your board. You can see the progress in the Terminal window.



If the Terminal window is not visible, you can go to **Terminal > New Terminal** to open a Terminal window.

You should get a message like the one shown in the previous image.

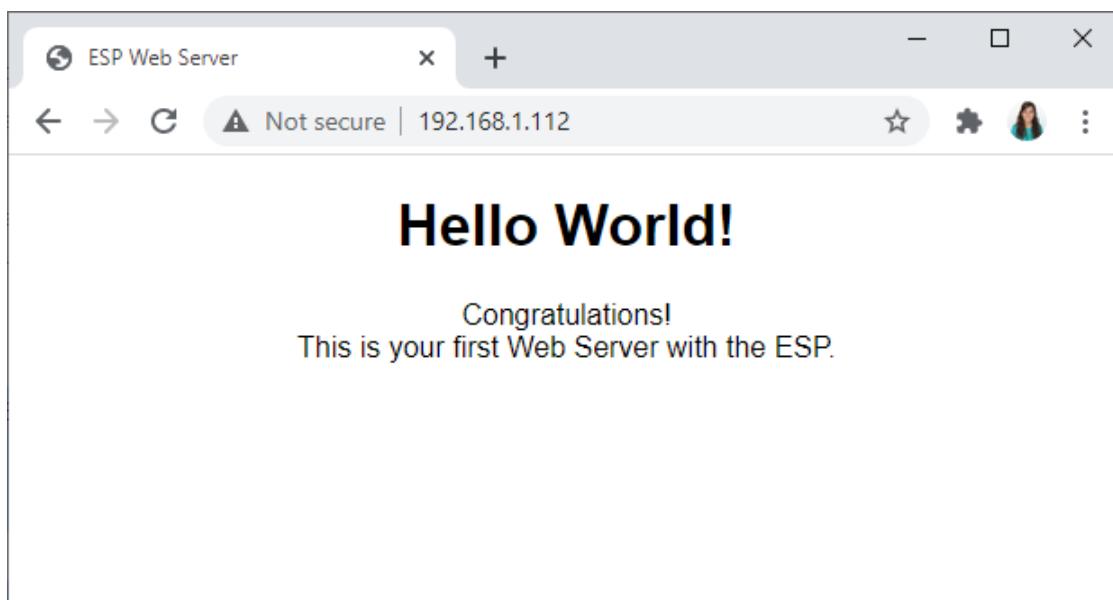
Now, click on the **Serial Monitor** icon to open the Serial Monitor. It should print the ESP32 or ESP8266 IP address.



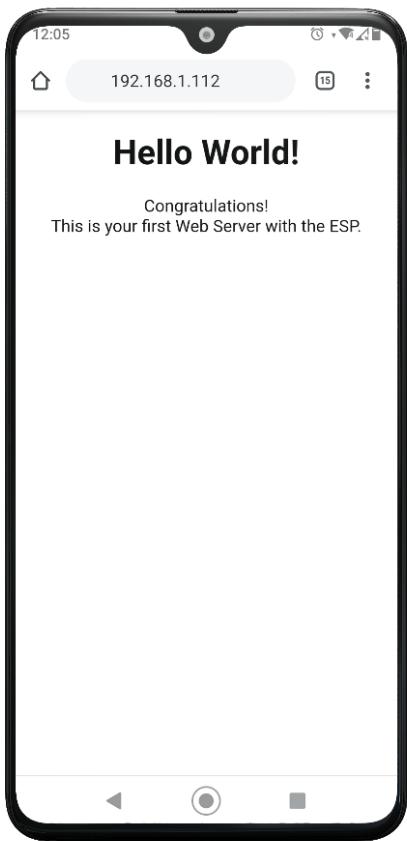
```
text-align: center;
}
body {
    max-width: 400px;
}
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5828
entry 0x400806ac
Connecting to WiFi ...192.168.1.112
```

If the IP address is not printed after a few seconds, press your on-board RST/EN button. Now, open a browser on your local network (it can be on your computer, smartphone, or tablet) and type the ESP IP address on the search bar.

This is how the web page served by the ESP32 or ESP8266 looks like in your browser.



And this is what it looks like on your smartphone.



Congratulations! You've built your first web server with the ESP32 or ESP8266 board.

Here's a summary of what is happening:

1. The ESP32 or ESP8266 is connected to your local network. Your router assigns an IP address to your board.
2. When you type the ESP IP address on the browser, you are making a request on the root (/) URL of your board.
3. The board receives that request and sends the HTML to build the "Hello World" web page.

How the Code Works

Continue reading to learn how the code to build the web server works.

Libraries

First, include the necessary libraries.

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

The `WiFi` library is included to use the Wi-Fi capabilities of the ESP32; the `AsyncTCP` and `ESPAsyncWebServer` are included to support communication to and from the web server.

If you're using an ESP8266, you should load the following libraries instead.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
```

Network Credentials

As mentioned previously, you should include your network credentials in the `ssid` and `password` variables.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

AsyncWebServer Object

Create an `AsyncWebServer` object on port 80 (the default port for HTTP servers).

```
AsyncWebServer server(80);
```

Creating the Web Page

The `index_html` variable contains all the text to build the HTML page. The `PROGMEM` keyword means that the variable will be saved in flash. The `R"rawliteral` allows us to spread the text on different lines. This is useful to improve the readability of our

`index_html` variable. The `R` means “Treat everything between these delimiters as a raw string”. That is, everything between `rawliteral`(at the beginning of the string) and `rawliteral` (at the end of the string). You can use any other word instead of `rawliteral` as long as that word is not used inside your string.

This is the HTML text that is saved in the `index_html` variable:

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,">
  <style>
    html {
      font-family: Arial;
      text-align: center;
    }
    body {
      max-width: 400px;
      margin: 0px auto;
    }
  </style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
```

This HTML text is responsible for building the web page you saw previously. If you've followed previous Units you are already familiar with how HTML works, but let's take a more in-depth look at it anyway.

The first line of any HTML document is always the following:

```
<!DOCTYPE html>
```

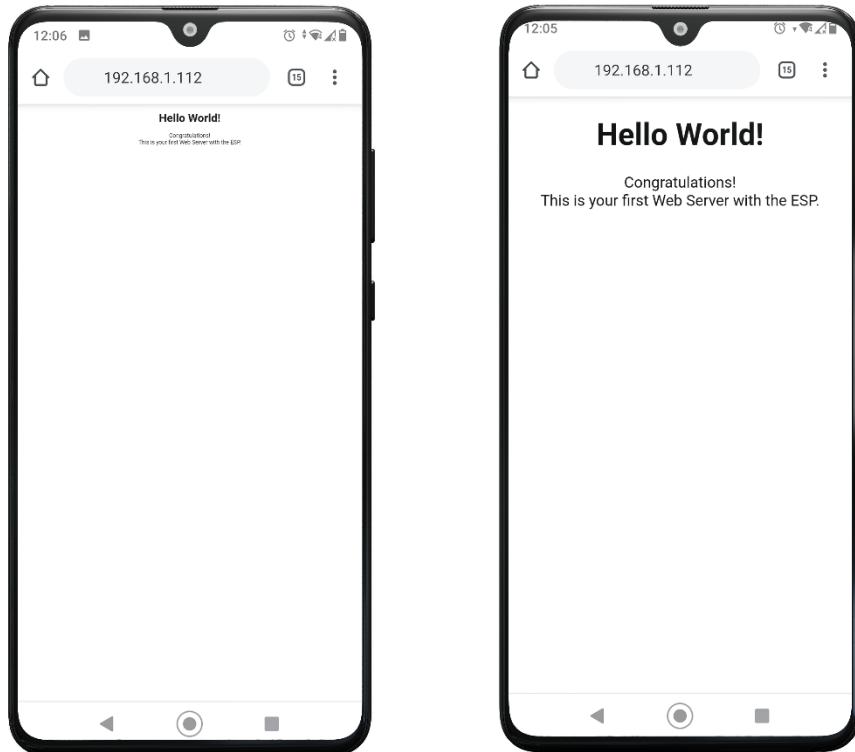
`!DOCTYPE` isn't an HTML-specific tag. It's simply an instruction that tells your web browser that it's reading an HTML document and which version of HTML you're using (HTML5). The HTML that defines the structure of your page goes between the `<html>` and `</html>` tags.

The `<head>` and `</head>` tags mark the start and end of the head. The head is where you insert data about the HTML document that is not directly visible to the end user, but adds functionality to the web page – this is called *metadata*.

The following meta tag makes your web page responsive. A responsive web design will automatically adjust for different screen sizes and viewports.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

For example, at the left you can see a web page without this meta tag and at the right with the meta tag.



We use the following meta tag because we won't serve a favicon for our web page in this particular example.

```
<link rel="icon" href="data:,>
```

The following line gives a title to the web page. In this case, it is set to `ESP Web Server`. You can change it if you want. The title is exactly what it sounds like: the title of your document, which shows up in your web browser's title bar.

```
<title>ESP Web Server</title>
```

In the head of our HTML document, we've also included some CSS to style the web page, between the `<style></style>` tags.

```
<style>
  html {
    font-family: Arial;
    text-align: center;
  }
  body {
    max-width: 400px;
    margin: 0px auto;
  }
</style>
```

We're setting our HTML document to use Arial font and align the text to the center.

The maximum width of the body is 400px. This means that even though you have a larger window, the body's content will only occupy 400px. Setting the margin to `0px auto`, sets the bottom and upper margin to `0` and `auto` to the left and right margins. Setting the left and right margins to `auto` will center the body - the browser automatically distributes the right amount of margin to either side.

The `<body>` and `</body>` tags mark the start and end of the body. Everything that goes inside those tags is the visible page content. In our case, we have a heading 1 with the `Hello World!` text.

```
<h1>Hello World!</h1>
```

There's also a paragraph with the following text. The `
` tag breaks the paragraph into another line.

```
<p>Congratulations!<br>This is your first Web Server with the ESP.</p>
```

Initialize Wi-Fi

In our code, we've created a function that initializes Wi-Fi called `initWiFi()`.

```
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}
```

Let's take a look at that function. First, set the ESP32 or ESP8266 as a station.

```
WiFi.mode(WIFI_STA);
```

Then, connect to Wi-Fi with the `ssid` and `password` defined earlier.

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

Finally, print the ESP IP address in the Serial Monitor.

```
Serial.println(WiFi.localIP());
```

setup()

In the `setup()`, initialize the serial monitor at a baud rate of 115200.

```
Serial.begin(115200);
```

Call the `initWiFi()` function to initialize Wi-Fi.

```
initWiFi();
```

Handle requests

The following lines handle what happens when the ESP receives a request on the root (/) URL (ESP IP address).

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){  
    request->send(200, "text/html", index_html);  
});
```

It sends the HTML text saved in the `index_html` variable to build the web page when it receives that request.

```
request->send(200, "text/html", index_html);
```

The `send()` method can receive different arguments depending on what we want to send. In this case, we're passing three parameters as described below:

```
void send(int code, const String& contentType=String(), const String& content=String());
```

The first argument is the response code. The HTTP 200 OK success status response code indicates that the request has succeeded. The second argument is the content type. In our case, we're sending HTML text ("text/html"). And finally, the content we want to send (`index_html`).

These previous lines of code run asynchronously. This means that these will only run when the ESP receives a request on that specific URL. So, you can add other tasks to your code, and the board will still be able to handle the web server request at any given time.

Finally, start the web server with the `begin()` method.

```
server.begin();
```

This is an asynchronous web server, so it is unnecessary to add anything to the `loop()`.

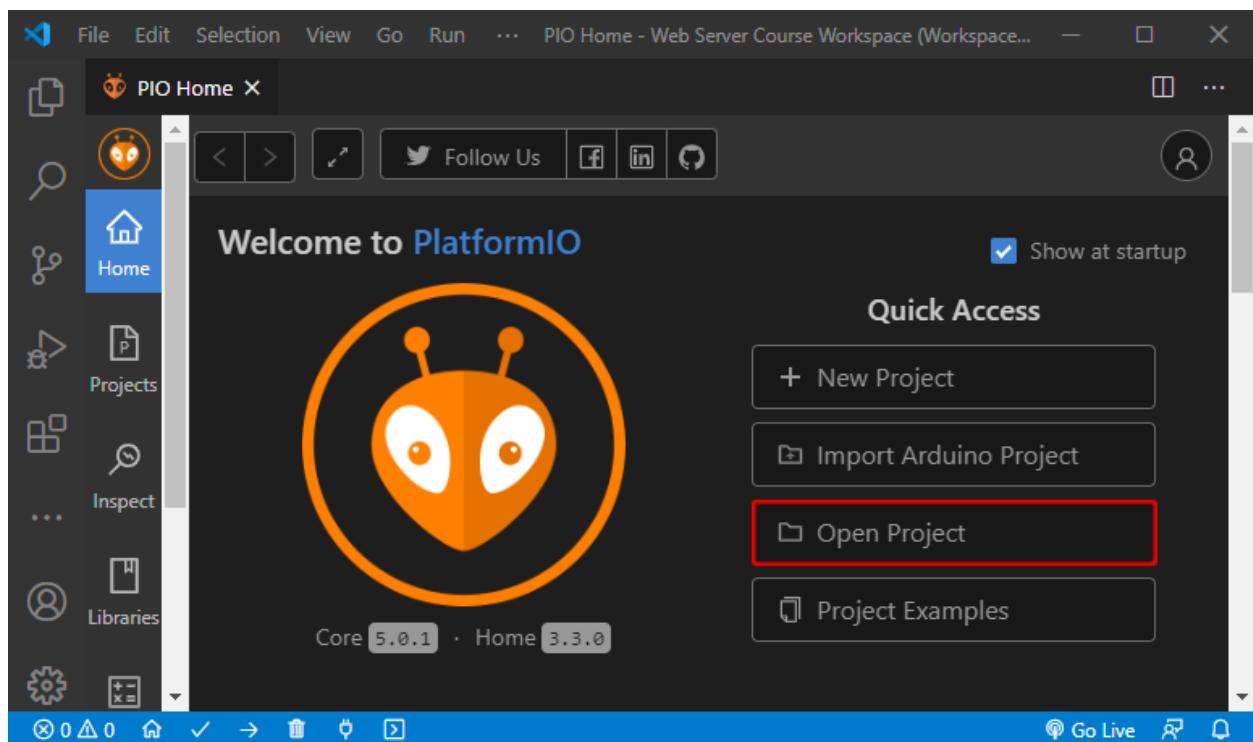
Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Open a Project Folder

To open a project folder on PlatformIO, open VS Code, go to PlatformIO Home and click on **Open Project**. Navigate through the files and select your project folder.



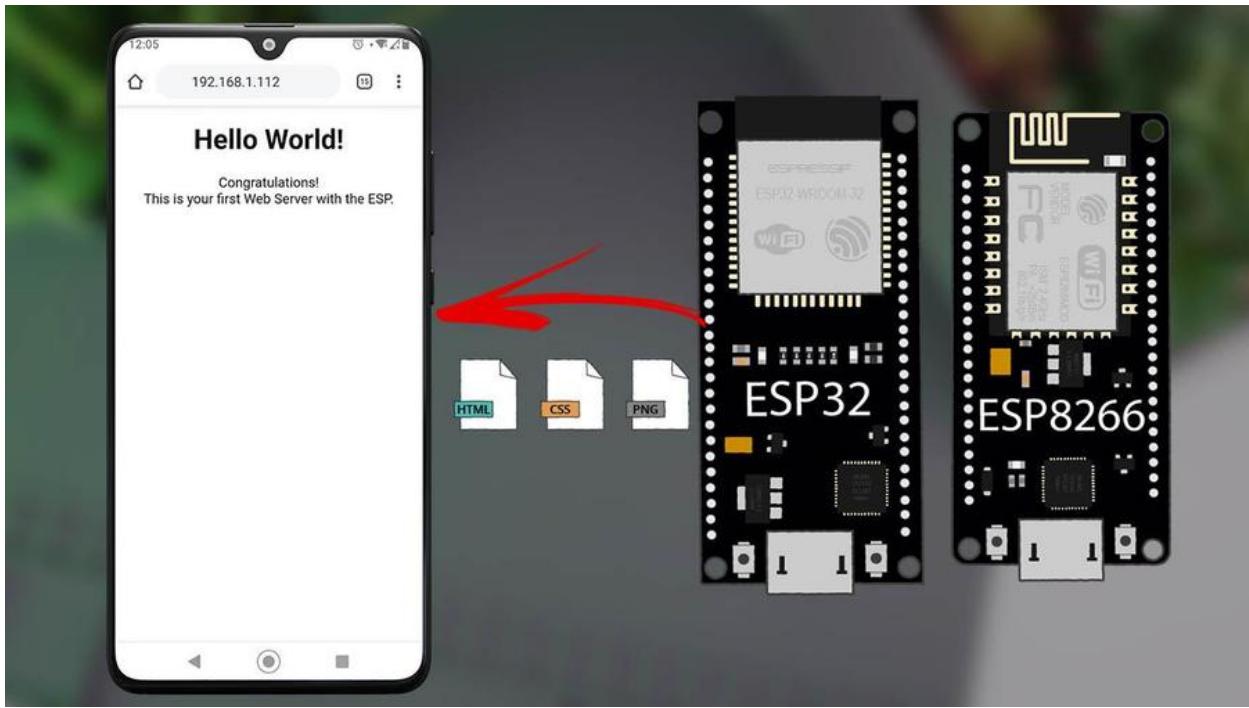
PlatformIO will open all the files within the project folder.

Wrapping Up

In this project, you've learned how to build a simple web server with the ESP32 and ESP8266 boards. When you make a request on the ESP root / URL, you send the HTML text to build the "Hello World!" web page.

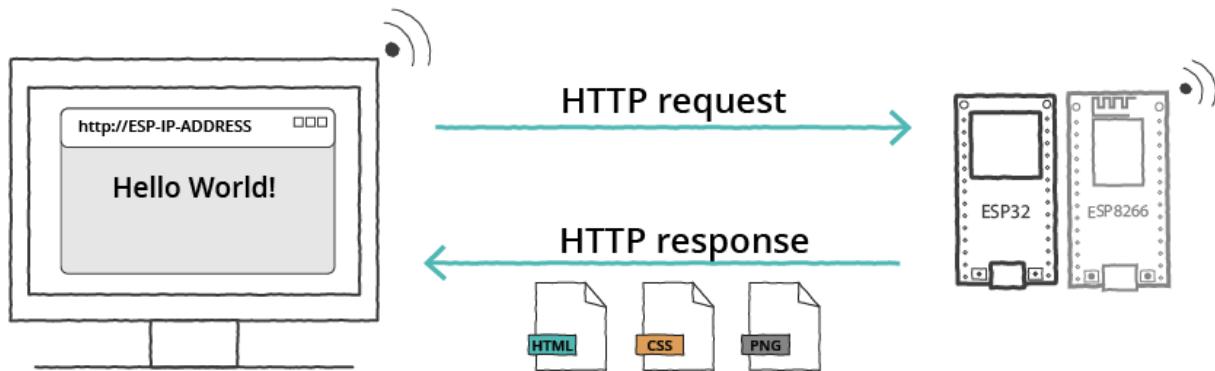
In this particular example, we've added all the HTML and CSS needed to build the web page on the *main.cpp* file. Most of the times, it is more practical to have the HTML and CSS on separated files that you upload to the board filesystem and reference in the code. That's what you're going to learn in the next Unit.

1.2 - Hello World Web Server (Serve Files from Filesystem)



Previously, we've created the HTML text to build the web page in the Arduino sketch (in the *main.cpp* file). This might not be the best approach because it makes your code harder to read and troubleshoot. Additionally, it is often more practical to have HTML, CSS, and JavaScript on individual files. That's what we'll do in this Unit.

Project Overview

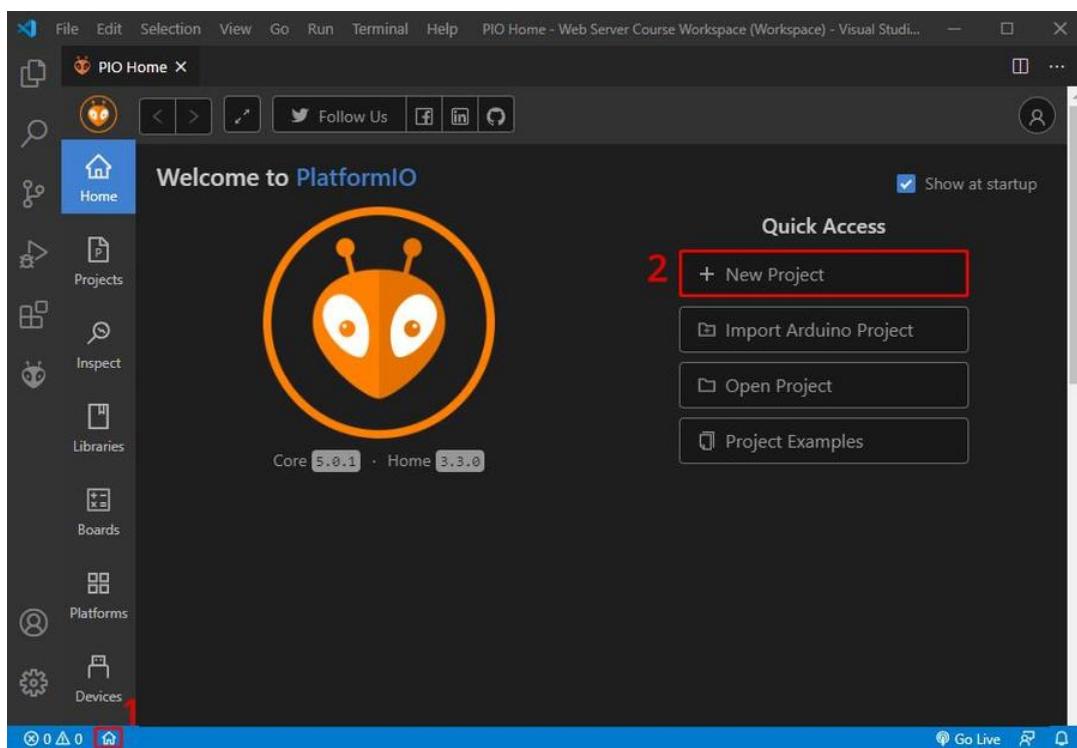


In this Unit, we'll show you how to build the same web server built in the previous project, but with HTML and CSS files stored on the ESP32/ESP8266 filesystem.

Instead of writing HTML and CSS in the Arduino sketch, we'll create separated HTML and CSS files. We'll also store an image file to serve a favicon to your web page. The favicon is the small icon that appears next to the title in the web browser tab.

Creating a New Project

Open VS Code. Go to PlatformIO **Home** (1) and create a new project by clicking on the **+New Project** button (2). Give your project a name, select the board you're using, and the location where you want to save it.



We'll call this new project `1_2_Hello_World_WS_SPIFFS`.

Your project is now created, and the project files should be accessible from the **File Explorer** tab at the left.

Organizing your Files

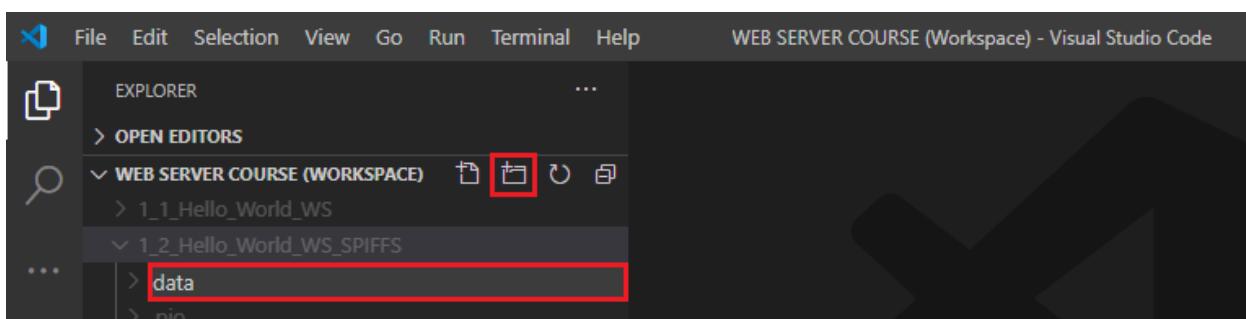
The files you want to upload to the ESP filesystem should be placed in a folder called *data* under the project folder. We'll move three files to that folder:

- *index.html* to build the web page
- *style.css* to style the web page
- *favicon.png*: a small image to add as favicon to the web page

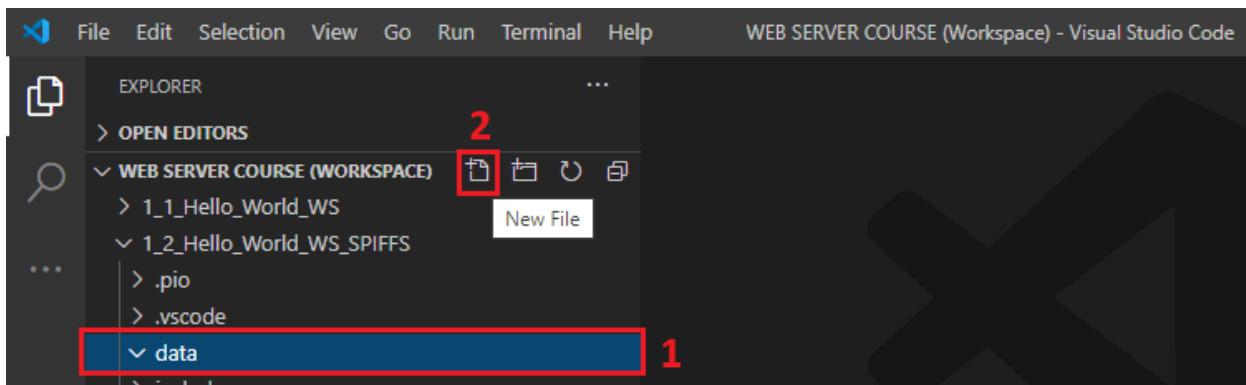
Creating a *data* Folder

Create a folder called *data* inside your project folder. This can be done on VS Code. With your mouse, select the project folder you're working on. Click on the **New Folder** icon to create a new folder.

This new folder must be called *data*. Otherwise, it won't work.



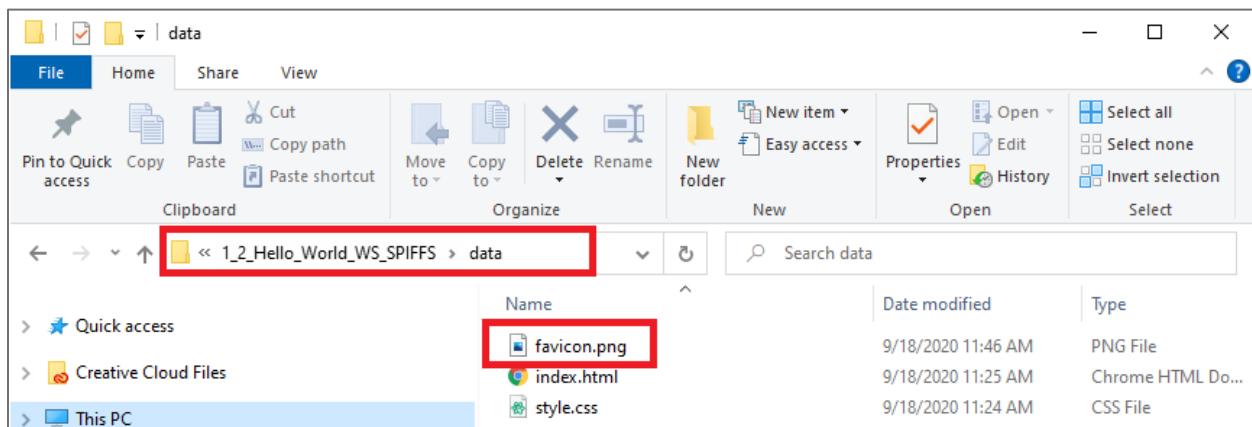
Then, select the newly created *data* folder and create two files inside by clicking on the **New File** icon.



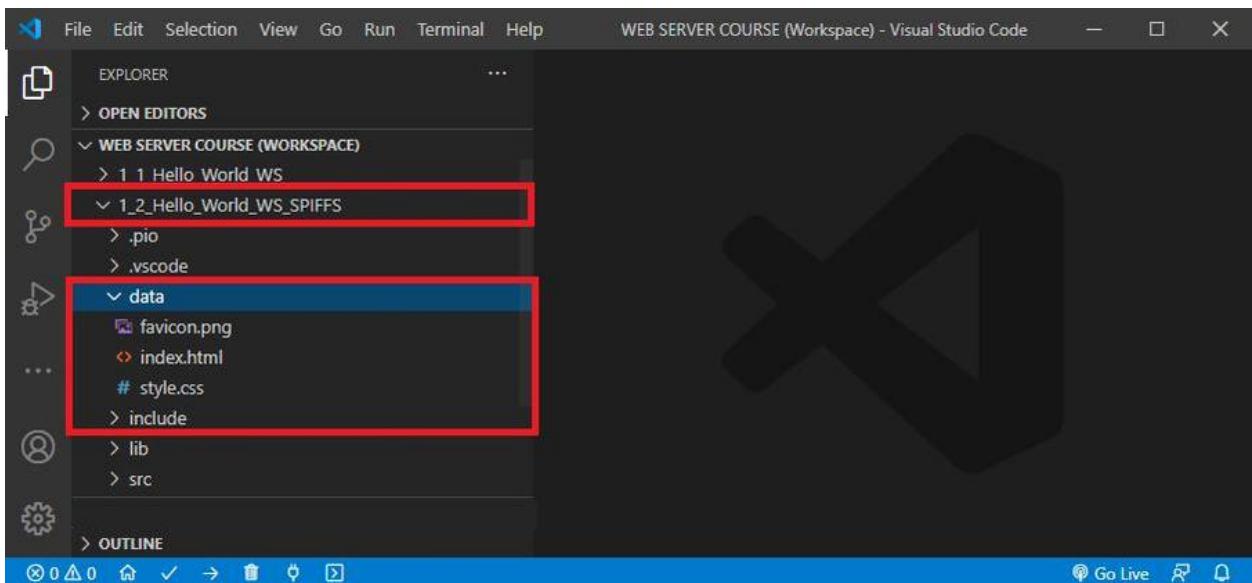
One file should be called *index.html*, and the other *style.css*.

You can give these files any other names as long as they have the *.html* and *.css* extensions. However, we recommend using these exact names. Otherwise, you also need to change the *main.cpp* file to make the web server work.

If you want to include a favicon in your web server, you can place an icon called *favicon.png* inside the *data* folder (you need to do it outside VS Code). We provide a favicon example in the project folder that you can use. The ideal size for the favicon is 16px x 16px.



After placing the image in the *data* folder, go back to VS Code. In the File Explorer, click on the **Refresh** icon. It will show the *data* folder with all the files.



HTML File

Copy the following to your *index.html* file. This contains the same HTML we've used to build the previous web server.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="style.css">
  <link rel="icon" type="image/png" href="favicon.png">
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
```

The only difference is that this time we're placing the HTML text on its own file, as well as the CSS. So, we must reference the CSS on the HTML file, as you've learned previously.

```
<link rel="stylesheet" type="text/css" href="style.css">
```

Additionally, we want to load a favicon, which you indicate as shown below:

```
<link rel="icon" type="image/png" href="favicon.png">
```

CSS File

The *style.css* file contains the styles to use on the web page. Copy this into your file and save it.

```
html {
  font-family: Arial;
  text-align: center;
}
body {
  max-width: 400px;
  margin: 0px auto;
}
```

platformio.ini file (ESP32)

The *platformio.ini* configuration file for the ESP32 should be like this (same as the previous project).

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The *platformio.ini* configuration file for the ESP8266 should be like this:

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
board_build.filesystem = littlefs
```

SPIFFS is currently deprecated and may be removed in future releases of the ESP8266 core. It is recommended to use *LittleFS* instead. LittleFS is under active development, supports directories, and is many times faster for most operations. The methods used for SPIFFS are compatible with LittleFS, so, we can simply use the expression `LittleFS` instead of `SPIFFS` in our code. However, when uploading files, we must specify that we want to use LittleFS filesystem instead of SPIFFS (default), so add the following line to the ESP8266 *platformio.ini* configuration file.

```
board_build.filesystem = littlefs
```

main.cpp file (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

void initSPIFFS() {
  if (!SPIFFS.begin(true)) {
    Serial.println("An error has occurred while mounting SPIFFS");
  }
  else{
    Serial.println("SPIFFS mounted successfully");
  }
}

void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

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

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

  server.serveStatic("/", SPIFFS, "/");
}

server.begin();
}

void loop() {
```

This code is very similar to the previous project but adds the lines of code to serve the files saved in the ESP32 filesystem.

Don't forget to modify the code to include your network credentials.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

Initialize SPIFFS

We'll use SPIFFS filesystem. To include the SPIFFS library in our code:

```
#include "SPIFFS.h"
```

You also need to initialize SPIFFS. The `initSPIFFS()` function does that:

```
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    else{
        Serial.println("SPIFFS mounted successfully");
    }
}
```

The `initSPIFFS()` function should be called in the `setup()`:

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

Serve Files From SPIFFS

When the ESP32 receives a request on the root URL, we want to send a response with content from a file stored in SPIFFS.

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

The first argument of the `send()` function is the filesystem where the files are saved. In this case, it is saved in SPIFFS. The second argument is the path where the file is located. The third argument refers to the content type (HTML text).

When the HTML file loads in your browser, it will request the CSS file and the favicon. These are static files saved in the same directory (SPIFFS). We can add the following line to serve static files in a directory when requested by the root URL. It serves the CSS and favicon files automatically.

```
server.serveStatic("/", SPIFFS, "/");
```

Alternatively, you could handle the request for the CSS file like this:

```
server.on("/style.css", HTTP_GET, [](AsyncWebRequest *request){
    request->send(SPIFFS, "/style.css", "text/css");
});
```

main.cpp file (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
}
```

```

    else{
        Serial.println("LittleFS mounted successfully");
    }
}

void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

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

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

    server.serveStatic("/", LittleFS, "/");
    server.begin();
}

void loop() {
}

```

This code is very similar to the previous project but adds the lines of code to serve the files saved in the ESP8266 filesystem (LittleFS).

Don't forget to modify the code to include your network credentials.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Initialize LittleFS

We'll use LittleFS filesystem, so we need to include the LittleFS library in the code.

```
#include <LittleFS.h>
```

You need to initialize LittleFS. The `initFS()` function does that:

```
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    else{
        Serial.println("LittleFS mounted successfully");
    }
}
```

The `initFS()` function should be called in the `setup()`:

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

Serve Files From LittleFS

When the ESP8266 receives a request on the `/` URL, we want to send a response with content from a file stored in LittleFS.

```
server.on("/", HTTP_GET, [](AsyncWebRequest *request){
    request->send(LittleFS, "/index.html", "text/html");
});
```

The first argument of the `send()` function is the filesystem where the files are saved. In this case, those are saved in LittleFS. The second argument is the path where the file is located. The third argument refers to the content type (HTML text).

When the HTML file loads in your browser, it will request the CSS file and the favicon. These are static files saved in the same directory (LittleFS). We can add the following line to serve static files in a directory when requested by the root URL. It serves the CSS and favicon files automatically.

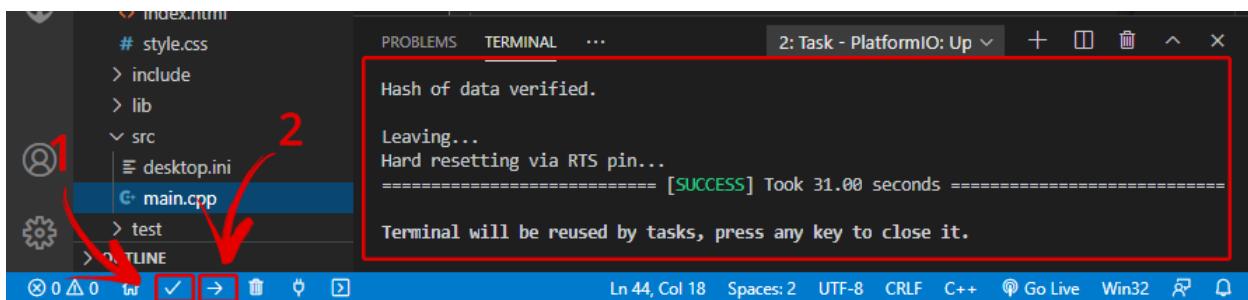
```
server.serveStatic("/", LittleFS, "/");
```

Alternatively, you could handle the request for the CSS file like this:

```
server.on("/style.css", HTTP_GET, [](AsyncWebRequest *request){
    request->send(LittleFS, "/style.css", "text/css");
});
```

Uploading Code

After modifying the code with your network credentials, click the **Compile** icon and then the **Upload** icon to upload your code.



Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, you need to upload the files (*index.html*, *style.css* and *favicon.png*) to the filesystem:

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.

The screenshot shows the PlatformIO IDE interface. On the left, the 'PROJECT TASKS' sidebar is open, displaying various development options. A red box highlights the 'env:esp32doit-devkit-v1' section, which is expanded. Within this section, a red box highlights the 'Upload Filesystem Image' option, which is then selected and highlighted with a blue border. The main area shows a portion of the 'main.cpp' file with code related to WiFi and SPIFFS. The status bar at the bottom indicates the terminal task is active.

```

27     delay(1000);
28 }
29 Serial.println(WiFi.localIP());
30 }

31 void setup() {
32 // Serial port for debugging purposes
33 Serial.begin(115200);
34 initWiFi();
35 initSPIFFS();

36 server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
37     request->send(SPIFFS, "/index.html", "text/html");
38 });

39 server.serveStatic("/", SPIFFS, "/");
40

41 server.begin();
42
43
44

```

Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. Or, if you're using the same board as the previous example, it will probably have the same IP.

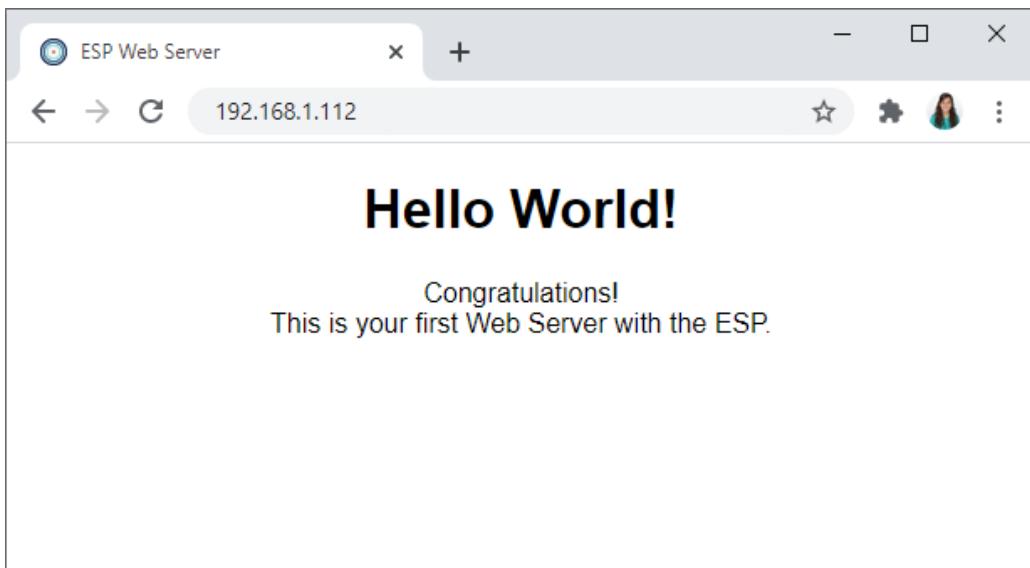
The screenshot shows the PlatformIO IDE with the 'TERMINAL' tab active. The terminal output displays the process of connecting to WiFi, with the last line showing the IP address '192.168.1.112'. A red arrow points from the bottom left towards the terminal window, indicating where to click to open it. The status bar at the bottom shows the terminal task is active.

```

clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5828
entry 0x400806ac
Connecting to WiFi ...192.168.1.112

```

Open a browser on your local network, and you should be able to access the "Hello World" web server. Notice that this time, it displays the favicon next to the web page title.



Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

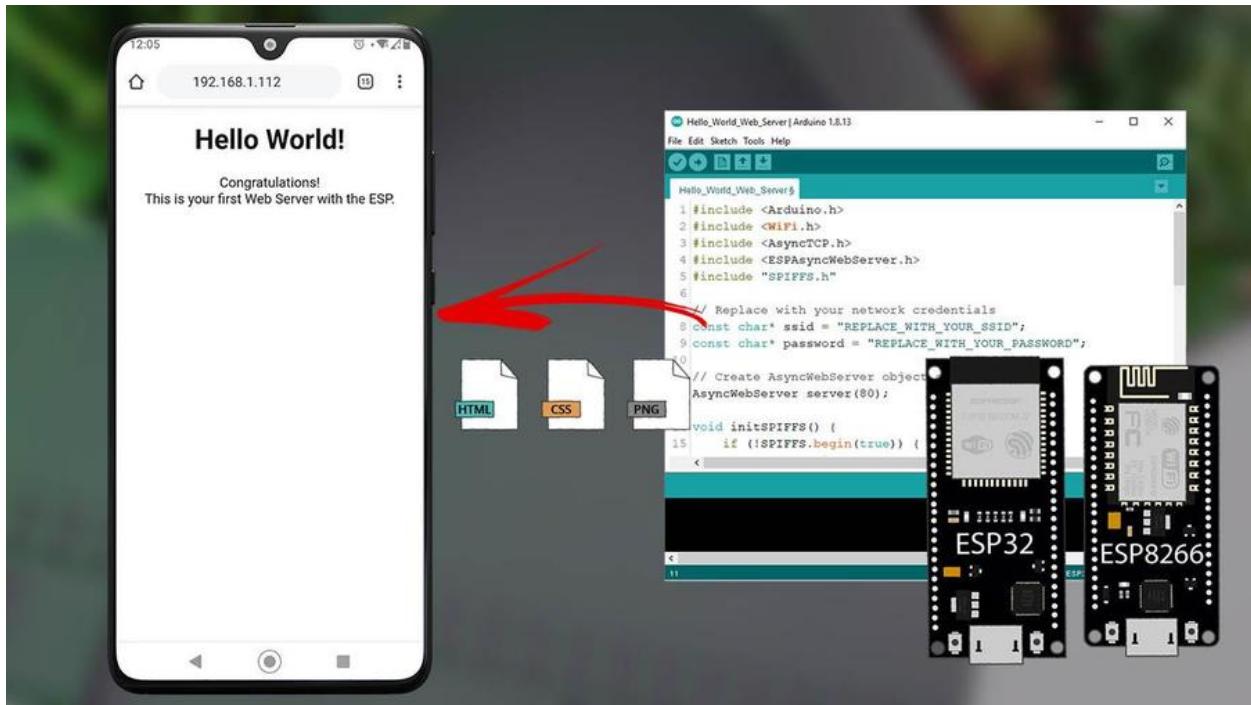
Wrapping Up

After these two web server projects, we hope that you are now more familiar with VS Code and the process to upload code to your board, create a *data* folder, work with multiple files and upload them to the board filesystem. That was the goal of these two projects.

The “Hello World” web server is not that interesting because it doesn’t allow you to interact with the ESP32/ESP8266 and vice-versa. In the next Units, you’ll finally start building web servers to interact with your boards.

If you don’t like using VS Code + PlatformIO to program the ESP32 and ESP8266 boards and you want to use Arduino IDE, follow the next Unit. We’ll show you how to proceed if you’re going to use Arduino IDE software instead.

1.3 - Hello World Web Server (Arduino IDE Software)



If you don't like using VS Code + PlatformIO IDE to program the ESP32 and ESP8266 boards, you can follow this Unit to learn how to upload code and files to the filesystem using Arduino IDE software. We'll build the same web server as the previous project. If you prefer to use VS Code + PlatformIO, you can skip this section.

Creating a New Project

Open Arduino IDE software and create a new file.

ESP32 Sketch

Copy the following code to the Arduino IDE if you're using an ESP32 (the same as the previous project).

```

#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}
void setup() {
    Serial.begin(115200);
    initWiFi();
    initSPIFFS();

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

    server.serveStatic("/", SPIFFS, "/");
    server.begin();
}

void loop() {
}

```

Don't forget to modify the code to include your network credentials.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Save your project, for example: `1_3_SPIFFS_Arduino`.

ESP8266 Sketch

If you're using an ESP8266 board, copy the following code instead.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>
// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
// Create AsyncWebServer object on port 80
AsyncWebServer server(80);

void initFS() {
  if (!LittleFS.begin()) {
    Serial.println("An error has occurred while mounting LittleFS");
  }
  Serial.println("LittleFS mounted successfully");
}

void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

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

  server.on("/", HTTP_GET, [] (AsyncWebRequest *request){
    request->send(LittleFS, "/index.html", "text/html");
  });
  server.serveStatic("/", LittleFS, "/");
  server.begin();
}

void loop() {
```

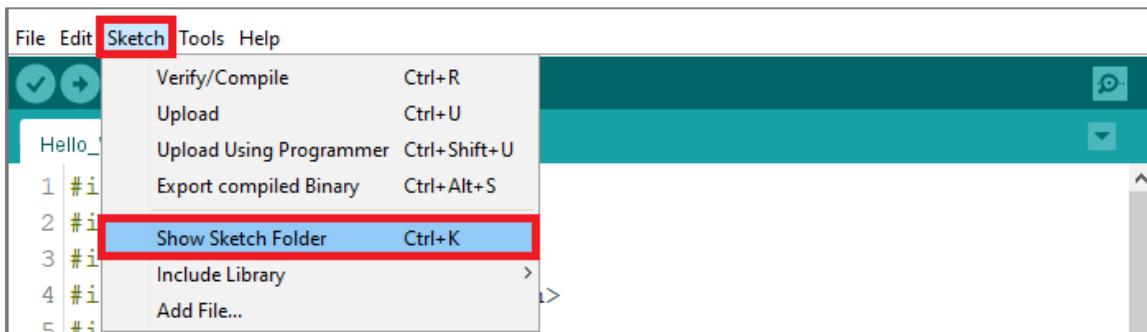
Don't forget to modify the code to include your network credentials.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

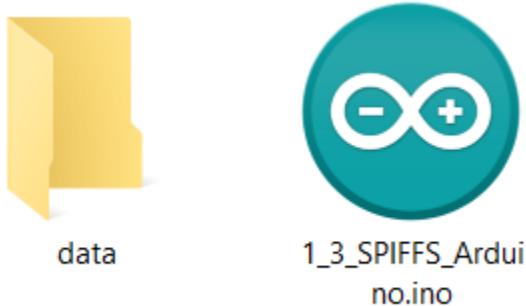
Save your project, for example: `1_3_LITTLEFS_ESP8266_Arduino`.

Creating a data Folder

In your Arduino IDE, go to **Sketch > Show Sketch Folder** or press **Ctrl+K**.



Inside your sketch folder, create a new folder called *data*.



Move the *index.html*, *style.css* and *favicon.png* files to the *data* folder.

HTML File

Here's the content of the *index.html* file.

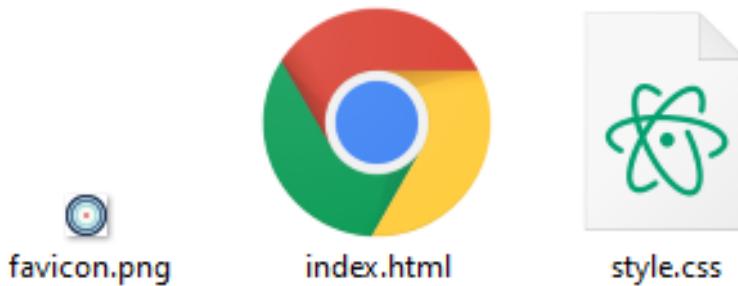
```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP Web Server</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" href="style.css">
    <link rel="icon" type="image/png" href="favicon.png">
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
  </body>
</html>
```

CSS File

Here's the content of the *style.css* file.

```
html {  
    font-family: Arial;  
    text-align: center;  
}  
body {  
    max-width: 400px;  
    margin: 0px auto;  
}
```

Here are all the files you should save on the *data* folder.

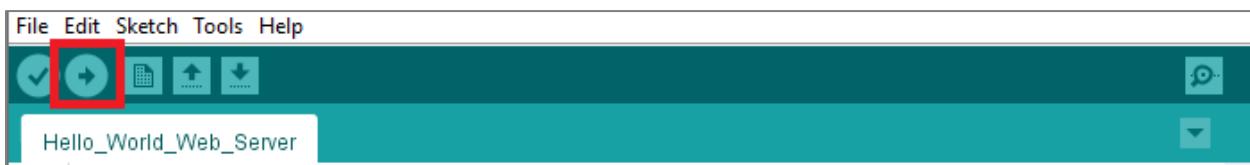


Uploading Code

After modifying the code with your network credentials, go to **Tools > Port** and select the COM port your board is connected to.

Go to **Tools > Board** and select the ESP board you're using. If you're using an ESP8266 board, you may also need to change the SPIFFS size on the "Flash size" option.

Finally, click on the Upload icon.

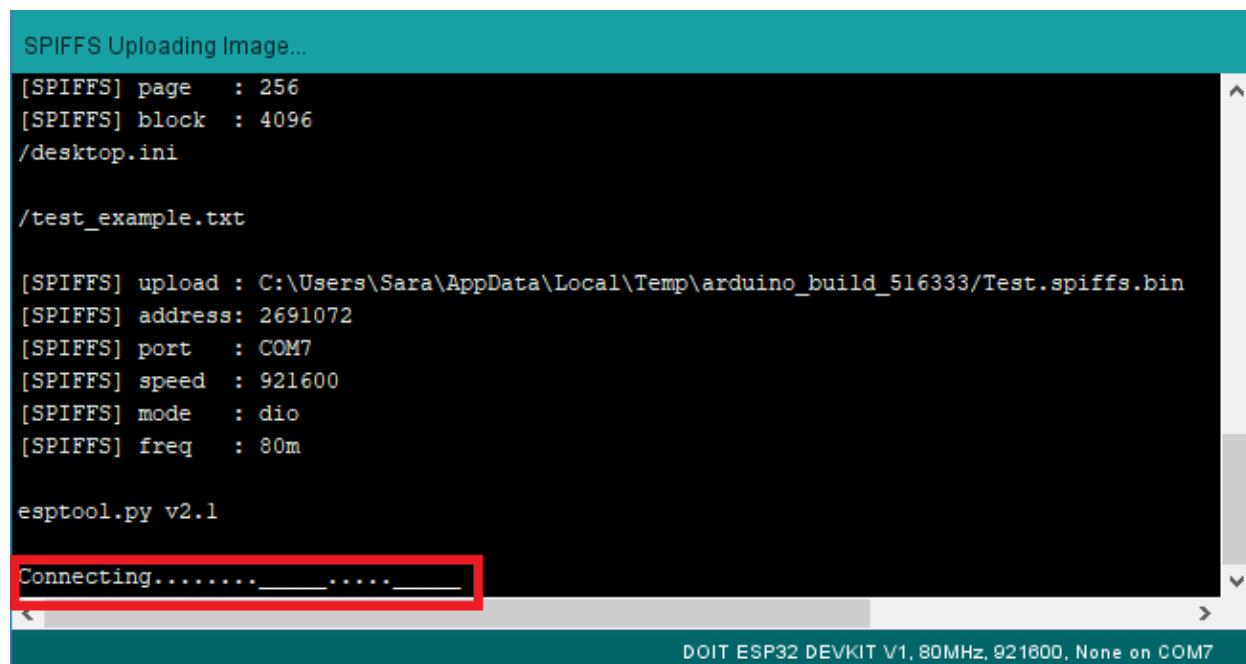


Uploading Filesystem Image

Finally, go to the **Tools** menu and select **ESP32 Data Sketch Upload** or **ESP8266 LittleFS Data Upload**, depending on the board your using.

Important: Make sure you have the Serial Monitor closed while trying to upload files to the filesystem. Otherwise, it will fail.

Note: in some ESP32 development boards, you need to press the ESP32 on-board BOOT button when you see the “Connecting_____” message.



```
SPIFFS Uploading Image...
[SPIFFS] page   : 256
[SPIFFS] block  : 4096
/desktop.ini

/test_example.txt

[SPIFFS] upload : C:\Users\Sara\AppData\Local\Temp\arduino_build_516333/Test.spiffs.bin
[SPIFFS] address: 2691072
[SPIFFS] port   : COM7
[SPIFFS] speed  : 921600
[SPIFFS] mode   : dio
[SPIFFS] freq   : 80m

esptool.py v2.1

Connecting....._____
```

After a few seconds, you should get the message “**SPIFFS Image Uploaded**” or “**LittleFS Image Uploaded**” as shown in the following picture. The files were successfully uploaded to the ESP32 or ESP8266 filesystem.



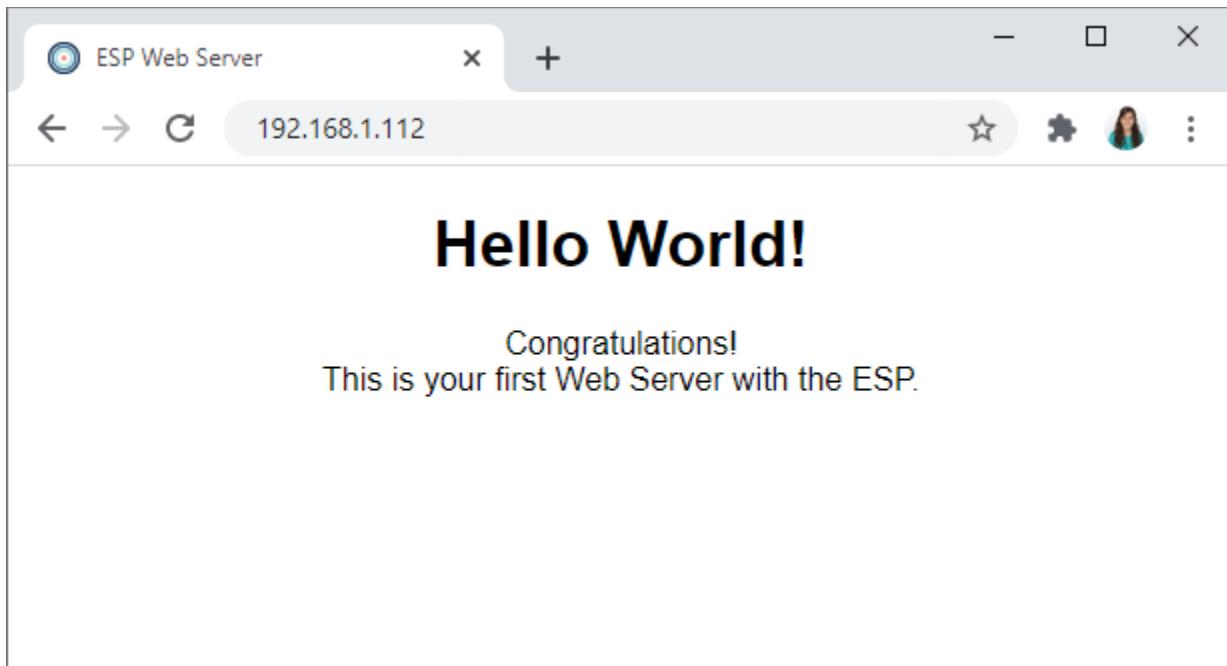
```
SPIFFS Image Uploaded
Hash of data verified.

Leaving...
Hard resetting...
DOIT ESP32 DEVKIT V1. 80MHz, 921600, None on COM7
```

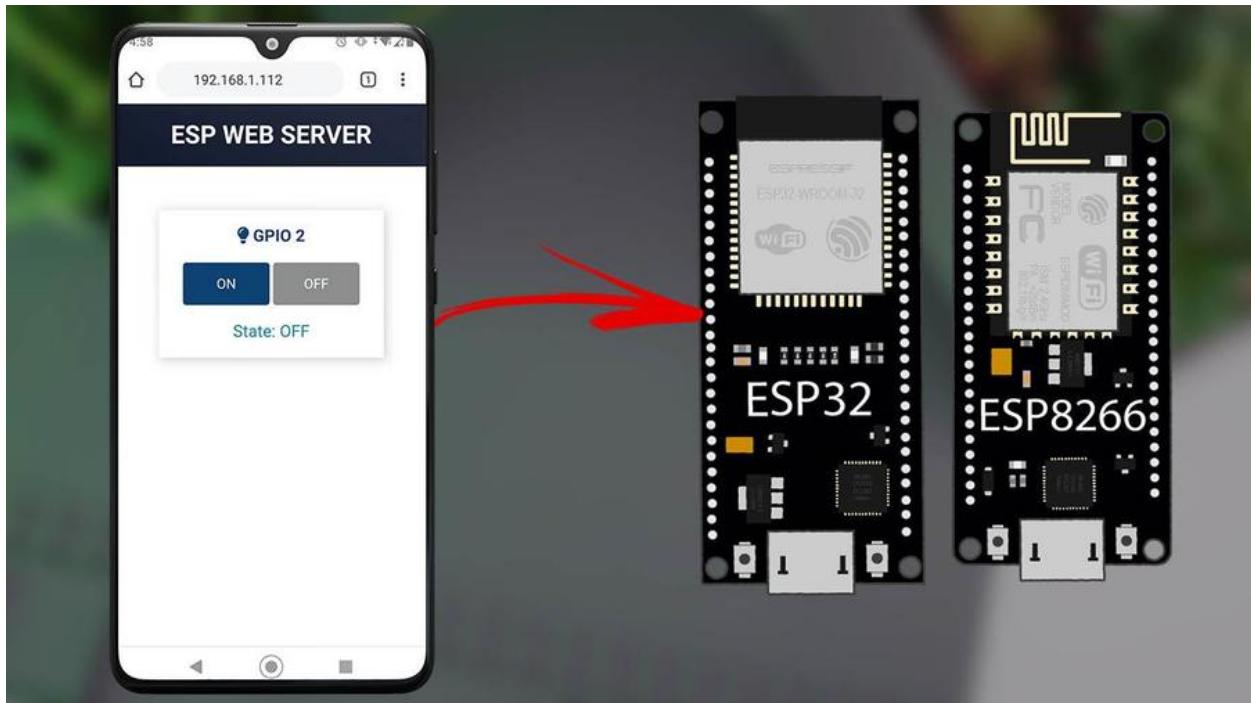
Demonstration

Now, you have everything ready to start your web server. With your board connected to your computer, press the on-board EN/RST button. Its IP address will be printed in the Serial Monitor.

Open a browser on your local network and type the ESP IP address. You should get access to the “Hello World” web server.



2.1 - Web Server - Control Outputs (ON/OFF Buttons)

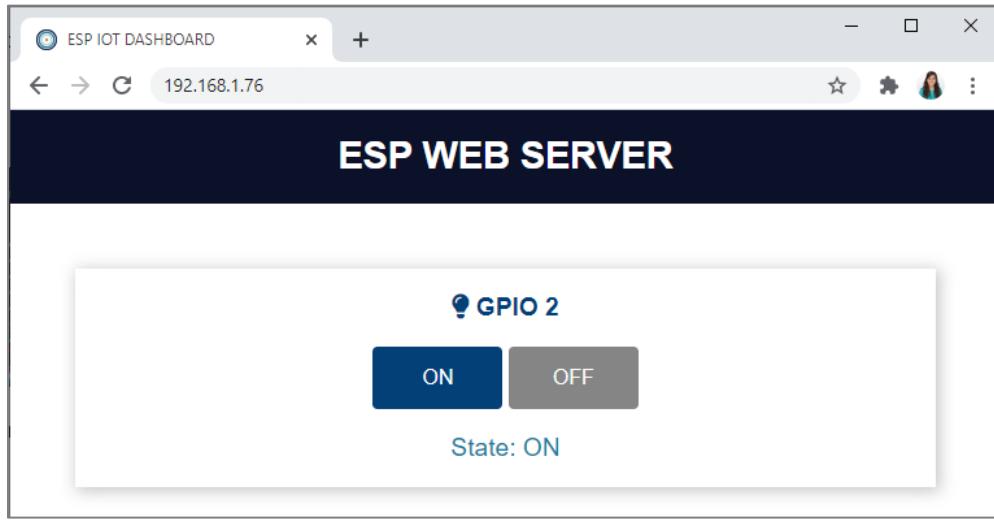


In this Unit, you'll create a web server to control the ESP32 or ESP8266 outputs. The web server displays ON and OFF buttons to control GPIO 2 (the on-board LED of the ESP32 and ESP8266). You can modify the project to control any other GPIOs. To keep things as simple as possible, we'll control one single GPIO. In the following Units, you'll learn how to control multiple outputs.

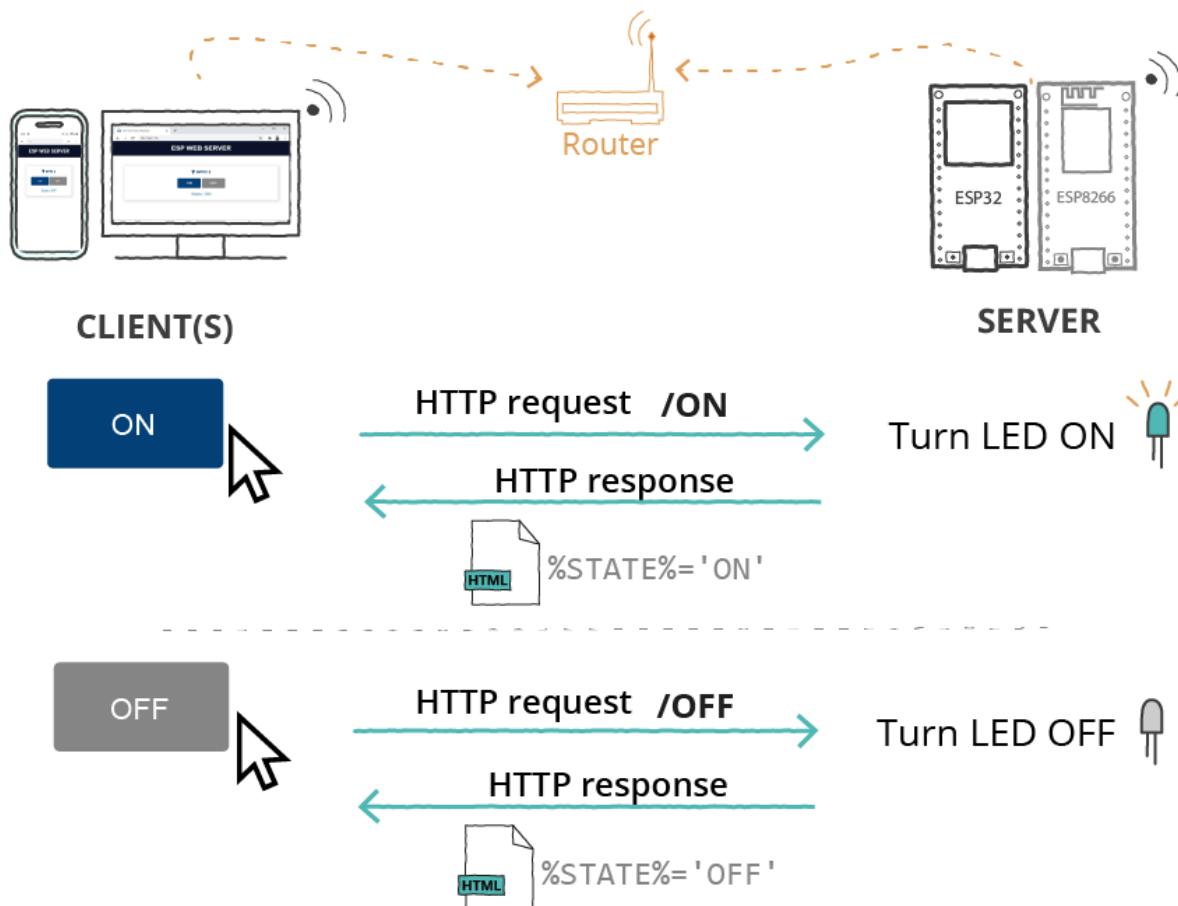
Project Overview

The web server we'll build in this Unit serves a web page that displays:

- ON button: turns GPIO 2 on.
- OFF button: turns GPIO 2 off.
- GPIO state: shows current GPIO state, either ON or OFF.



For this project, we'll use the following approach to control the ESP32/ESP8266 GPIOs:



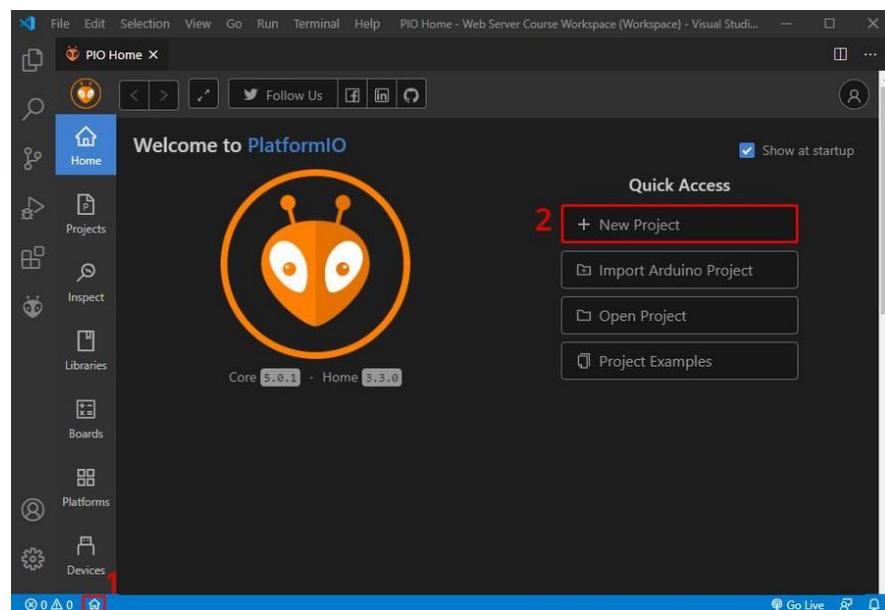
- When you type the ESP IP address in a browser on your local network, you can access the web page served by your board.

- When you click the **ON** button, you are redirected to the ESP IP address followed by /ON, so the ESP receives a request /ON.
- When you click on the **OFF** button, you are redirected to the ESP IP address followed by /OFF, so the ESP receives a request /OFF.
- Depending on the URL received, we'll control the GPIO state accordingly and serve the web page with the current GPIO state.
- In summary, when you click the buttons, the ESP turns the LED either on or off and sends the HTML text to build the web page. The HTML text is always the same except for the state of the GPIO.

Note: there are other ways to control the ESP GPIOs using a web server. In the next Units, we'll show you other approaches that may be more suitable for your projects, depending on the application. This approach is one of the simplest.

Creating a New Project

Open VS Code. Go to PlatformIO **Home** and create a new project by clicking on the **+New Project button**. Give your project a name, select the board you're using, and the location where you want to save it.



We'll call this new project *2_1_Output*.

Your project is now created, and the project files should be accessible from the **File Explorer** tab at the left.

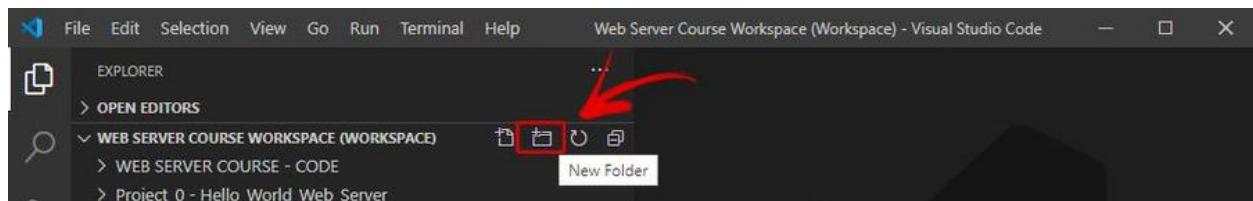
Organizing your Files

The files you want to upload to the ESP file system should be placed in a folder called *data* under the project folder. We'll have the following files in that folder:

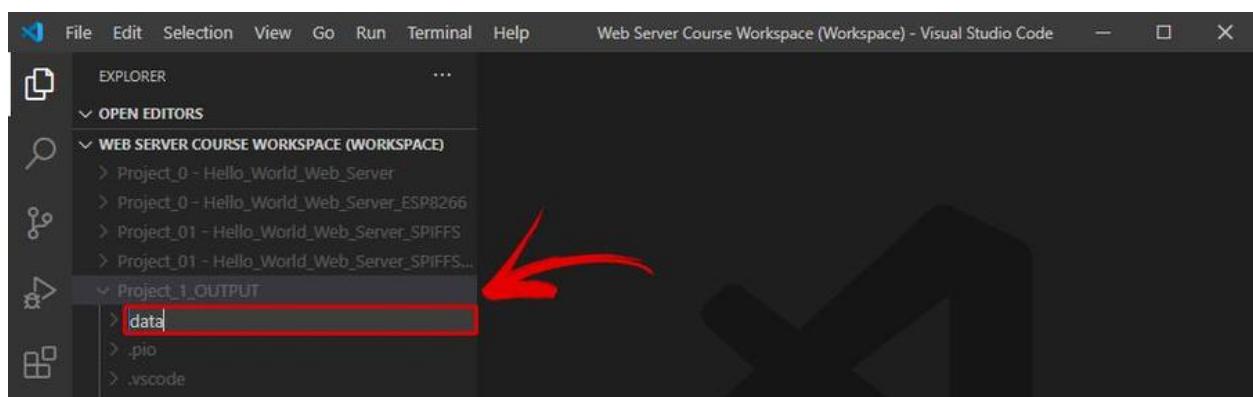
- *index.html* to build the web page
- *style.css* to format the web page
- *favicon.png*: a small image to add a favicon to the web page

Creating a *data* Folder

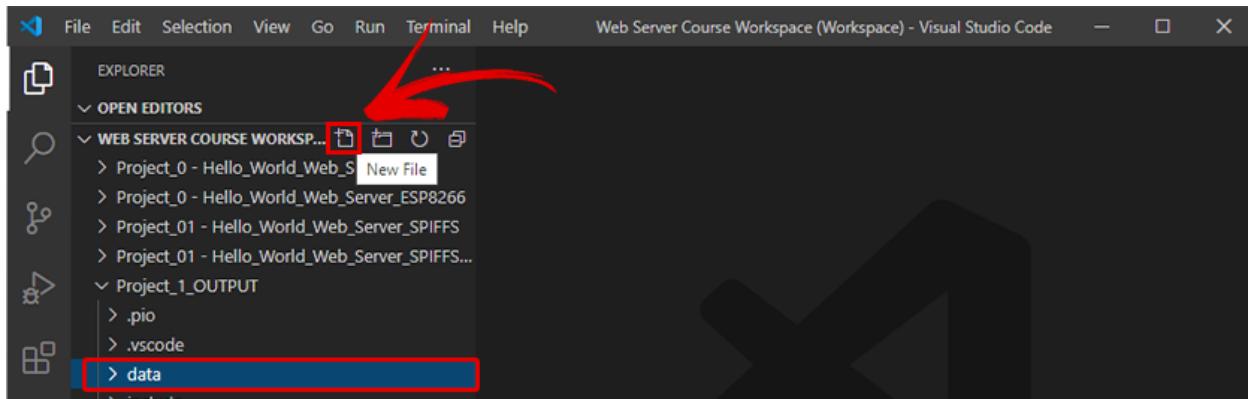
Create a folder called *data* inside your project folder. This can be done on VS Code. With your mouse, select the project folder you're working on. Click on the **New Folder** icon to create a new folder.



This new folder must be called *data*. Otherwise, it won't work.



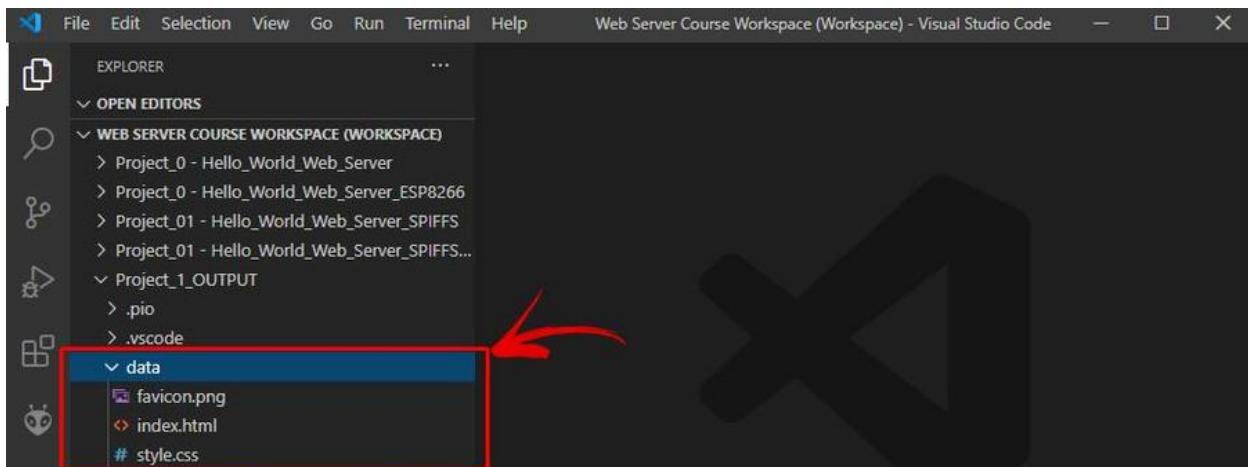
Then, select the newly created *data* folder and create two files inside by clicking on the **New File** icon.



One file should be called *index.html*, and the other *style.css*.

If you want to include a favicon in your web server, you can place an icon called *favicon.png* image inside the *data* folder. We provide a favicon example in the project folder that you can use. The ideal size for the favicon is 16px x 16px.

After placing the image in the *data* folder, go back to VS Code. In the File Explorer, click on the **Refresh** icon. The File Explorer shows the following folders and files under your project folder.



Now that you have all your files and folders organized, let's start building the web server.

Building the Web Page

For this project, we'll build the web page shown in the Project Overview section.

Here's the web page with some annotations relating to its `<div>` tags and elements' class names.



Head

In the head of the HTML file (between the `<head></head>` tags) include the title. In our case the title is `ESP IOT Dashboard` but you can give it any other title. The title goes between the `<title></title>` tags:

```
<title>ESP IOT DASHBOARD</title>
```

Include the following meta tag for a responsive web design:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Because we'll use a separate CSS file, we must reference it in the head of our HTML document like this:

```
<link rel="stylesheet" type="text/css" href="style.css">
```

Include the favicon. The favicon image is called *favicon.png*. You can download the image into the project folder.

```
<link rel="icon" type="image/png" href="favicon.png"></head>
```

Finally, include the Font Awesome website styles to include icons in the web page like the lightbulb icon.

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
```

Top Bar

There's a top bar with a heading in the web page. It is a heading 1 and it is placed inside a `<div>` tag with the class name `topnav`. Placing your HTML elements between `<div>` tags, makes them easy to style using CSS.

```
<div class="topnav">
  <h1>ESP WEB SERVER</h1>
</div>
```

Content Grid

All the other content is placed inside a `<div>` tag called `content`.

```
<div class="content">
```

The buttons are inside a box (div) with a grey margin. That div looks like a card, so we choose `card` for its class name.

```
<div class="card">
```

If you want to add more cards and have everything aligned, it is good to use the CSS grid layout. Go back to the CSS Unit if you don't remember what the CSS grid is).

Our card corresponds to a grid cell. Grid cells need to be inside a grid container, so it needs to be placed inside another `<div>` tag. This has the class name `card-grid`.

```
<div class="content">
  <div class="card-grid">
    <div class="card">
```

Card Title

Inside the card there's a paragraph displaying a title for the card (GPIO 2). Because we want to style it individually, we assigned a class name to that, in this case `card-title`.

```
<p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
```

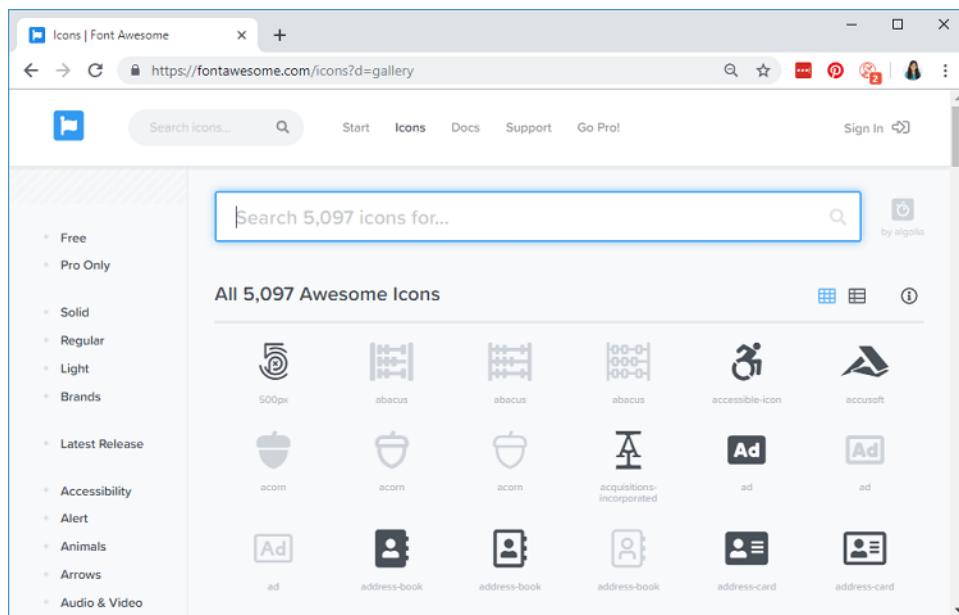
Before the `GPIO 2` text, there's a small lamp icon. That's a Font Awesome icon and it is displayed using the `<i></i>` tags with a specific class name.

```
<i class="fas fa-lightbulb"></i>
```

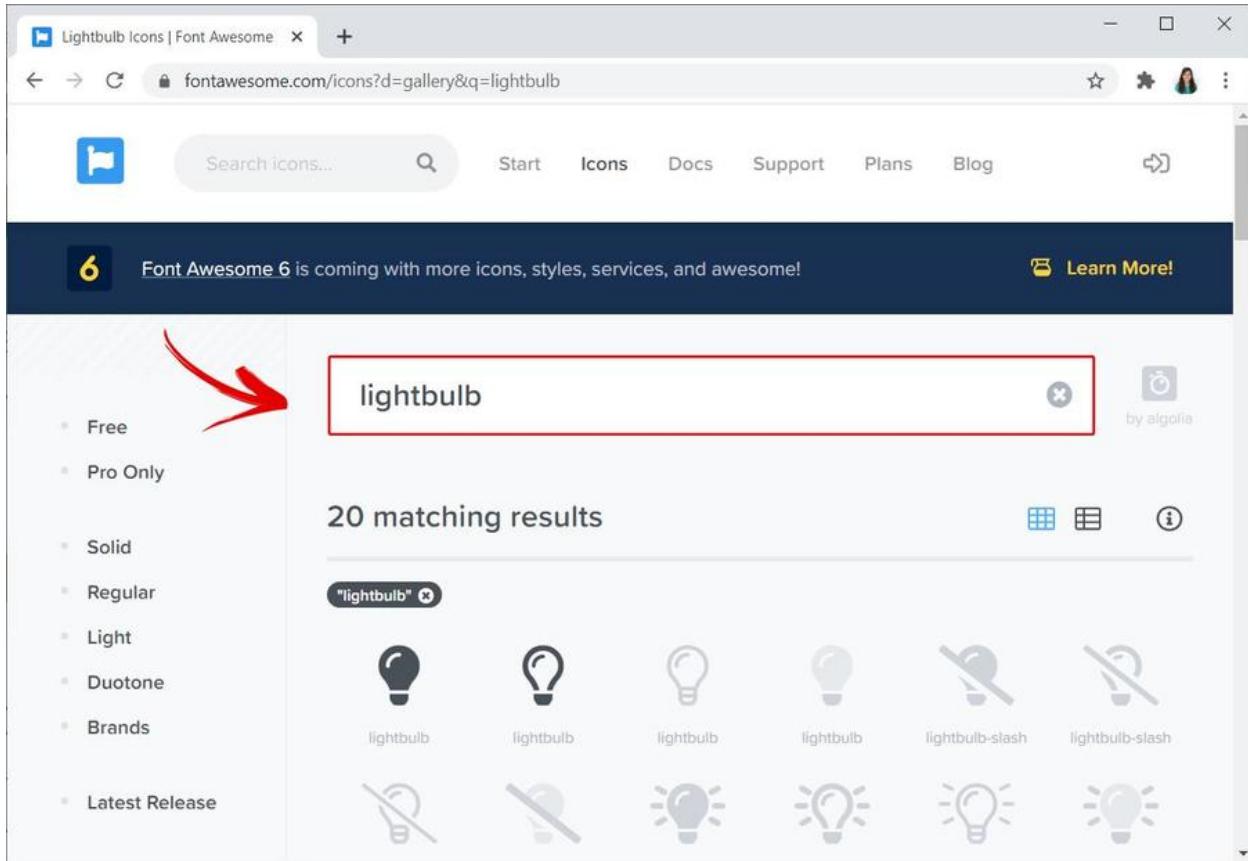
Using Other Icons from Font Awesome

Simply go to the Font Awesome Icons Website, choose the icon(s) of interest, and copy the HTML text displayed for that icon into your project's `.html` file, as we have done with the lightbulb icon. Here's the step-by-step process:

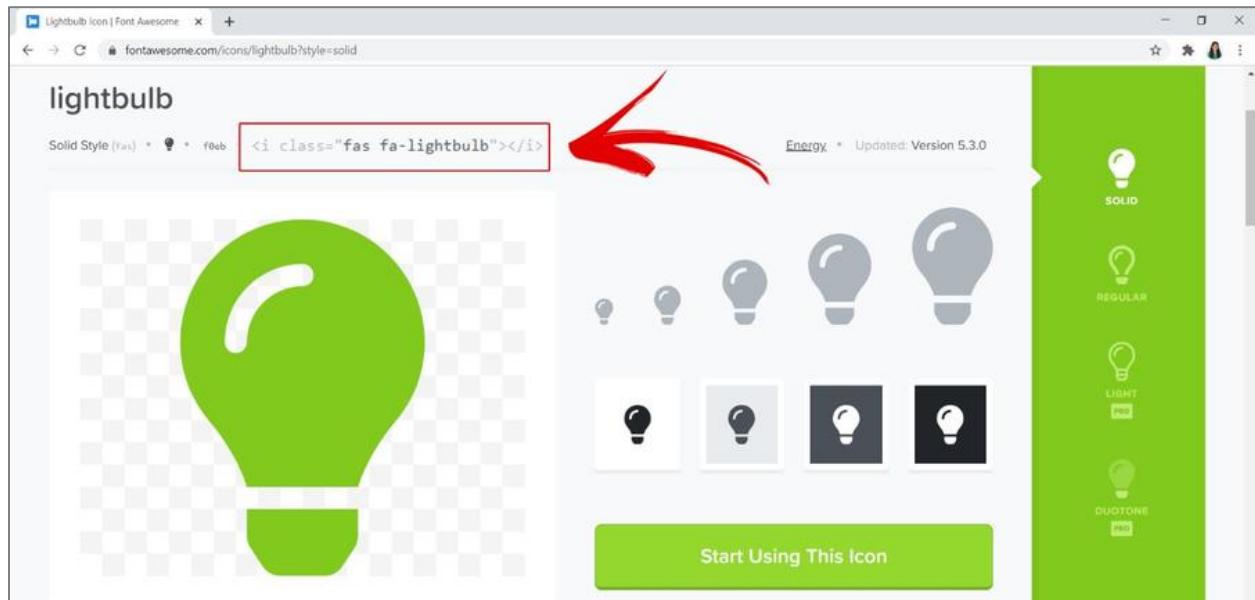
- 1) To choose the icons, go to the [Font Awesome Icons website](https://fontawesome.com/icons?d=gallery).



2) Search the icon you're looking for. For example, "lightbulb".



3) Select the desired icon. Then, you just need to copy the HTML text provided.



ON/OFF Buttons

Inside the card, there are two buttons. We'll place those buttons inside paragraph tags `<p></p>`. You already know how to create a button. Use the `<button></button>` tags and write the button text between those tags. Creating the on and off buttons is as simple as this:

```
<p>
  <button>ON</button>
  <button>OFF</button>
</p>
```

Did you notice that the ON and OFF buttons are styled differently? To be able to do that later on using CSS, we need to give each button a different class name, for example `button-on` and `button-off`.

```
<p>
  <button class="button-on">ON</button>
  <button class="button-off">OFF</button>
</p>
```

We mentioned previously in the *Project Overview* section that we want to be redirected to the `/on` URL when we click the ON button and to the `/off` URL when we click the OFF button. So, add a hyperlink to those buttons using the `<a>` tags like this:

```
<p>
  <a href="on"><button class="button-on">ON</button></a>
  <a href="off"><button class="button-off">OFF</button></a>
</p>
```

The `href` attribute specifies the link's destination.

GPIO State

Finally, there's a paragraph displaying the GPIO state. The state must change according to the current GPIO state. We can add a placeholder in our HTML text that

will then be replaced with the current value by the ESP board when sending the web page to the client. The placeholders in the HTML text should go between % signs.

```
<p class="state">State: %STATE%</p>
```

This means that this %STATE% text is like a variable that will then be replaced by the actual value.

HTML File

Here's the complete HTML text. Copy it to your *index.html* file and save it.

```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" href="style.css">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWI1j8LyTjo7m0USTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
  </head>
  <body>
    <div class="topnav">
      <h1>ESP WEB SERVER</h1>
    </div>
    <div class="content">
      <div class="card-grid">
        <div class="card">
          <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
          <p>
            <a href="on"><button class="button-on">ON</button></a>
            <a href="off"><button class="button-off">OFF</button></a>
          </p>
          <p class="state">State: %STATE%</p>
        </div>
      </div>
    </div>
  </body>
</html>
```

Open your HTML document in your browser. This is what you get:



At the moment, it looks weird because it doesn't have any styles applied. That's what we're going to do in the next section.

Styling the Web Page

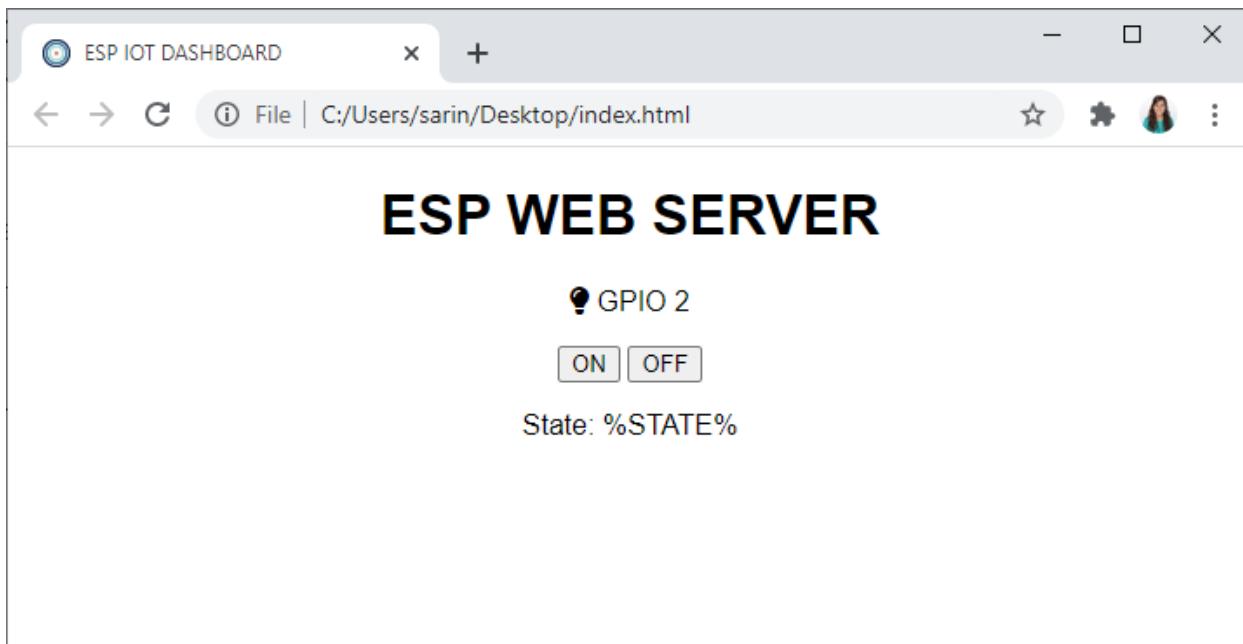
Open the `style.css` file to start styling your web page.

Styling the HTML Document

All the text in the HTML document is aligned at the center. So, add the `text-align` property with the `center` value to the `<html>` selector. Additionally, if you want to set a specific font to be used throughout your web page, this is a good place to do it. It will be applied to all elements in the HTML document.

```
html {  
  font-family: Arial, Helvetica, sans-serif;  
  text-align: center;  
}
```

Save your file. Open the HTML file in your browser and this is what you should get:



Styling the topnav div

The `topnav` div contains the `ESP WEB SERVER` text. That text is a heading 1 element. It is styled with a white color and it is a little bigger than the default heading 1 font. Use the `font-size` and `color` properties to style it:

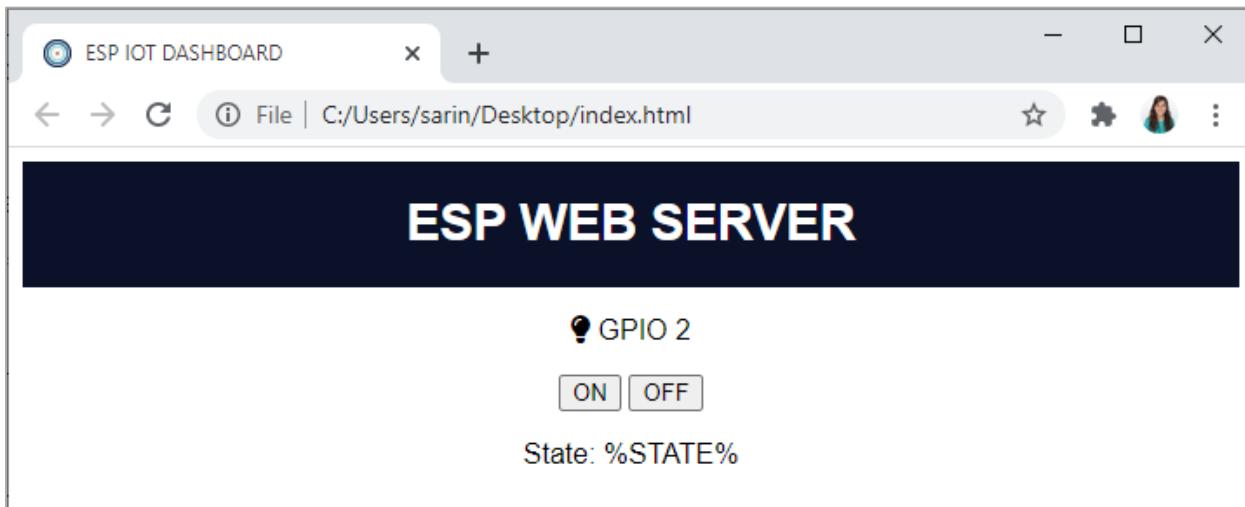
```
h1 {  
  font-size: 1.8rem;  
  color: white;  
}
```

The `rem` Unit for the font size is relative to the font-size of the root element. This means that the font size is 1.8 times bigger than the default size used by the browser.

Set the `topnav` background color using the `background-color` property. You can choose any background color. We're using `#0A1128`. Additionally, set the `overflow` property to `hidden` like this:

```
.topnav {  
  overflow: hidden;  
  background-color: #0A1128;  
}
```

The top bar should now be styled, as shown in the following screenshot.



Styling the buttons

We use the following properties to style the buttons.

```
Button {  
    border: none;  
    color: #FEFCFB;  
    padding: 15px 32px;  
    text-align: center;  
    display: inline-block;  
    font-size: 16px;  
    width: 100px;  
    border-radius: 4px;  
    transition-duration: 0.4s;  
}
```

By default, buttons have a border. In our example, we want to get rid of the border, set the `border` property to `none`.

`border: none;`

The `color` property sets the color of the text inside the button element. The `#FEFCFB` is the color we choose (it's almost white). You can choose any other color.

`color: #FEFCFB;`

The `padding` property adds a padding to the button: 15px at the top and bottom and 32px at the left and right.

`padding: 15px 32px;`

If you're confused about what padding is, go back to the CSS Unit and look at the CSS *Box Model*.

The text of the buttons is aligned at the center.

```
text-align: center;
```

We had already set the `text-align` to `center` in the HTML document before. Because the `button` element is a child of the `html` element, it will inherit its properties. So, it is not necessary to include it.

Set the `width`, `font-size` and `border-radius` of the button like this:

```
font-size: 16px;  
width: 100px;  
border-radius: 4px;
```

The `border-radius` property adds a round corner to the button. You can modify its size and see how it changes.

We'll add an effect to the buttons to make them cooler so that the buttons change color when you hover your mouse over them.

The `transition-duration` property specifies how many seconds a transition effect takes to complete. This is the time that the button will take to change color. Adding this property makes the transition smoother.

```
transition-duration: 0.4s;
```

The on and off buttons have different background colors. We can style them using their class names. Set the background color of each button like this:

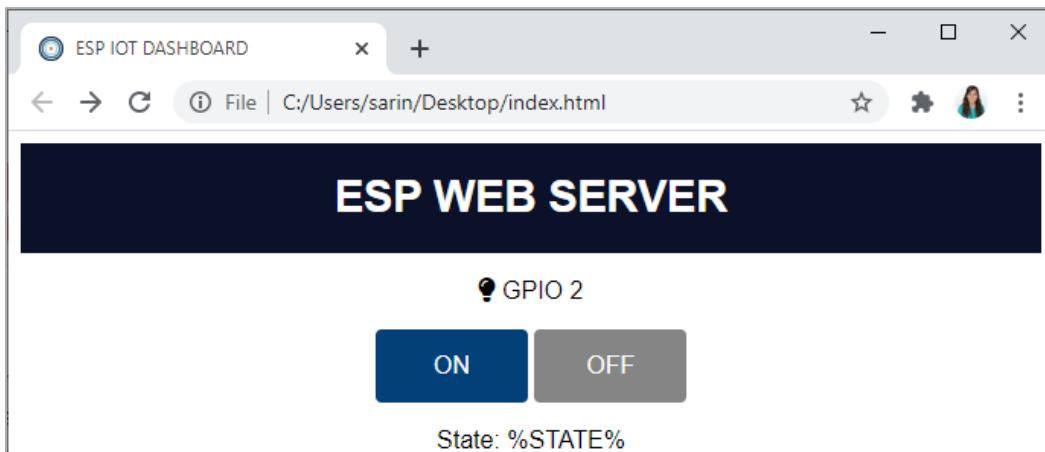
```
.button-on {  
    background-color: #034078;  
}  
.button-off {  
    background-color: #858585;  
}
```

Set the color of the buttons when you hover your mouse over them. To do that, you must use their class names followed by the `:hover` selector like this:

```
.button-on:hover {  
    background-color: #1282A2;  
}  
.button-off:hover {  
    background-color: #252524;  
}
```

Note: the `:hover` selector can be used in all HTML elements, not just buttons.

Save your CSS file and refresh your web browser. Now, your buttons are styled.



Styling the card title and state

Style the card title's color, size and set it to bold.

```
.card-title {  
    font-size: 1.2rem;  
    font-weight: bold;  
    color: #034078  
}
```

Additionally, style the paragraph that displays the state: the `color` and `font-size` properties.

```
.state {  
    font-size: 1.2rem;  
    color: #1282A2;  
}
```

You can choose any other colors and font sizes. Here's the result:



CSS Grid Layout

Set the `card-grid` properties as shown below.

```
.card-grid {  
    max-width: 800px;  
    margin: 0 auto;  
    display: grid;  
    grid-gap: 2rem;  
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));  
}
```

The `max-width` property, as the name suggests, sets the maximum width of the grid.

To place everything aligned at the center, set the `margin` of `card-grid` to `0 auto`.

Setting the `margin` property to `0 auto` centers the division within its parent container.

It is the same as setting the top and bottom margins to zero and the left and right margins to `auto`. The browser automatically distributes the right amount of margin to either side.

The `display` property sets the `card-grid` element as a `grid` container. The `grid-gap` defines the space between each grid cell (box) – in this case, we just have one grid cell.

The `grid-template-columns` property sets the width of the columns. The `repeat` means that it will apply the same settings to all columns. The `minmax(200px, 1fr)`

means that the width of the columns adjusts to the size of the browser window to a minimum of 200px. The `1fr` allows you to set the size of a track as a fraction of the free space of the grid container. For example, if you had three cards, it would set each item to one third the width of the grid container. At the moment, this may not be noticeable because we're just using one grid cell—you'll better understand how this works when we add more cards in future projects.

To learn more about the `fr` unit, take a look at the following article: <https://css-tricks.com/introduction-fr-css-unit/>

Style the `card` (grid cell) with these properties:

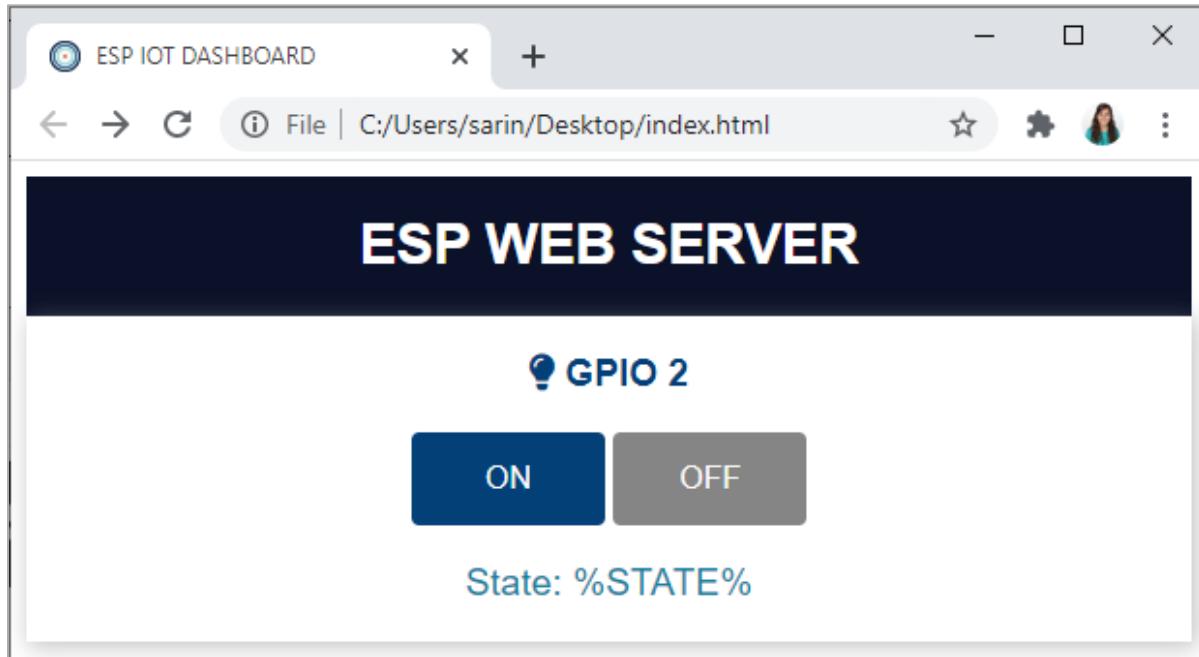
```
.card {  
  background-color: white;  
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);  
}
```

The `background-color` of the card is set to white, but you can use any other color. The `box-shadow` property attaches one or more shadows to the element. Here's what each value means from left to right:

- offset-x: horizontal distance;
- offset-y: vertical distance;
- blur-radius: the larger this value, the bigger the blur, so the shadow becomes bigger and lighter;
- spread-radius: positive values will cause the shadow to expand and grow bigger, negative values will cause the shadow to shrink.
- color: we're using RGBA color code. RGBA stands for red green blue alpha. The alpha corresponds to the opacity of the color.

We recommend that you play with the `box-shadow` properties to better understand how they work.

Save your file and refresh your web browser. Here's the result:



Styling the body and content

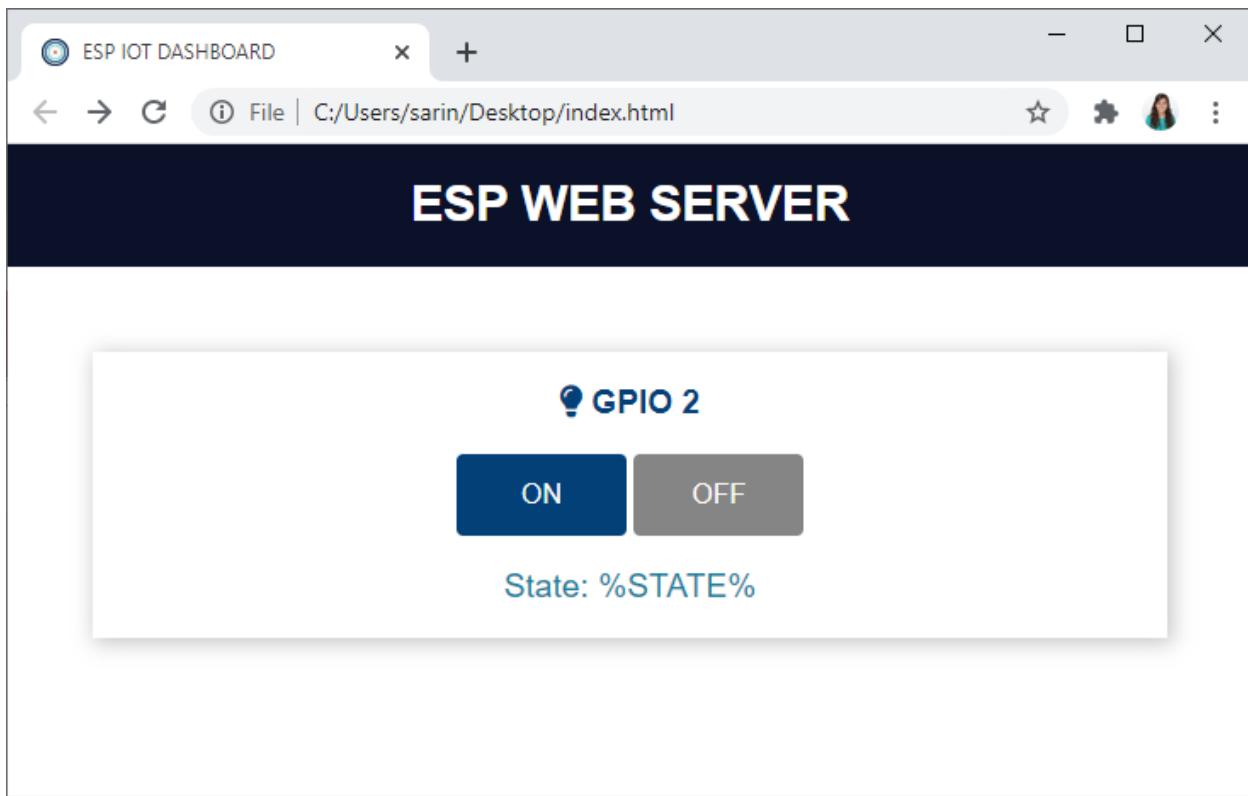
To make that weird margin around the body of the web page disappear, set the body `margin` to 0.

```
body {  
  margin: 0;  
}
```

Then, add some padding to the `content` div, so that the card grid doesn't collapse to the body's margins.

```
.content {  
  padding: 50px;  
}
```

Save the file. And voilà, here's the result.



Now that you know how to build the web page, you can play with the CSS properties to change its look.

CSS File

Here's the complete `style.css` file.

```
html {  
    font-family: Arial, Helvetica, sans-serif;  
    text-align: center;  
}  
h1 {  
    font-size: 1.8rem;  
    color: white;  
}  
.topnav {  
    overflow: hidden;  
    background-color: #0A1128;  
}  
body {  
    margin: 0;  
}  
.content {  
    padding: 50px;  
}
```

```

.card-grid {
  max-width: 800px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
  background-color: white;
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
  font-size: 1.2rem;
  font-weight: bold;
  color: #034078
}
.state {
  font-size: 1.2rem;
  color: #1282A2;
}
Button {
  border: none;
  color: #FEFCFB;
  padding: 15px 32px;
  text-align: center;
  text-decoration: none;
  font-size: 16px;
  width: 100px;
  border-radius: 4px;
  transition-duration: 0.4s;
}
.button-on {
  background-color: #034078;
}
.button-on:hover {
  background-color: #1282A2;
}
.button-off {
  background-color: #858585;
}
.button-off:hover {
  background-color: #252524;
}

```

At the moment, you have the *index.html* and *style.css* files ready and saved in the *data* folder under the project folder. Now, let's set up the web server.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* configuration file for the ESP32 should be like this (same as the previous project).

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The *platformio.ini* configuration file for the ESP8266 should be like this (same as the previous project).

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps      = ESP Async WebServer
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Set LED GPIO
const int ledPin = 2;

// Stores LED state
String ledState;

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

// Replaces placeholder with LED state value
String processor(const String& var){
    if(var == "STATE") {
        if(digitalRead(ledPin)) {
            ledState = "ON";
        }
        else {
            ledState = "OFF";
        }
        return ledState;
    }
    return String();
}

void setup() {

```

```

// Serial port for debugging purposes
Serial.begin(115200);
initWiFi();
initSPIFFS();

// Set GPIO2 as an OUTPUT
pinMode(ledPin , OUTPUT);

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

server.serveStatic("/", SPIFFS, "/");

// Route to set GPIO state to HIGH
server.on("/on", HTTP_GET, [] (AsyncWebServerRequest *request){
    digitalWrite(ledPin, HIGH);
    request->send(SPIFFS, "/index.html", "text/html", false, processor);
});

// Route to set GPIO state to LOW
server.on("/off", HTTP_GET, [] (AsyncWebServerRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(SPIFFS, "/index.html", "text/html", false, processor);
});

// Start server
server.begin();
}

void loop() {
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Set LED GPIO
const int ledPin = 2;

// Stores LED state
String ledState;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

// Replaces placeholder with LED state value
String processor(const String& var) {
    if(var == "STATE") {
        if(digitalRead(ledPin)) {
            ledState = "OFF";
        }
        Else {
            ledState = "ON";
        }
        return ledState;
    }
    return String();
}

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

```

```

initWiFi();
initFS();

// Set GPIO2 as an OUTPUT
pinMode(ledPin, OUTPUT);

server.on("/", HTTP_GET, [](AsyncWebRequest *request){
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});

server.serveStatic("/", LittleFS, "/");

    // Route to set GPIO state to HIGH (inverted logic for ESP8266)
server.on("/on", HTTP_GET, [](AsyncWebRequest *request){
    digitalWrite(ledPin, LOW);
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});

    // Route to set GPIO state to LOW (inverted logic for ESP8266)
server.on("/off", HTTP_GET, [](AsyncWebRequest *request){
    digitalWrite(ledPin, HIGH);
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});
server.begin();
}

void loop() {
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How The Code Works

The code is very similar to the previous project but adds the lines to control the LED. Let's take a look at the relevant parts for this example.

Define LED Pin

Create a variable called `ledPin` that holds the GPIO number you want to control and a variable called `ledState` to hold the current GPIO state.

```

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
String ledState;

```

processor() function

The `processor()` function replaces the placeholders in the HTML text with the actual values.

```
String processor(const String& var) {
    if(var == "STATE"){
        if(digitalRead(ledPin)) {
            ledState = "ON";
        }
        else {
            ledState = "OFF";
        }
        return ledState;
    }
    return String();
}
```

When the web page is requested, the ESP checks if the HTML text has any placeholders. If it finds the `%STATE%` placeholder, read the current GPIO state with `digitalRead(ledPin)` and set the `ledState` variable accordingly. The function returns the current GPIO state as a string variable.

Note: the ESP8266 on-board LED works with inverted logic. It lights up when you send a LOW signal and turns off when you send a HIGH signal.

setup()

In the `setup()`, initialize the Serial Monitor, Wi-Fi and the filesystem. If you're using the ESP32, initialize SPIFFS. If you're using the ESP8266, initialize LittleFS.

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

Set GPIO 2 as an `OUTPUT`:

```
pinMode(ledPin , OUTPUT);
```

Handle Requests

The following lines handle what happens when you receive a request on the root (/) URL (ESP IP address).

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

When it receives that request, it sends the HTML text saved in the `index.html` file to build the web page. It also needs to pass the `processor` function, which replaces all the placeholders in the HTML text with the right values.

The first argument of the `send()` function is the filesystem where the files are saved, in this case it is saved in `SPIFFS` (or `LittleFS` for the ESP8266). The second argument is the path where the file is located. The third argument refers to the content type (HTML text). The fourth argument means `download=false`. Finally, the last argument is the `processor` function.

When the HTML file loads in your browser, it will make a request for the CSS file and the favicon. These are static files saved in the same directory (SPIFFS or LittleFS). So, we can simply add the following line to serve files in a directory when requested by the root URL. It serves the CSS and favicon files automatically.

```
server.serveStatic("/", SPIFFS, "/");
```

If you're using an ESP8266, it should be like this:

```
server.serveStatic("/", LittleFS, "/");
```

/on request

When you click on the ON button on the web page, you make a request on the /on URL. So, we need to handle what happens when the ESP receives such request.

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

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

Set the GPIO state to HIGH to light up the LED.

```
digitalWrite(ledPin, HIGH);
```

And send the HTML page again.

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

Because we've changed the LED state, we need to send the HTML file again with the processor function. In this case, the processor function will replace the %STATE% placeholder with ON.

/off request

In a similar way, when you click on the OFF button, you make a request on the /off URL. So, set the GPIO state to LOW and send the web page again to update the current GPIO state.

```
server.on("/off", HTTP_GET, [](AsyncWebServerRequest *request){  
    digitalWrite(ledPin, LOW);  
    request->send(SPIFFS, "/index.html", "text/html", false, processor);  
});
```

Finally, start the web server with the begin() method.

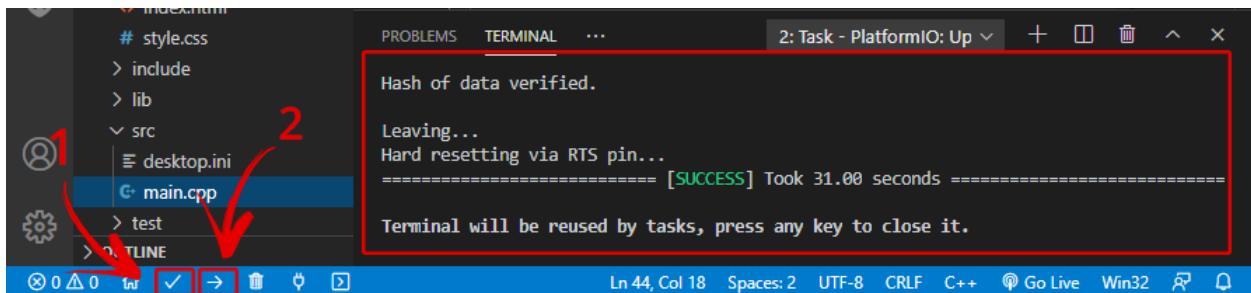
```
server.begin();
```

This is an asynchronous web server, so it is not necessary to add anything to the loop().

```
void loop() {}
```

Uploading Code

After modifying the code with your network credentials, save the code. Click the **Compile** icon and then on the **Upload** icon to upload code to your board.

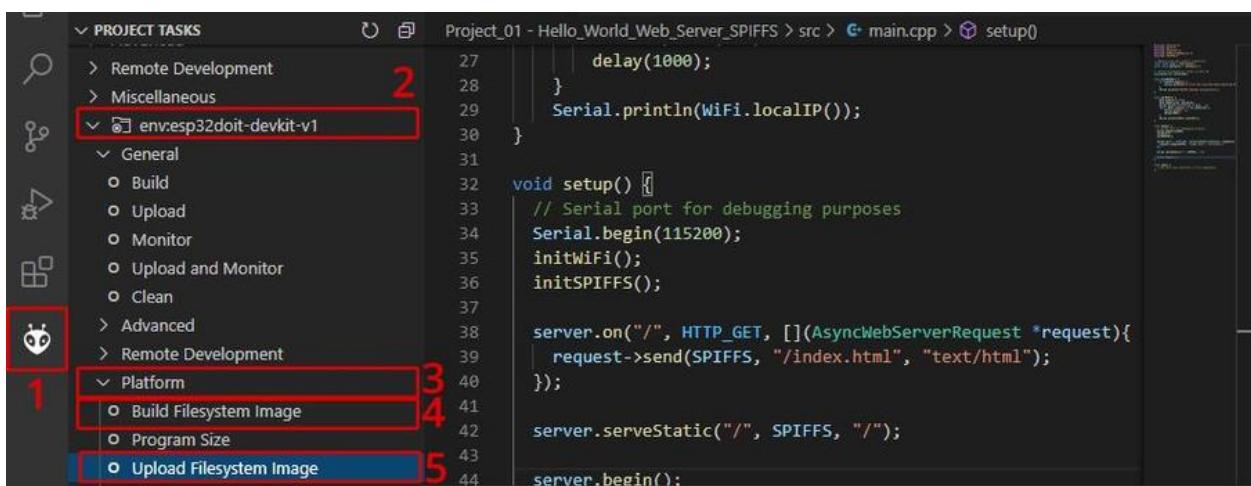


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

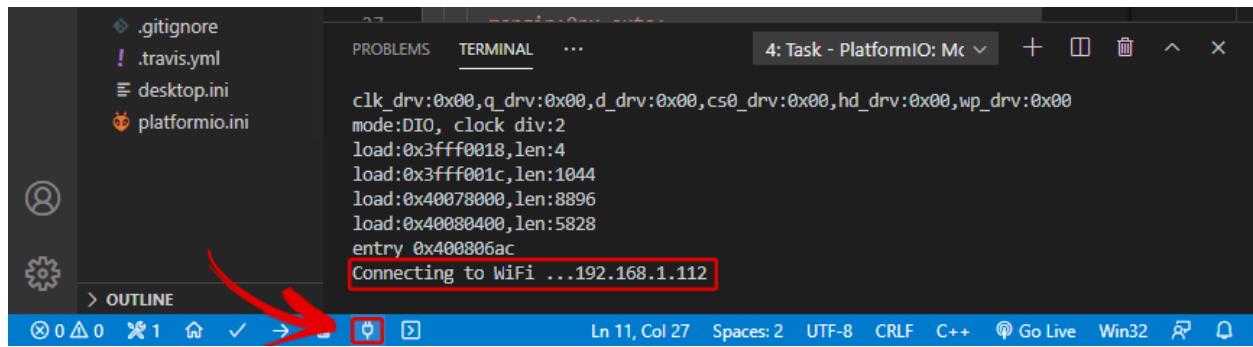
Finally, upload the files (*index.html*, *style.css* and *favicon.png*) to the filesystem:

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.



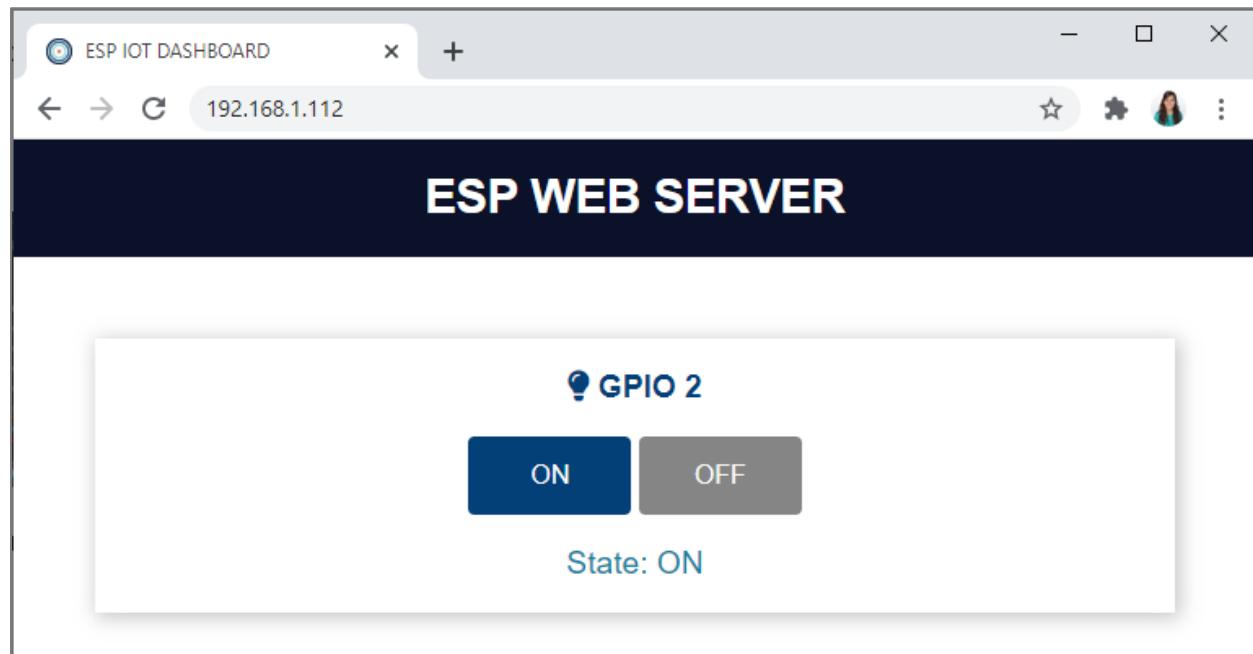
Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous example, it will probably have the same IP.



```
PROBLEMS TERMINAL ...
c1k_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5828
entry 0x400806ac
Connecting to WiFi ...192.168.1.112
```

Open a browser on your local network and you should be able to access your newly created web server.



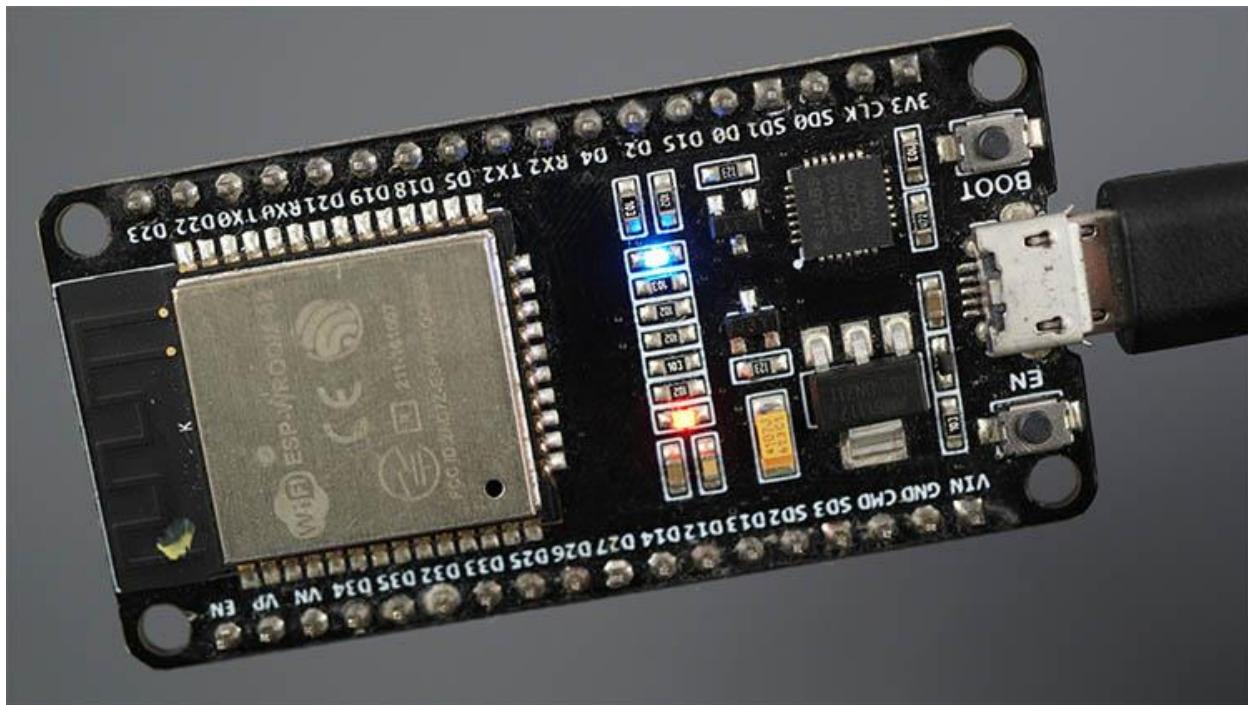
Notice that the buttons change color when you hover the mouse over them. For example, for the ON button:



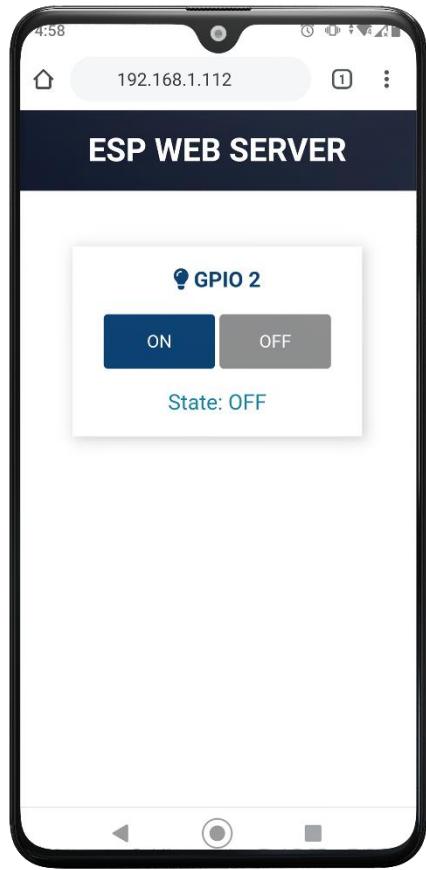
And for the OFF button:



Click on the buttons, and you should be able to control the ESP32 or ESP8266 on-board LED.



This is how the web server looks like on a smartphone.



Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this project, you've learned how to build your first web server to control the ESP32 and ESP8266 boards' outputs. As an example, we've controlled the on-board LED. The idea is to modify this project to control any other GPIOs.

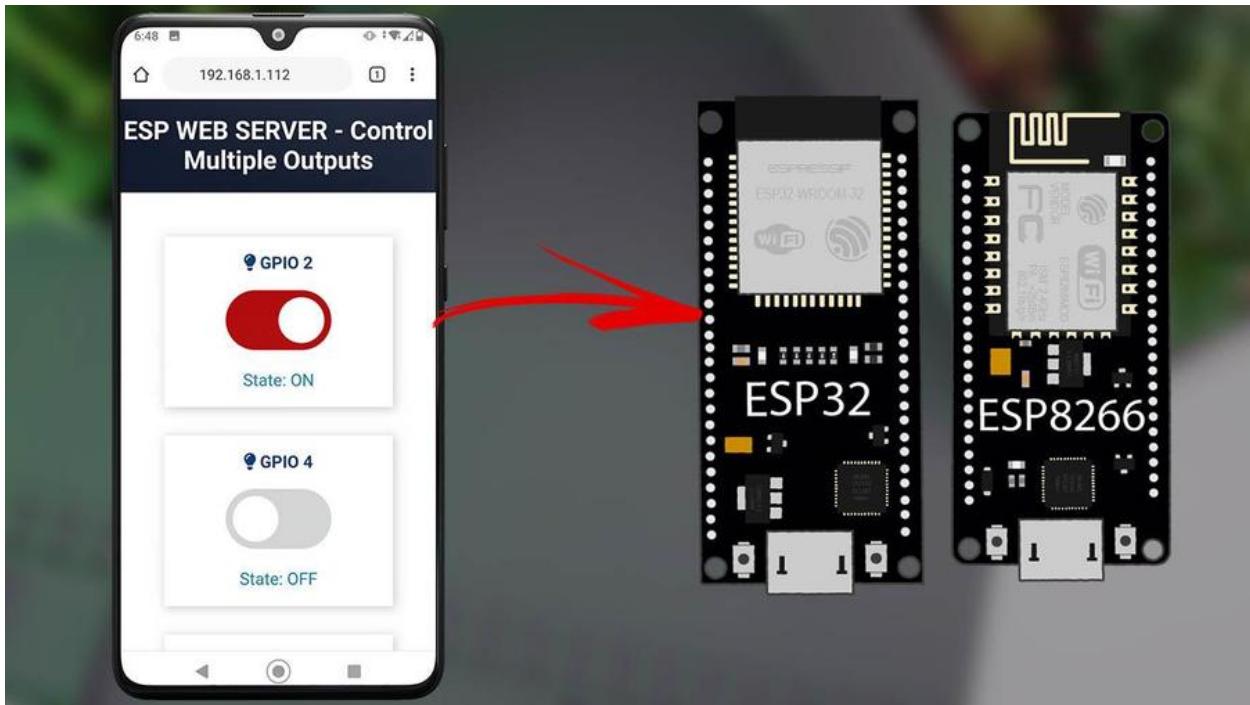
You can easily duplicate the card in the HTML document several times to display multiple cards to control different outputs. Each button needs to request a different

URL path so that you know which GPIO sent the request. Additionally, you need a different placeholder for each GPIO.

We recommend that you wire some LEDs to other GPIOs on your board and control them with the web server using the concepts learned.

As you'll see later, there are other ways to control multiple GPIOs without having to create numerous URL paths. You'll learn that in one of the subsequent Units. However, this is one of the easiest ways to achieve that, and it is also a good exercise.

2.2 - Web Server - Control Multiple Outputs (Toggle Switches)



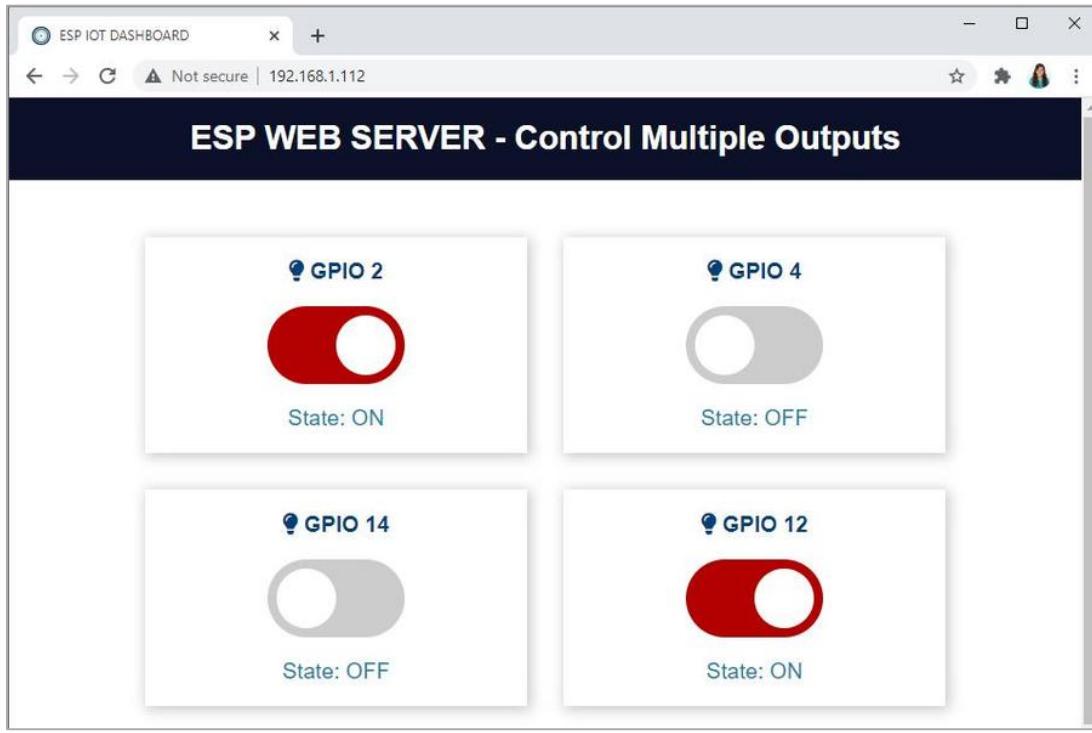
This Unit shows how to create a web page with toggle switches to control multiple ESP32 or ESP8266 outputs.

Project Overview

The web server we'll build in this Unit serves a web page that displays:

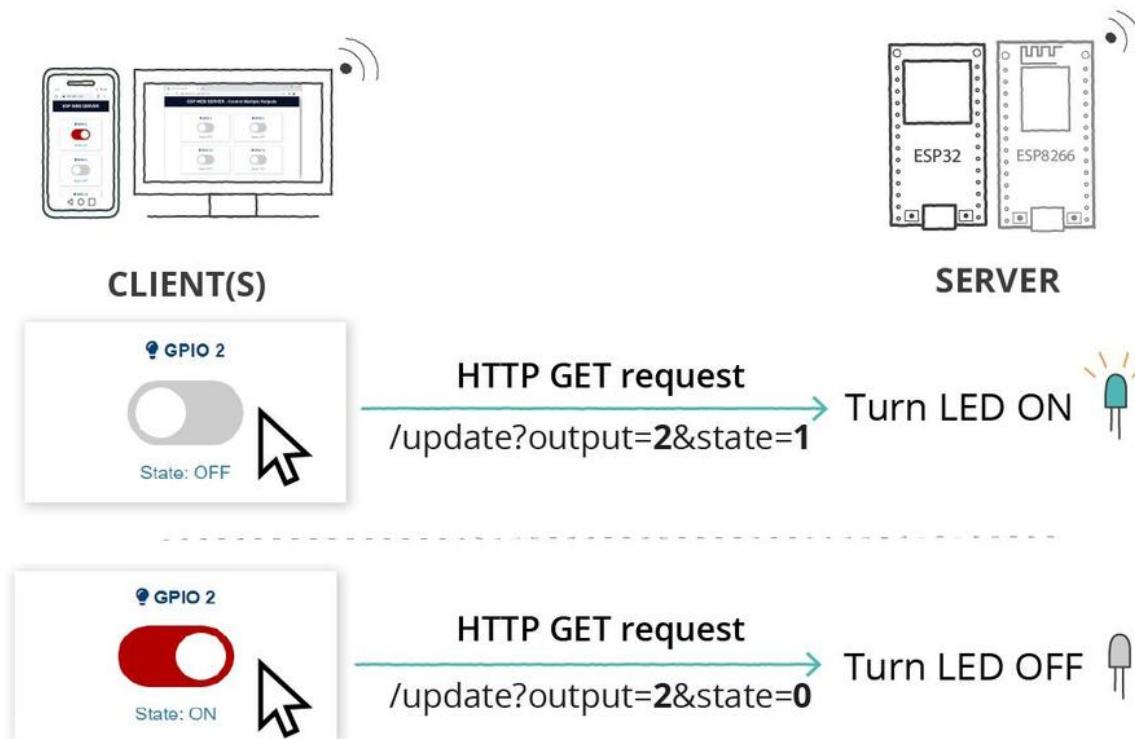
- One heading with “ESP WEB SERVER – Control Multiple Outputs” text.
- Four toggle switches to control four GPIOs;
- Each toggle switch has a label indicating the GPIO output pin. You can easily remove/add GPIOs. There’s also a label indicating the current GPIO state.
- The toggle switches change color accordingly to the GPIO state.

Here's a preview of the web page.



How it Works?

Let's see what happens when you toggle the buttons. Here's an example for GPIO 2. It works similarly to the other buttons.



In the first scenario, you toggle the button to turn GPIO 2 on. When that happens, the browser makes an HTTP GET request on the `/update?output=2&state=1` URL. Based on that URL, the ESP changes the state of GPIO 2 to `HIGH` (1) and turns the LED on. The GPIO state is updated on the web browser using JavaScript.

In the second example, you toggle the button to turn GPIO 2 off. When that happens, the browser makes an HTTP GET request on the `/update?output=2&state=0` URL. Based on that URL, we change the state of GPIO 2 to `LOW` (0) and turn the LED off.

Building the Circuit

In this example, we'll control multiple ESP32/ESP8266 GPIOs. As an example, we'll control four LEDs attached to the following GPIOs: 2, 4, 12, and 14.

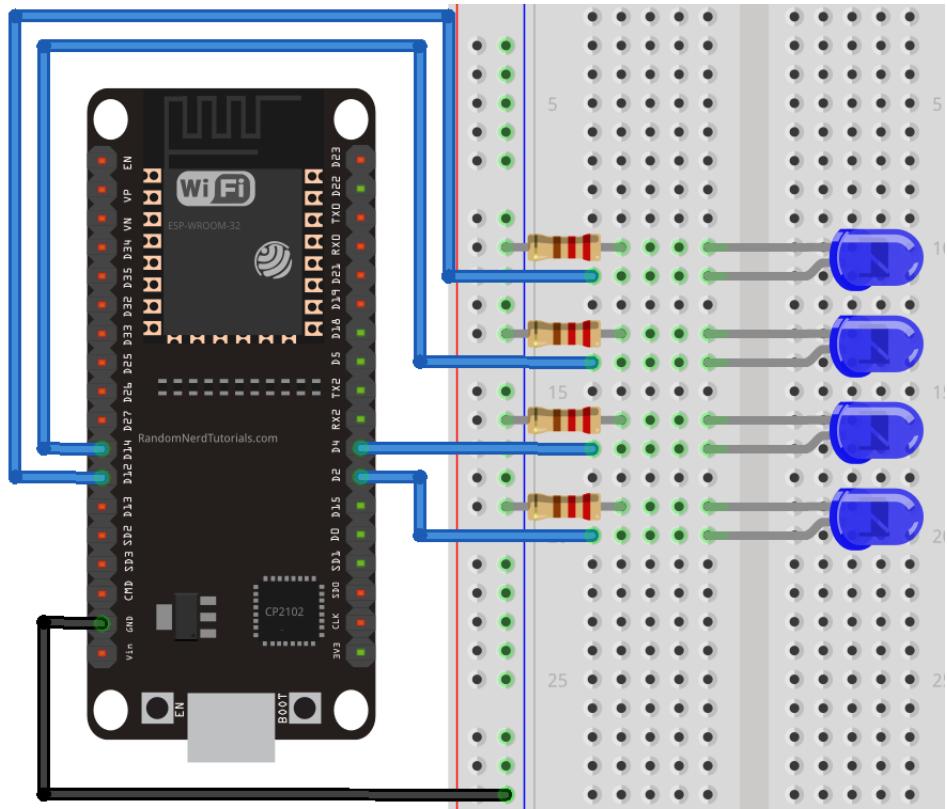
Parts Required:

Here are the parts needed to build the circuit:

- [ESP32](#) or [ESP8266](#) board
- 4x [LEDs](#)
- 4x [220 Ω resistors](#)
- [Breadboard](#)
- [Jumper wires](#)

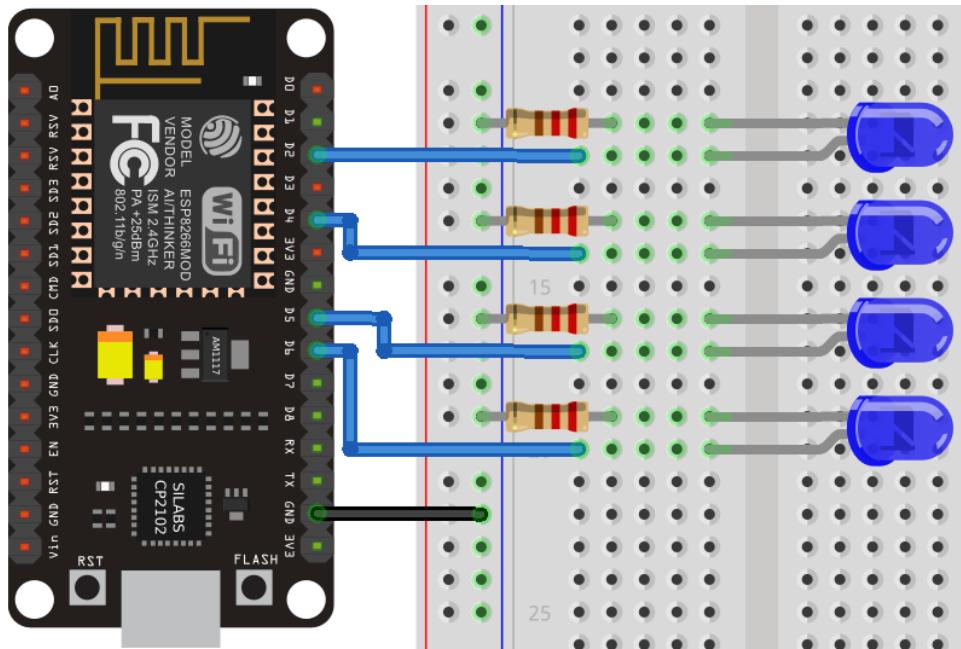
Assemble a circuit as shown in the following schematic diagrams. Choose the correct diagram for the board you're using.

ESP32 – Schematic Diagram



Connect to GPIOs: 2, 4, 12, and 14.

ESP8266 – Schematic Diagram



In the ESP8266, the GPIOs are labeled on the silkscreen like this:

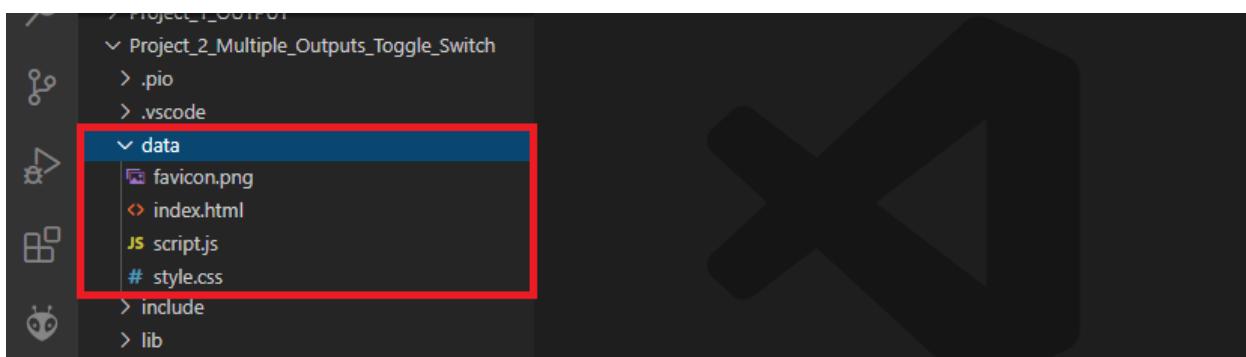
- GPIO 4 → D2
- GPIO 2 → D4
- GPIO 14 → D5
- GPIO 12 → D6

Building the Web Page

Start by creating a new project in VS Code as you did in the previous Units.

To build the web page for this project, make sure you put all the following files inside a *data* folder under your project folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*



We'll use the same template from the previous project for the web page, but instead of placing ON and OFF buttons inside the card, we'll place toggle switches. Additionally, we'll create several cards to control multiple GPIOs.

Because we've already explained the HTML to build the web page in great detail, we'll just take a look at the relevant parts for this project.

Toggle Switch

Although we'll need to create toggle switches, we'll do this by creating checkboxes (as in the example below for GPIO 2) and using CSS, later on, to make them look like switches.

```
<label class="switch">
  <input type="checkbox" onchange="toggleCheckbox(this)" id="2">
  <span class="slider"></span>
</label>
```

Let's break down the previous HTML text into simple chunks. In HTML, a toggle switch is created like this:

```
<label class="switch">
  <input type="checkbox">
  <span class="slider"></span>
</label>
```

The `<input>` tag specifies an input field where the user can enter data. The toggle switch is an input field of type `checkbox`. There are many other input field types as we've seen previously.

```
<input type="checkbox">
```

Initially, the checkbox can be checked or not. If you want it to be checked, you have something like this:

```
<input type="checkbox" checked>
```

If you want it to be unchecked initially, it is simply like this:

```
<input type="checkbox">
```

onchange event

Whenever we click on the slider, we want to send a request to the ESP32/ESP8266 with the new GPIO state. To make things happen when something changes, we can use HTML events. We use the `onchange` event. This is an event attribute that occurs

when we change the element's value (the checkbox). Whenever you check or uncheck the toggle switch, it calls the `toggleCheckbox()` JavaScript function and passes as argument `this`. The `this` keyword corresponds to the checkbox element.

```
onchange="toggleCheckbox(this)"
```

The `id` attribute specifies a unique id for an HTML element. The `id` allows us to manipulate the element using JavaScript or CSS. We'll give a different `id` to each card to know which button was toggled (see the JavaScript file that we'll create later on). The `id` for the toggle switch that controls GPIO 2 will be `2`. So, we define the input type like this:

```
<input type="checkbox" onchange="toggleCheckbox(this)" id="2">
```

Here's the `id` for the other toggle switches:

- Toggle switch for GPIO 4 → `id = "4"`
- Toggle switch for GPIO 12 → `id = "12"`
- Toggle switch for GPIO 14 → `id = "14"`

The `id` for each toggle switch corresponds to the GPIO it controls – this makes our life easier, as you'll see later on.

As in the previous project, the current GPIO state is displayed on the card using a paragraph `<p>` tag. However, this time, we won't use the processor function to update the state. We'll use a response sent by the server (ESP). The response is handled by a JavaScript function. So, the text for the state should have a specific `id`. For simplicity, we'll use the following `ids` (GPIO number followed by an "s"):

- State id for GPIO 2 → `id="2s"`
- State id for GPIO 2 → `id="4s"`
- State id for GPIO 2 → `id="12s"`
- State id for GPIO 2 → `id="14s"`

The card for GPIO 2 will be like this (for the other GPIOs, use different ids).

```
<div class="card">
  <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
  <label class="switch">
    <input type="checkbox" onchange="toggleCheckbox(this)" id="2">
    <span class="slider"></span>
  </label>
  <p class="state">State: <span id="2s"></span></p>
</div>
```

Referencing a JavaScript File

As we've mentioned previously, we'll create some JavaScript functions to send requests to the ESP and handle its responses. You can either include the JavaScript in the body of the HTML document, between the `<style></style>` tags, or you can place it in a separate `.js` file and reference it.

We'll use a separate `.js` file that we reference in the HTML document. The file is called `script.js` and it is in the same folder as the HTML file. So, you reference it like this (before the `</body>` closing tag).

```
<script src="script.js"></script>
```

HTML File

Here's the complete HTML file for this project.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="style.css">
  <link rel="icon" type="image/png" href="favicon.png">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER - Control Multiple Outputs</h1>
  </div>
```

```

<div class="content">
  <div class="card-grid">
    <div class="card">
      <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
      <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="2">
        <span class="slider"></span>
      </label>
      <p class="state">State: <span id="2s"></span></p>
    </div>
    <div class="card">
      <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 4</p>
      <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="4">
        <span class="slider"></span>
      </label>
      <p class="state">State: <span id="4s"></span></p>
    </div>
    <div class="card">
      <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 14</p>
      <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="14">
        <span class="slider"></span>
      </label>
      <p class="state">State: <span id="14s"></span></p>
    </div>
    <div class="card">
      <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 12</p>
      <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="12">
        <span class="slider"></span>
      </label>
      <p class="state">State: <span id="12s"></span></p>
    </div>
  </div>
  <script src="script.js"></script>
</body>
</html>

```

CSS File

We'll use the same styles as the previous project. We just need to add the following styles for the toggle switch.

```

.switch {
  position: relative;
  display: inline-block;
  width: 120px;
  height: 68px
}

```

```
.switch input {
  display: none
}
.slider {
  position: absolute;
  top: 0; left: 0; right: 0; bottom: 0;
  background-color: #ccc;
  border-radius: 50px
}
.slider:before {
  position: absolute;
  content: "";
  height: 52px;
  width: 52px;
  left: 8px;
  bottom: 8px;
  background-color: #fff;
  transition: .4s;
  border-radius: 50px;
}
input:checked+.slider {
  background-color: #b30000;
}
input:checked+.slider:before {
  -webkit-transform: translateX(52px);
  -ms-transform: translateX(52px);
  transform: translateX(52px);
}
```

The `.switch` selector refers to the box around the slider. You can change the `width` and `height` properties of the slider to adjust its size. The defined values work well for the switch we created. The `position` and `display` properties must be set to `relative` and `inline-block`.

```
.switch {
  position: relative;
  display: inline-block;
  width: 120px;
  height: 68px
}
```

The following lines are needed to hide the default HTML checkbox.

```
.switch input {
  display: none
}
```

The `.slider` selector styles the slider itself. We set the `border-radius` property to get a slider with round borders. You can also change the `background-color` property. In this case, it's a shade of gray. This is the background color when it is not enabled.

```
.slider {  
  position: absolute;  
  top: 0; left: 0; right: 0; bottom: 0;  
  background-color: #ccc;  
  border-radius: 50px}
```

The `:before` creates a pseudo element that is the first child of the selected element (`slider`). This creates and styles the circle that moves on the slider. We set the following properties.

```
.slider:before {  
  position: absolute;  
  content: "";  
  height: 52px;  
  width: 52px;  
  left: 8px;  
  bottom: 8px;  
  background-color: #fff;  
  transition: .4s;  
  border-radius: 50px;  
}
```

The properties' names are pretty self-explanatory. They set up width, height (equal width and height with a border radius create a circle), set left and bottom borders, border radius, and the transition time for the animation (when you click the switch).

Set the background color when the switch is toggled. In our case, it's a shade of red. You can set any other color.

```
input:checked+.slider {  
  background-color: #b30000;  
}
```

The `+` signal in CSS is used to select the elements placed immediately after the specified element but not inside the particular elements. This way, we can style the

background. The `:checked` pseudoelement styles the background with that color when the toggle switch (checkbox) is checked.

Finally, apply the following properties to make the switch move when you check the slider.

```
input:checked+.slider:before {  
  -webkit-transform: translateX(52px);  
  -ms-transform: translateX(52px);  
  transform: translateX(52px);  
}
```

Vendor Prefixes

Vendor prefixes allow a browser to support new CSS features before they become fully supported. The most commonly used browsers use the following prefixes:

- webkit- Chrome, Safari, newer versions of Opera, almost all iOS browsers,
- moz- Firefox,
- o- Old versions of Opera,
- ms- Microsoft Edge and Internet Explorer.

Vendor prefixes are temporary. Once the properties are fully supported by the browser you use, you don't need them.

You can use the following reference to check if the property you're using needs prefixes: <http://shouldiprefix.com/>

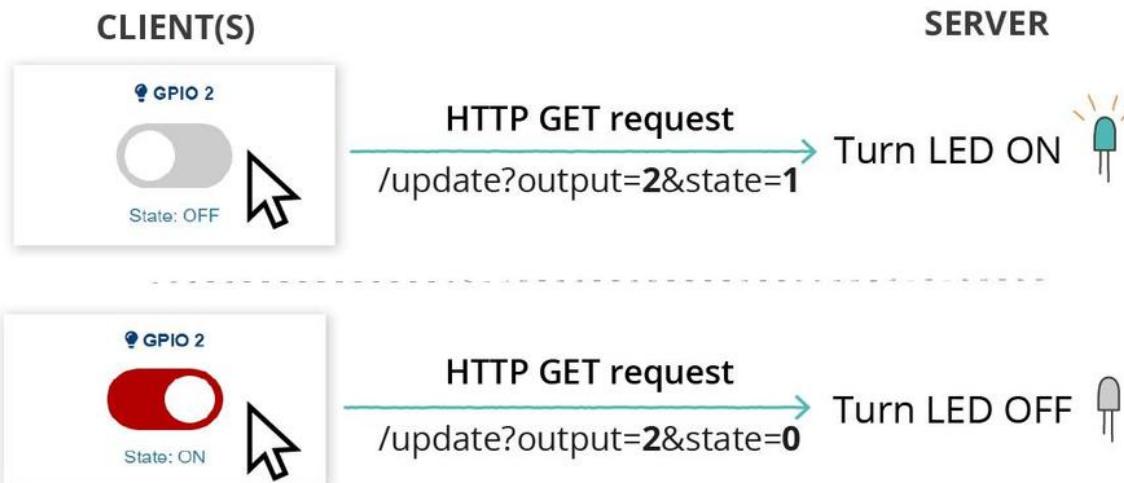
At the time of writing this eBook, it is recommended to use prefixes with the `transform` property.

JavaScript File

We want to send requests to the ESP32 or ESP8266 when we click on the toggle switches. As we've seen previously, we'll call a JavaScript function named `toggleCheckBox()` that accepts as argument the `id` of the checkbox.

Sending a Request to the Server (ESP32/ESP8266) using JavaScript

Take a look at the following diagram to remember the requests you need to make to the server.



Here's the JavaScript function that does that (`toggleCheckbox()`).

```
// Send Requests to Control GPIOs
function toggleCheckbox (element) {
  var xhr = new XMLHttpRequest();
  if (element.checked){
    xhr.open("GET", "/update?output="+element.id+"&state=1", true);
    document.getElementById(element.id+"s").innerHTML = "ON";
  }
  else {
    xhr.open("GET", "/update?output="+element.id+"&state=0", true);
    document.getElementById(element.id+"s").innerHTML = "OFF";
  }
  xhr.send();
}
```

To create a function in JavaScript, use the `function` keyword followed by the function name and parentheses. Inside the parentheses we pass any parameters needed.

```
function toggleCheckbox (element) {
```

Then, create a new variable called `xhr` that corresponds to a `XMLHttpRequest` object.

To create a new variable use the `var` keyword.

```
var xhr = new XMLHttpRequest();
```

The XMLHttpRequest object can be used to exchange data with a web server behind the scenes. This allows us to update parts of a web page without reloading the whole page. To send a request to a server, use the `open()` and `send()` methods of the XMLHttpRequest object. Here are the arguments accepted by each method:

`open(method, url, async)`

- `method`: the type of request: GET or POST
- `url`: the request URL
- `async`: `true` (asynchronous) or `false` (synchronous)

`send()`

You don't need to pass any arguments to the `send()` method if you're using a GET request.

Should you use GET or a POST request? Making a GET request is simpler and faster than POST, and it can be used in most cases. In our examples, we'll use GET requests. For a more in-depth explanation about GET and POST, we recommend reading the following tutorials:

- [ESP32 HTTP GET and HTTP POST with Arduino IDE \(JSON, URL Encoded, Text\)](#)
- [ESP8266 HTTP GET and HTTP POST with Arduino IDE \(JSON, URL Encoded, Text\)](#)

We'll make a different request depending on whether the checkbox is checked or not.

If it is checked (`element.checked`), we send the request with `state=1`.

```
if (element.checked) {  
    xhr.open("GET", "/update?output="+element.id+"&state=1", true);
```

We also update the corresponding state on the card to `ON`. The element that displays the state has the checkbox id followed by `s`. So, in case of GPIO 2, it is "2s".

```
document.getElementById(element.id+"s").innerHTML = "ON";
```

If the element is not checked, we make a similar request, but sending `state=0` and set the state to OFF.

```
else {
  xhr.open("GET", "/update?output=" + element.id + "&state=0", true);
  document.getElementById(element.id + "s").innerHTML = "OFF";
}
```

Finally, send the request to the server using the `send()` method:

```
xhr.send();
```

Getting Current GPIO States using JavaScript

Every time you access the web page, you want it to display the current GPIO states. In particular, when the page loads for the first time, we'll request the server (ESP32/ESP8266) to get a response with all GPIO states and put them in the corresponding place on the web page.

We can add an event listener to the window like this:

```
// Get current GPIO states when the page loads
window.addEventListener('load', getStates);
```

The `window` object represents an open window in a browser. The `addEventListener()` method sets up a function to be called when a certain event happens. In this case, we'll call the `getStates` function when the page loads ('load') to get the GPIO states.

Now, let's take a look at the `getStates()` function. You already know how to make requests using JavaScript. Create a new `XMLHttpRequest` object. Then, send a GET request to the server on the `/states` URL using the `open()` and `send()` methods.

```
function getStates() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "/states", true);
  xhr.send();
}
```

When we make that request, the ESP responds with the required information. So, we need to handle what happens when we receive the response. We'll use the `onreadystatechange` property that defines a function to be executed when the `readyState` property changes. The `readyState` property holds the status of the `XMLHttpRequest`. The response of the request is ready when the `readyState` is 4 and the `status` is 200.

- `readyState = 4` means that the request finished and the response is ready;
- `status = 200` means "OK"

So, the request should look something like this:

```
function getStates(){
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      ... DO WHATEVER YOU WANT WITH THE RESPONSE ...
    }
  };
  xhr.open("GET", "/states", true);
  xhr.send();
}
```

The response sent by the ESP is the following text in JSON format

```
{
  "gpios": [
    {
      "output": "2",
      "state": "0"
    },
    {
      "output": "4",
      "state": "0"
    },
    {
      "output": "12",
      "state": "0"
    },
    {
      "output": "14",
      "state": "0"
    }
  ]
}
```

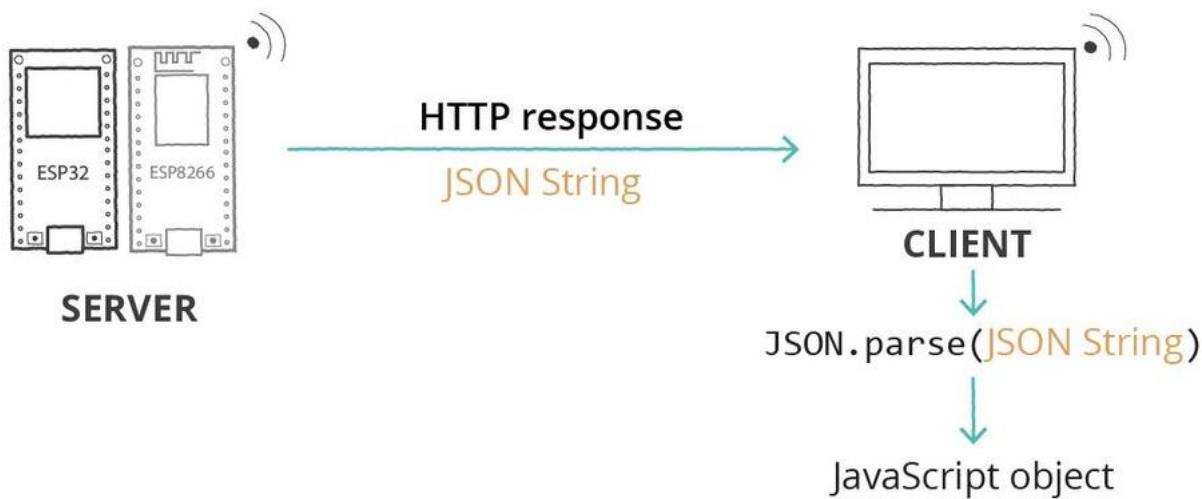
Understanding JSON Syntax

JSON stands for **J**avaScript **O**bject **N**otation, and it is a standard for storing and exchanging data in a way that's convenient for servers and clients. In JSON syntax:

- Data is represented in name/value pairs;
- Each name is followed by a colon (:) ;
- Name/value pairs are separated with commas;
- Curly brackets hold objects;
- Square brackets hold arrays.

In this particular example, we have a JSON object with the name `gpios` and its slightly complex value is the array of 4 items found between the [and]. Each of those 4 items is a JSON object made up of 2 name/value pairs: one pair holds the GPIO number, and the other holds its state."

JSON is text, and any JavaScript object can be converted into JSON text and sent to the server. Additionally, the server can send JSON text to the client that can be converted into a JavaScript object. JavaScript has a built-in function to convert a string, written in JSON format, into native JavaScript objects: `JSON.parse()`.



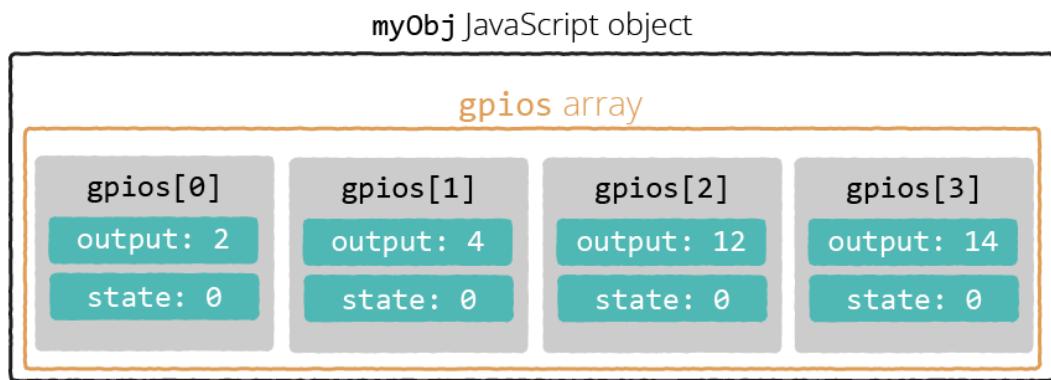
You can get the server's response as a JavaScript string using the `responseText` property. The response comes in JSON format, so we can save the response in a JSON object using the `JSON.parse()` method like this:

```
var myObj = JSON.parse(this.responseText);
```

For debugging purposes, you can display the value of the `myObj` JSON variable into the console using `console.log()`.

```
console.log(myObj);
```

Now, we need a `for` loop to go through all the outputs and corresponding states.



Take a look at the previous diagram to better understand the structure of the `myObj` object.

Here's how to access each output and corresponding state:

- `myObj.gpios[0].output` returns 2
- `myObj.gpios[0].state` returns 0, the state for GPIO 2
- `myObj.gpios[1].output` returns 4
- `myObj.gpios[1].state` returns 0, the state for GPIO 4
- `myObj.gpios[2].state` returns 12
- `myObj.gpios[2].state` returns 0, the state for GPIO 12
- `myObj.gpios[3].output` returns 14
- `myObj.gpios[3].state` returns 0, the state for GPIO 14

The following `for` loop goes through all objects inside the `gpios` array , gets the GPIOs and corresponding states and saves them in the `output` and `state` JavaScript variables.

```
for (i in myObj.gpios) {  
    var output = myObj.gpios[i].output;  
    var state = myObj.gpios[i].state;  
    console.log(output);  
    console.log(state);
```

Let's examine what happens in the first loop, `i=0`. The `output` variable is 2 and the `state` will be whatever the current state is. In this case, it is 0.

Now, we need to find the element with the `id="2`" and update the corresponding state either to ON or OFF and set the slider to `checked` or not. Like this:

```
for (i in myObj.gpios) {  
    var output = myObj.gpios[i].output;  
    var state = myObj.gpios[i].state;  
    console.log(output);  
    console.log(state);  
    if (state == "1") {  
        document.getElementById(output).checked = true;  
        document.getElementById(output+s).innerHTML = "ON";  
    }  
    else {  
        document.getElementById(output).checked = false;  
        document.getElementById(output+s).innerHTML = "OFF";  
    }  
}
```

If the state is "1", we get the element with `id="2`" (output) and set it to checked, to check the checkbox.

```
document.getElementById(output).checked = true;
```

We also need to update the state text to ON. Get the element with the `id="2s"` (`output+s`) and update the text to ON.

```
document.getElementById(output+s).innerHTML = "ON";
```

A similar process is done when the state is "0".

```
else {
    document.getElementById(output).checked = false;
    document.getElementById(output+s).innerHTML = "OFF";
}
```

Now, the `getStates()` function is complete.

```
function getStates() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            for (i in myObj.gpios) {
                var output = myObj.gpios[i].output;
                var state = myObj.gpios[i].state;
                console.log(output);
                console.log(state);
                if (state == "1") {
                    document.getElementById(output).checked = true;
                    document.getElementById(output+s).innerHTML = "ON";
                }
                else {
                    document.getElementById(output).checked = false;
                    document.getElementById(output+s).innerHTML = "OFF";
                }
            }
        }
    };
    xhr.open("GET", "/states", true);
    xhr.send();
}
```

script.js file

Here's the complete `script.js` file.

```
window.addEventListener('load', getStates);

// Function to get and update GPIO states on the web page
function getStates(){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            for (i in myObj.gpios){
                var output = myObj.gpios[i].output;
                var state = myObj.gpios[i].state;
                console.log(output);
                console.log(state);
                if (state == "1") {
```

```

        document.getElementById(output).checked = true;
        document.getElementById(output+s).innerHTML = "ON";
    }
    else {
        document.getElementById(output).checked = false;
        document.getElementById(output+s).innerHTML = "OFF";
    }
}
};

xhr.open("GET", "/states", true);
xhr.send();
}

// Send Requests to Control GPIOs
function toggleCheckbox (element) {
    var xhr = new XMLHttpRequest();
    if (element.checked) {
        xhr.open("GET", "/update?output="+element.id+"&state=1", true);
        document.getElementById(element.id+s).innerHTML = "ON";
    }
    else {
        xhr.open("GET", "/update?output="+element.id+"&state=0", true);
        document.getElementById(element.id+s).innerHTML = "OFF";
    }
    xhr.send();
}

```

Setting Up the Web Server

Follow the following steps to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* configuration file for the ESP32 should be like this. Notice that we need to add the `Arduino_JSON` library.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = Arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
            arduino-libraries/Arduino_JSON @ 0.1.0
```

platformio.ini file (ESP8266)

The *platformio.ini* configuration file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

//Input Parameters
const char* PARAM_INPUT_OUTPUT = "output";
const char* PARAM_INPUT_STATE = "state";

// Set number of outputs
#define NUM_OUTPUTS 4

// Array with outputs you want to control
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
```

```

        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}
// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}
// Return JSON with Current Output States
String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
        //myArray["gpios"][i] = "{\"output\":\"" + String(outputGPIOs[i]) + "\", \"state\": \"\" + String(digitalRead(outputGPIOs[i])) + "\"}";
    }
    String jsonString = JSON.stringify(myArray);
    Serial.print(jsonString);
    return jsonString;
}
void setup(){
    // Serial port for debugging purposes
    Serial.begin(115200);

    // Set GPIOs as outputs
    for (int i =0; i<NUM_OUTPUTS; i++){
        pinMode(outputGPIOs[i], OUTPUT);
    }

    initWiFi();
    initSPIFFS();

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/index.html", "text/html", false);
    });

    server.serveStatic("/", SPIFFS, "/");
}

server.on("/states", HTTP_GET, [] (AsyncWebServerRequest *request){
    String json = getOutputStates();
    request->send(200, "application/json", json);
    json = String();
});

//GET request to <ESP_IP>/update?output=<output>&state=<state>

```

```

server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String output;
    String state;
    // GET input1 value on <ESP_IP>/update?output=<output>&state=<state>
    if (request->hasParam(PARAM_INPUT_OUTPUT) && request-
>hasParam(PARAM_INPUT_STATE)) {
        output = request->getParam(PARAM_INPUT_OUTPUT)->value();
        state = request->getParam(PARAM_INPUT_STATE)->value();
        // Control GPIO
        digitalWrite(output.toInt(), state.toInt());
    }
    else {
        output = "No message sent";
        state = "No message sent";
    }
    Serial.print("GPIO: ");
    Serial.print(output);
    Serial.print(" - Set to: ");
    Serial.println(state);

    request->send(200, "text/plain", "OK");
});
// Start server
server.begin();
}
void loop() {
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file under the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>
#include <Arduino_JSON.h>

// Replace with your network credentials

```

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

//Input Parameters
const char* PARAM_INPUT_OUTPUT = "output";
const char* PARAM_INPUT_STATE = "state";

// Set number of outputs
#define NUM_OUTPUTS 4
// Array with outputs you want to control
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

// Return JSON with Current Output States
String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
        //myArray["gpios"][i] = "{\"output\":\"" + String(outputGPIOs[i]) + "\", \"state\": \"\" + String(digitalRead(outputGPIOs[i])) + "\"}";
    }
    String jsonString = JSON.stringify(myArray);
    Serial.print(jsonString);
    return jsonString;
}

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

    // Set GPIOs as outputs
}

```

```

for (int i =0; i<NUM_OUTPUTS; i++){
    pinMode(outputGPIOs[i], OUTPUT);
}

initWiFi();
initFS();

// Route for root / web page
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/index.html", "text/html", false);
});

server.serveStatic("/", LittleFS, "/");

server.on("/states", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getOutputStates();
    request->send(200, "application/json", json);
    json = String();
});

//GET request to <ESP_IP>/update?output=<output>&state=<state>
server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String output;
    String state;
    // GET input1 value on <ESP_IP>/update?output=<output>&state=<state>
    if (request->hasParam(PARAM_INPUT_OUTPUT) && request-
>hasParam(PARAM_INPUT_STATE)) {
        output = request->getParam(PARAM_INPUT_OUTPUT)->value();
        state = request->getParam(PARAM_INPUT_STATE)->value();
        // Control GPIO
        digitalWrite(output.toInt(), state.toInt());
    }
    else {
        output = "No message sent";
        state = "No message sent";
    }
    Serial.print("GPIO: ");
    Serial.print(output);
    Serial.print(" - Set to: ");
    Serial.println(state);

    request->send(200, "text/plain", "OK");
});
// Start server
server.begin();
}
void loop() {
}

```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

How The Code Works

You should already be familiar with most lines of code from previous projects. We'll just take a look at the relevant parts for this project.

JSON Library

As mentioned previously, we'll send a JSON string as a response to the client with the current GPIO states. To make it easier to handle JSON strings, we'll use the `Arduino_JSON.h` library.

```
#include <Arduino_JSON.h>
```

Network Credentials

Don't forget to include your network credentials in the following variables:

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

Input Parameters

When you click on the toggle buttons on the web page, it makes a request with the following format:

```
<ESP_IP>/update?output=<output>&state=<state>
```

So, we'll search in the URL, for the parameters `output` and `state`.

```
const char* PARAM_INPUT_OUTPUT = "output";
const char* PARAM_INPUT_STATE = "state";
```

Set up Outputs

The code is prepared to control GPIOs 2, 4, 12 and 14. You can modify the `NUM_OUTPUTS` and `outputGPIOs` variables to change the number of GPIOs and which ones you want to control.

The `NUM_OUTPUTS` variable defines the number of GPIOs. The `outputGPIOs` is an array with the GPIO numbers you want to control.

```
// Set number of outputs
#define NUM_OUTPUTS 4

// Array with outputs you want to control
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};
```

Note: if you change the number of GPIOs and the GPIOs you want to control, you must also change the ids of the HTML elements in your `index.html` document.

JSON String with Current Output States

The `getOutputStates()` function checks the state of all your GPIOs and returns a JSON string variable with that information.

```
// Return JSON with Current Output States
String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    Serial.print(jsonString);
    return jsonString;
}
```

First, create a variable of type json called `myArray` using the `JSONVar` keyword. Then, add content to your array variable. As we've seen previously, the JSON variable contains a `gpios` array with the following structure:

gpios array

gpios[0]	gpios[1]	gpios[2]	gpios[3]
output: 2	output: 4	output: 12	output: 14
state: 0	state: 0	state: 0	state: 0

That's what we do in these lines:

```
for (int i =0; i<NUM_OUTPUTS; i++){
    myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
    myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
}
```

As we've explained previously, clients and servers can exchange text in JSON format. To convert the array into text in JSON format, use the `stringify()` function provided by the JSON library, like this:

```
String jsonString = JSON.stringify(myArray);
```

Finally, return the JSON string variable.

```
return jsonString;
```

Set GPIOs as Outputs

In the `setup()`, set all your GPIOs as outputs. This will automatically get all GPIOs in the `outputGPIOs` array variable and set them as outputs.

```
for (int i =0; i<NUM_OUTPUTS; i++){
    pinMode(outputGPIOs[i], OUTPUT);
}
```

Handle Requests

The following lines handle what happens when you receive a request on the root (/) URL (ESP IP address).

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

When it receives that request, it sends the HTML text saved in the `index.html` file to build the web page. This time, we won't use the `processor` function because the states are sent as a response to the `/states` request. We're using SPIFFS for the ESP32 and LittleFS for the ESP8266.

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

When the HTML file loads in your browser, it will make a request for the CSS, JavaScript, and favicon files. These are static files saved in the same directory (SPIFFS or LittleFS). So, we can simply add the following line to serve files in a directory when requested by the root URL. It will serve the CSS, JavaScript, and favicon files automatically.

```
server.serveStatic("/", SPIFFS, "/");
```

/states

When you first access the web page, it makes a request on the `/states` URL to get the current GPIO states. This is how you send the JSON string as a response:

```
server.on("/states", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getOutputStates();
    request->send(200, "application/json", json);
    json = String();
});
```

First, you get the current GPIO states as a JSON string by calling the `getOutputStates()` function created previously and save it in the `json` variable.

```
String json = getOutputStates();
```

Then, use the following arguments on the `send()` method:

```
request->send(200, "application/json", json);
```

The first argument is the response code. The HTTP 200 OK success status response code indicates that the request has succeeded. The second argument is the content type. In our case, we're sending JSON ("application/json"). And finally, the content we want to send (`json` variable).

/update

As we've seen previously, you receive a request like this when you press the toggle switches:

```
<ESP_IP>/update?output=<output>&state=<state>
```

We check if the request contains the `PARAM_INPUT_OUTPUT` variable value (`output`) and the `PARAM_INPUT_STATE` variable (`state`) and save the corresponding values in the `output` and `state` variables.

```
if (request->hasParam(PARAM_INPUT_OUTPUT) && request->hasParam(PARAM_INPUT_STATE)){
    output = request->getParam(PARAM_INPUT_OUTPUT)->value();
    state = request->getParam(PARAM_INPUT_STATE)->value();
```

Then, control the corresponding GPIO with the corresponding state:

```
digitalWrite(output.toInt(), state.toInt());
```

The `digitalWrite()` function accepts *int* variables as arguments. The `output` and `state` are string variables. That's why we use the `toInt()` method.

Here's the complete code to handle the HTTP GET /update request:

```
//GET request to <ESP_IP>/update?output=<output>&state=<state>
server.on("/update", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String output;
    String state;
    // GET input1 value on <ESP_IP>/update?output=<output>&state=<state>
    if (request->hasParam(PARAM_INPUT_OUTPUT) && request->hasParam(PARAM_INPUT_STATE)) {
        output = request->getParam(PARAM_INPUT_OUTPUT)->value();
        state = request->getParam(PARAM_INPUT_STATE)->value();
        // Control GPIO
        digitalWrite(output.toInt(), state.toInt());
    }
    else {
```

```

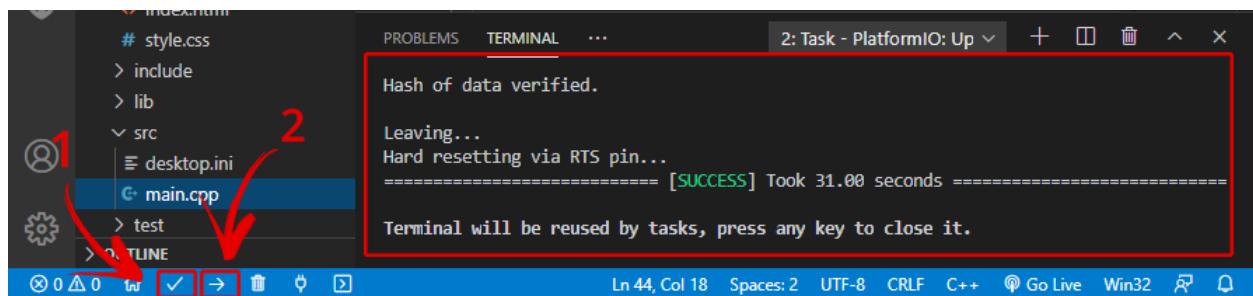
        output = "No message sent";
        state = "No message sent";
    }
    Serial.print("GPIO: ");
    Serial.print(output);
    Serial.print(" - Set to: ");
    Serial.println(state);

    request->send(200, "text/plain", "OK");
});

```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



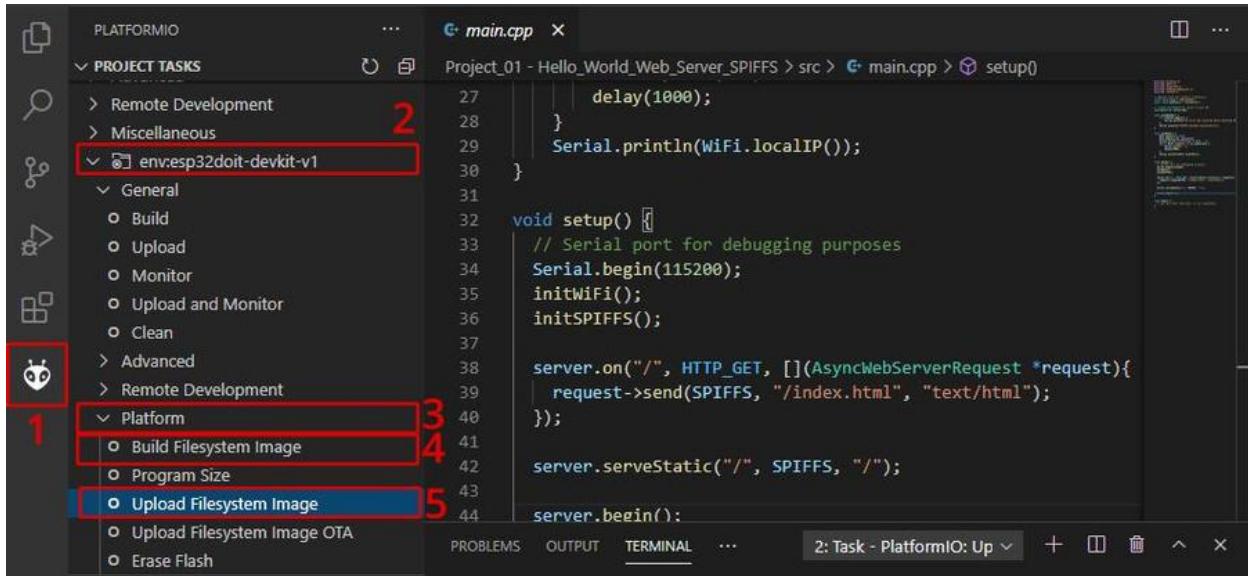
Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, you need to upload the files (*index.html*, *style.css*, *script.js* and *favicon.png*) to the filesystem:

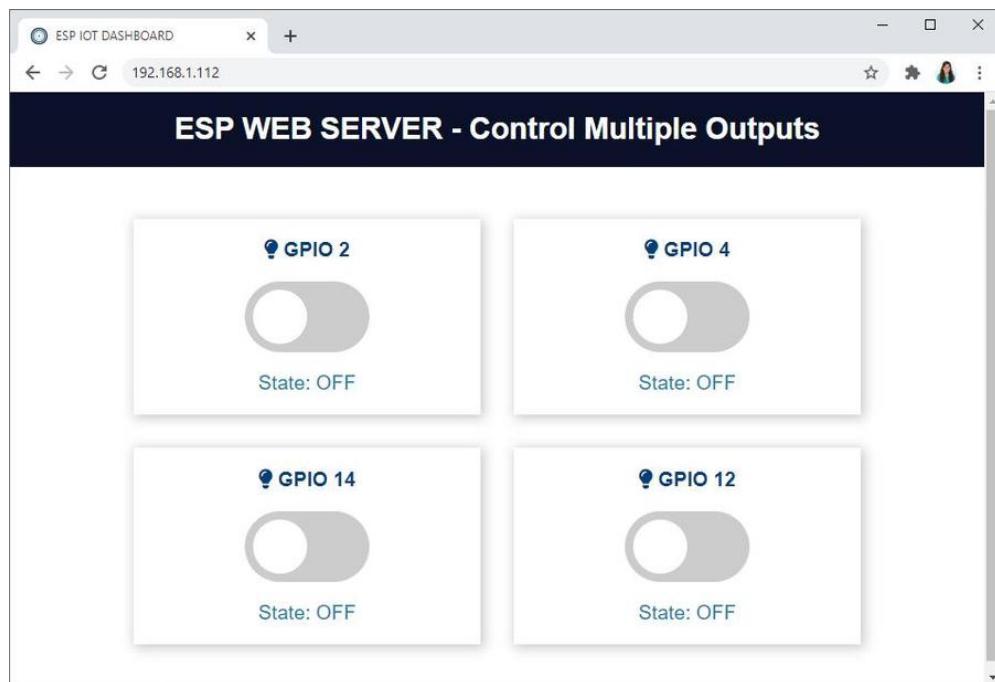
1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.

5. Finally, click **Upload Filesystem Image**.

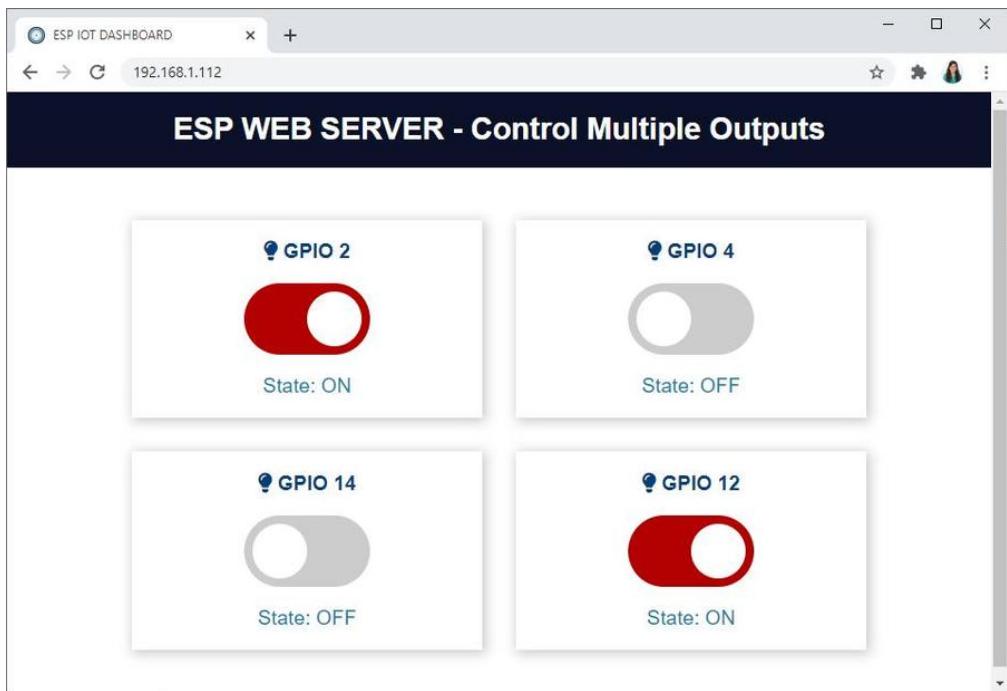


Demonstration

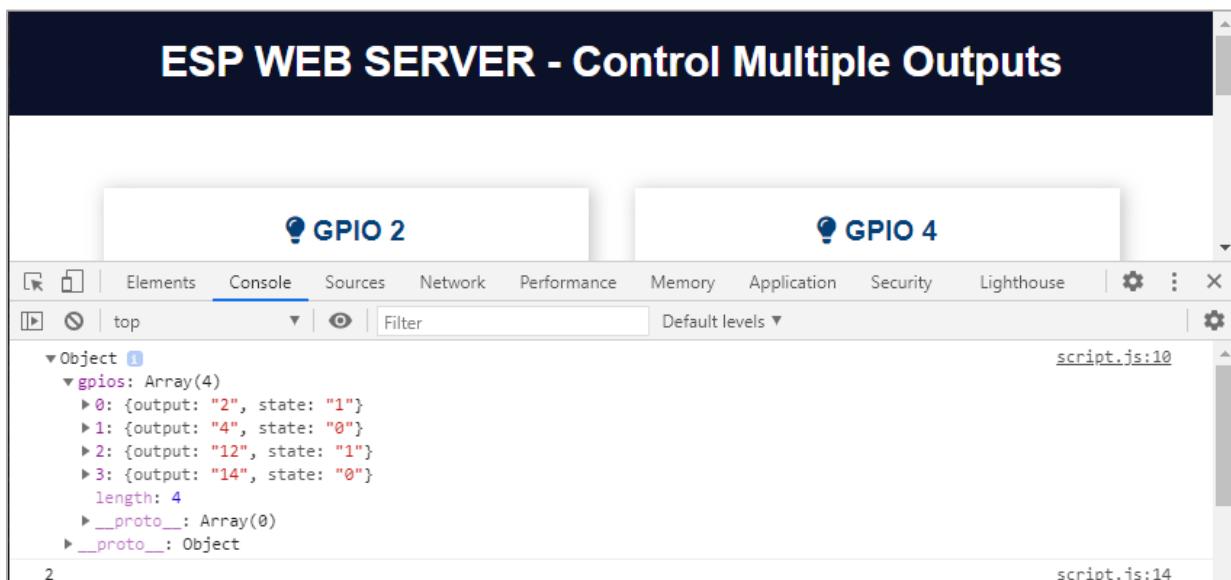
After uploading the code, access your web server by typing the ESP IP address in your browser. This is what you should see.



Toggle the buttons to control the ESP32/ESP8266 GPIOs.



If you close your browser and open it again, the web page will automatically load the current GPIO states. In your browser, you can click **Ctrl+Shift+J** (Windows) or **Option+⌘+C** (Mac OS) to see the JavaScript console and check that it receives the JSON response to update the states.



Here's how the web page looks on your smartphone.



Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this project, you've learned how to pass parameters to the request URL and how to get the parameters' values on the ESP board. As an example, you've controlled multiple outputs. This technique can also be used to pass data that a user enters in an input field, for example. You also learned how to send a JSON response using the ESP board and get the data from that response to make something happen on your web page.

With this example, we also covered how to make requests using JavaScript when some event happens: when the web page loads or when you click on a button, for example.

In the next Unit, you'll learn how to build the previous project using the WebSocket protocol. Using the WebSocket protocol instead of XMLHttpRequests can have several advantages, as we'll see later on.

2.3 - WebSocket Web Server: Control Outputs (ON/OFF Buttons)

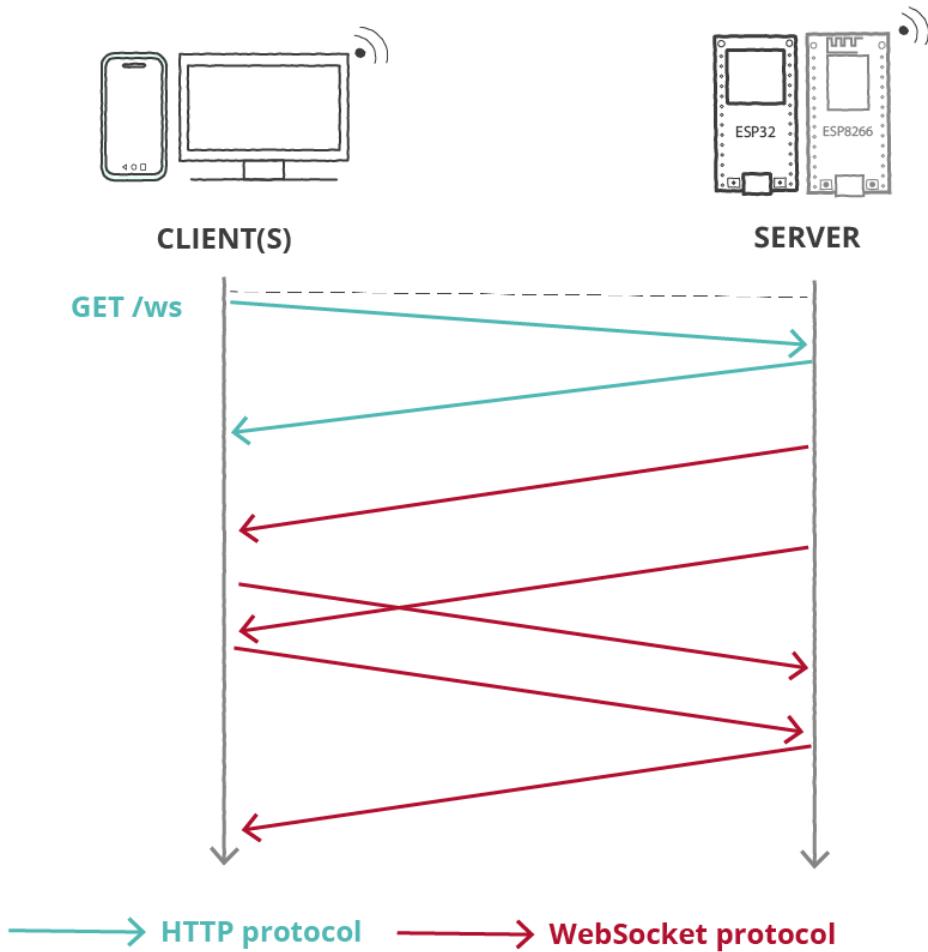


If you've been testing the web servers created previously, you may have noticed that if you have several tabs (in the same or on different devices) opened at the same time, the state doesn't update in all tabs automatically unless you refresh the web page. To solve this issue, we can use the WebSocket protocol – all clients can be notified when a change occurs and update the web page accordingly.

In this tutorial, you'll learn how to build a web server with the ESP32 and ESP8266 using the WebSocket communication protocol. We'll use the same web page from Project 2.1. The output state is displayed on the web page, and it updates automatically in all clients whenever there's a change.

WebSocket

A WebSocket is a persistent connection between a client and a server that allows bidirectional communication between both parties using a TCP connection. This means you can send data from the client to the server and from the server to the client at any given time.



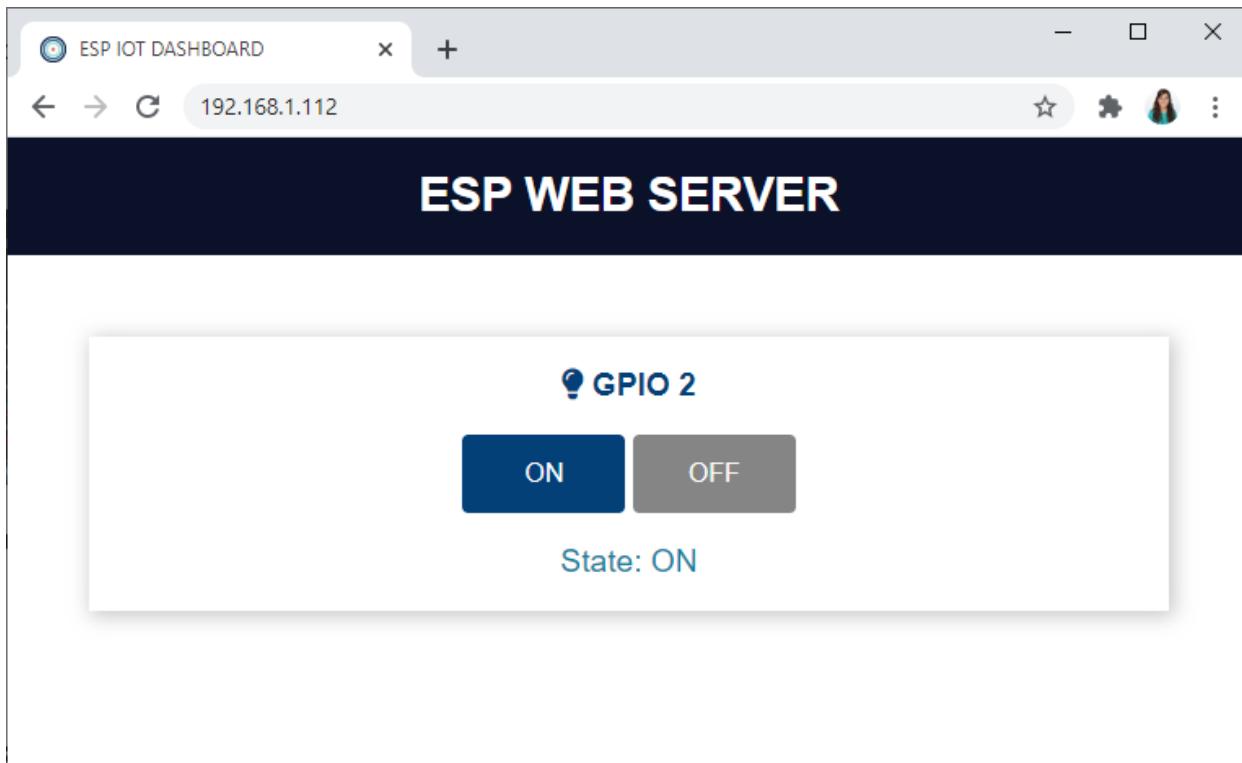
The client establishes a WebSocket connection with the server through a process known as a *WebSocket handshake*. The handshake starts with an HTTP request/response, allowing servers to handle HTTP connections and WebSocket connections on the same port. Once the connection is established, the client and the server can send WebSocket data in full duplex mode.

With the WebSocket protocol, the server (ESP32 and ESP8266) can send information to the client or all clients without being requested. This allows us to send data to the web browser when a change occurs.

This change can be something that happened on the web page (you clicked a button) or something that happened on the ESP side, like pressing a physical button on a circuit.

Project Overview

In this Unit, we'll build the same web server presented in Project 2.1, but we'll use the WebSocket protocol.

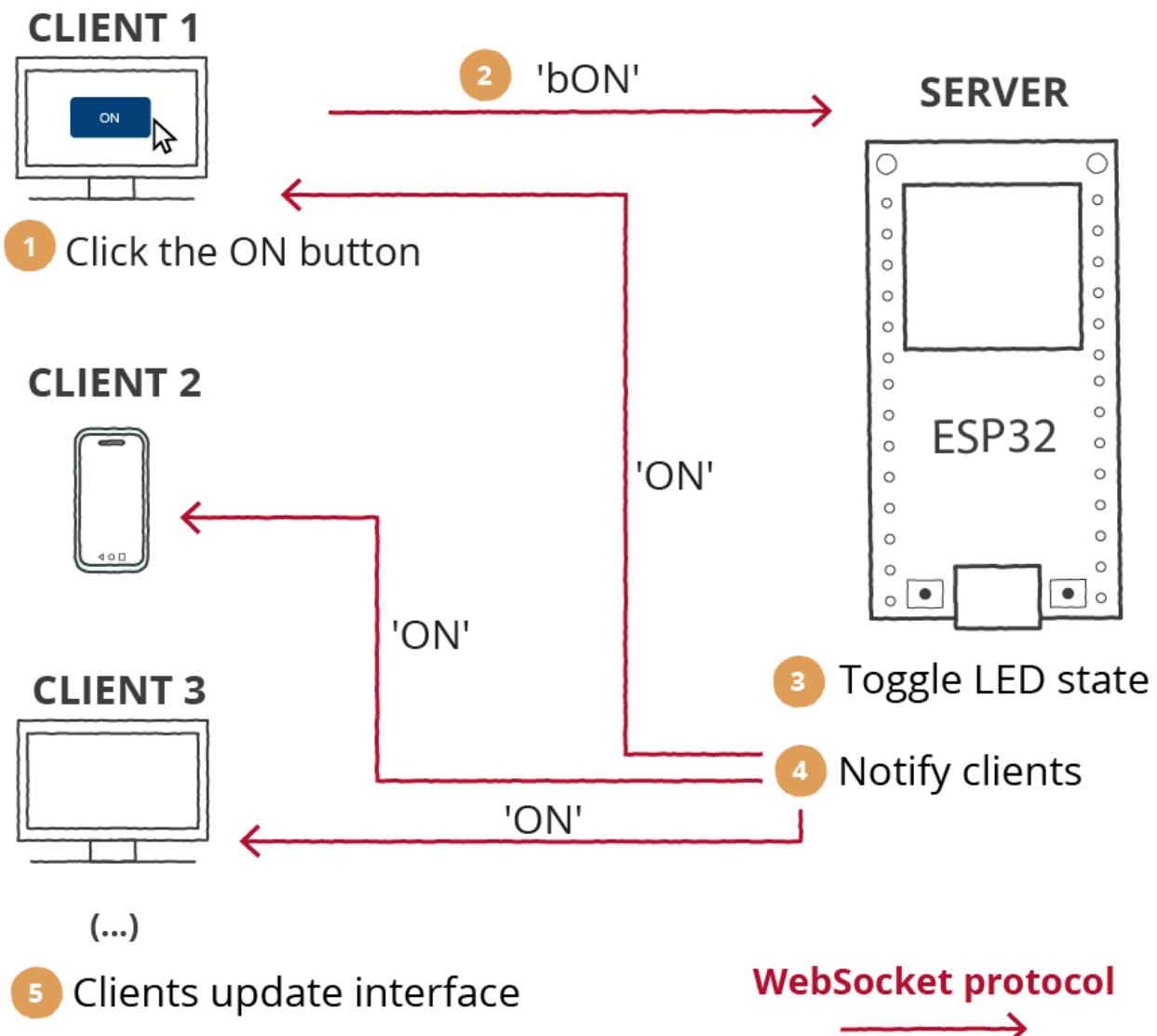


- The web server displays ON and OFF buttons to control GPIO 2;
- The interface shows the current GPIO state. Whenever a change occurs on the GPIO state, the interface is updated instantaneously;

- The GPIO state is updated automatically in all clients. This means that if you have several web browser tabs opened on the same device or different devices, they are all updated simultaneously.

How it Works?

The following image describes what happens when you click the ON button.



- Click the "ON" button;
- The client (your browser) sends data via the WebSocket protocol with the "bON" message;

3. The server (ESP32 or ESP8266) receives this message, so it knows it should turn the LED on;
4. Then, it sends data with the new LED state to all clients also through the WebSocket protocol;
5. The clients receive the message and update the LED state on the web page accordingly. This allows us to update all clients almost instantaneously when a change happens.

Building the Web Page

To build the web page for this project, place the following files in a *data* folder within your project folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*

HTML File

The *index.html* file for this project is very similar to the one used in Unit 2.1. Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/png" href="favicon.png">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqi7RrhN7udi9RwhKkMhpvLbHG9Sr" crossorigin="anonymous">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER</h1>
```

```

</div>
<div class="content">
  <div class="card-grid">
    <div class="card">
      <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
      <p>
        <button class="button-on" id="bON">ON</button>
        <button class="button-off" id="bOFF">OFF</button>
      </p>
      <p class="state">State: <span id="state">%STATE%</span></p>
    </div>
  </div>
  <script src="script.js"></script>
</body>
</html>

```

The only difference is that we add a `` tag to the paragraph that displays the state. That tag adds an `id` to a specific part of the text, so that we can modify it using JavaScript.

```
<p class="state">State: <span id="state">%STATE%</span></p>
```

CSS File

The `style.css` file for this project is the same as Project 2.1. If you want to learn how this CSS file works, go back to that project.

```

html {
  font-family: Arial, Helvetica, sans-serif;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
}
.content {
  padding: 50px;
}
.card-grid {
  max-width: 800px;
}

```

```
margin: 0 auto;
display: grid;
grid-gap: 2rem;
grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
background-color: white;
box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
font-size: 1.2rem;
font-weight: bold;
color: #034078
}
.state {
font-size: 1.2rem;
color: #1282A2;
}
button {
border: none;
color: #FEFCFB;
padding: 15px 32px;
text-align: center;
text-decoration: none;
font-size: 16px;
width: 100px;
border-radius: 4px;
transition-duration: 0.4s;
}
.button-on {
background-color: #034078;
}
.button-on:hover {
background-color: #1282A2;
}
.button-off {
background-color: #858585;
}
.button-off:hover {
background-color: #252524;
}
```

JavaScript File

Copy the following code to your `script.js` file. This file is responsible for initializing a WebSocket connection with the server as soon the web interface is fully loaded in the browser and handling data exchange through the WebSocket protocol.

```
var gateway = `ws://${window.location.hostname}/ws`;
var websocket;
window.addEventListener('load', onload);

function onload(event) {
    initWebSocket();
    initButton();
}

function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    console.log('Connection opened');
}

function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}

function onMessage(event) {
    document.getElementById('state').innerHTML = event.data;
    console.log(event.data);
}

function initButton() {
    document.getElementById('bON').addEventListener('click', toggleON);
    document.getElementById('bOFF').addEventListener('click', toggleOFF);
}

function toggleON(event) {
    websocket.send('bON');
}
function toggleOFF(event) {
    websocket.send('bOFF');
}
```

Let's take a closer look at how this JavaScript code works.

The gateway is the entry point to the WebSocket interface.

```
var gateway = `ws://${window.location.hostname}/ws`;
```

`window.location.hostname` gets the current page address (the web server IP address).

Create a new global variable called `websocket`.

```
var websocket;
```

Add an event listener that will call the `onload` function when the web page loads.

```
window.addEventListener('load', onload);
```

The `onload()` function calls the `initWebSocket()` function to initialize a WebSocket connection with the server and the `initButton()` function to add event listeners to the buttons.

```
function onload(event) {
    initWebSocket();
    initButton();
}
```

The `initWebSocket()` function initializes a WebSocket connection on the gateway defined earlier. We also assign several callback functions for when the WebSocket connection is opened, closed or when a message is received.

```
function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}
```

When the connection is opened, print a message in the console.

```
function onOpen(event) {
    console.log('Connection opened');
}
```

If for some reason the websocket connection is closed, call the `initWebSocket()` function again after 2000 milliseconds (2 seconds).

```
function onClose(event) {
  console.log('Connection closed');
  setTimeout(initWebSocket, 2000);
}
```

The `setTimeout()` method calls a function or evaluates an expression after a specified number of milliseconds.

Finally, we need to handle what happens when the client receives a new message.

The server (your ESP board) will either send an 'ON' or an 'OFF' message.

We want to place that text on the paragraph that displays the state. Remember that `` tag with `id="state"`? We'll get that element and set its value to the data received.

```
function onMessage(event) {
  document.getElementById('state').innerHTML = event.data;
  console.log(event.data);
}
```

The `initButton()` function gets the buttons by their ids ('`bON`' and '`bOFF`') and adds an event listener of type '`click`'.

```
function initButton() {
  document.getElementById('bON').addEventListener('click', toggleON);
  document.getElementById('bOFF').addEventListener('click', toggleOFF);
}
```

This means that when you click the ON button, the `toggleON` function is called. When you click the OFF button, the `toggleOFF` function is called.

The `toggleON()` function sends a message using the WebSocket connection with the '`bON`' text.

```
function toggleON(event) {
  websocket.send('bON');
}
```

The `toggleOFF()` function sends a message using the WebSocket connection with the `'bOFF'` text.

```
function toggleOFF(event) {  
    websocket.send('bOFF');  
}
```

Then, the ESP32 or ESP8266 should handle what happens when it receives these messages — turn the LED on or off and notify all clients.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file within the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` configuration file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]  
platform = espressif32  
board = esp32doit-devkit-v1  
framework = Arduino  
monitor_speed = 115200  
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The `platformio.ini` configuration file for the ESP8266 should be like this.

```
[env:esp12e]  
platform = espressif8266  
board = esp12e  
framework = arduino  
monitor_speed = 115200  
lib_deps = ESP Async WebServer  
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
bool ledState = 0;

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}
```

```

// Replaces placeholder with LED state value
String processor(const String& var){
    if(var == "STATE"){
        if(digitalRead(ledPin)){
            ledState = 1;
            return "ON";
        }
        else{
            ledState = 0;
            return "OFF";
        }
    }
    return String();
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "bON") == 0) {
            ledState = 1;
            notifyClients("ON");
        }
        if (strcmp((char*)data, "bOFF") == 0) {
            ledState = 0;
            notifyClients("OFF");
        }
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

```

```

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

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

    server.serveStatic("/", SPIFFS, "/");
}

// Start server
server.begin();
}

void loop() {
    ws.cleanupClients();
    digitalWrite(ledPin, ledState);
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

```

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
bool ledState = 0;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

// Replaces placeholder with LED state value
String processor(const String& var){
    if(var == "STATE"){
        if(digitalRead(ledPin)){
            ledState = 0;
            return "ON";
        }
        else{
            ledState = 1;
            return "OFF";
        }
    }
    return String();
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info-
>opcode == WS_TEXT) {
        data[len] = 0;
    }
}

```

```

    if (strcmp((char*)data, "bON") == 0) {
        ledState = 0;
        notifyClients("ON");
    }
    if (strcmp((char*)data, "bOFF") == 0) {
        ledState = 1;
        notifyClients("OFF");
    }
}
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}
void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

void setup() {
    pinMode(ledPin, OUTPUT);
    Serial.begin(115200);
    initFS();
    initWiFi();
    initWebSocket();
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html", false, processor);
    });

    server.serveStatic("/", LittleFS, "/");
}

// Start server
server.begin();
}

void loop() {
    ws.cleanupClients();
    digitalWrite(ledPin, ledState);
}

```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

How The Code Works

The code is very similar to previous projects. So, we'll just take a look at the relevant parts for this WebSocket tutorial.

Define LED Pin

Create a variable called `ledPin` that holds the GPIO number you want to control and a variable called `ledState` to hold the current GPIO state.

```
// Set LED GPIO
const int ledPin = 2;
// Stores LED state
String ledState;
```

processor() function

The `processor()` function will replace the placeholders in the HTML text with the actual values.

```
String processor(const String& var){
  if(var == "STATE"){
    if(digitalRead(ledPin)){
      ledState = "ON";
    }
    else{
      ledState = "OFF";
    }
    return ledState;
  }
  return String();
}
```

When the web page is requested, the ESP checks if the HTML text has any placeholders. If it finds the `%STATE%` placeholder, we read the current GPIO state with

`digitalRead(ledPin)` and set the `ledState` value variable accordingly. The function returns the current GPIO state as a string variable.

Note: the ESP8266 on-board LED works with inverted logic. It lights up when you send a LOW signal and turns off when you send a HIGH signal.

Notify All Clients

The `notifyClients()` function notifies all clients with a message containing whatever you pass as an argument. In this case, we want to notify all clients of the current LED state whenever there's a change.

```
void notifyClients(String state) {
    ws.textAll(state);
}
```

The `AsyncWebSocket` class provides a `textAll()` method for sending the same message to all clients that are connected to the server at the same time.

Handle WebSocket Messages

The `handleWebSocketMessage()` function is a callback function that will run whenever we receive new data from the clients via the WebSocket protocol.

```
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info-
>opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "bON") == 0) {
            ledState = 1;
            notifyClients("ON");
        }
        if (strcmp((char*)data, "bOFF") == 0) {
            ledState = 0;
            notifyClients("OFF");
        }
    }
}
```

If we receive the "bON" message, set the value of the ledState variable to 1 and notify all clients by calling the `notifyClients()` function.

```
if (strcmp((char*)data, "bON") == 0) {
    ledState = 1;
    notifyClients("ON");
}
```

If the message received is "bOFF", set the ledState variable to 0 and notify all clients.

```
if (strcmp((char*)data, "bOFF") == 0) {
    ledState = 0;
    notifyClients("OFF");
}
```

This way, all clients are notified when there's a change and update the interface accordingly.

Configure the WebSocket server

Now we need to configure an event listener to handle the different asynchronous steps of the WebSocket protocol. This event handler can be implemented by defining the `onEvent()` as follows:

```
void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}
```

The `type` argument represents the event that occurs. It can take the following values:

- `WS_EVT_CONNECT` when a client has logged in;
- `WS_EVT_DISCONNECT` when a client has logged out;
- `WS_EVT_DATA` when a data packet is received from the client;
- `WS_EVT_PONG` in response to a ping request;
- `WS_EVT_ERROR` when an error is received from the client.

Initialize WebSocket

Finally, the `initWebSocket()` function initializes the WebSocket protocol.

```
void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
```

setup()

In the `setup()`, set GPIO 2 as an `OUTPUT`, initialize the Serial Monitor, Wi-Fi, the filesystem and the WebSocket protocol. If you're using the ESP32, initialize SPIFFS. If you're using the ESP8266 initialize LittleFS.

```
void setup() {
    pinMode(ledPin, OUTPUT);
    Serial.begin(115200);
    initSPIFFS();
    initWiFi();
    initWebSocket();
```

Handle Requests

The following lines handle what happens when you receive a request on the root (/) URL (ESP IP address).

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

When it receives that request, it sends the HTML text saved in the `index.html` file to build the web page. It also needs to pass the `processor` function as an argument to replace all the placeholders with the correct values.

The first argument of the `send()` function is the filesystem where the files are saved. In this case, it is saved in SPIFFS (or LittleFS in the case of the ESP8266). The second argument is the path where the file is located. The third argument refers to the content type (HTML text). The third argument means `download=false`. Finally, the last argument is the `processor` function.

When the HTML file loads in your browser, it will make a request for the CSS, JavaScript, and favicon files. These are static files saved in the same directory (SPIFFS or LittleFS). So, we can simply add the following line to serve files in a directory when requested by the root URL. It will serve the CSS and favicon files automatically.

```
server.serveStatic("/", SPIFFS, "/");
```

If you're using an ESP8266, it should be like this:

```
server.serveStatic("/", LittleFS, "/");
```

Finally, start the server.

```
server.begin();
```

loop()

The LED is physically controlled on the `loop()`.

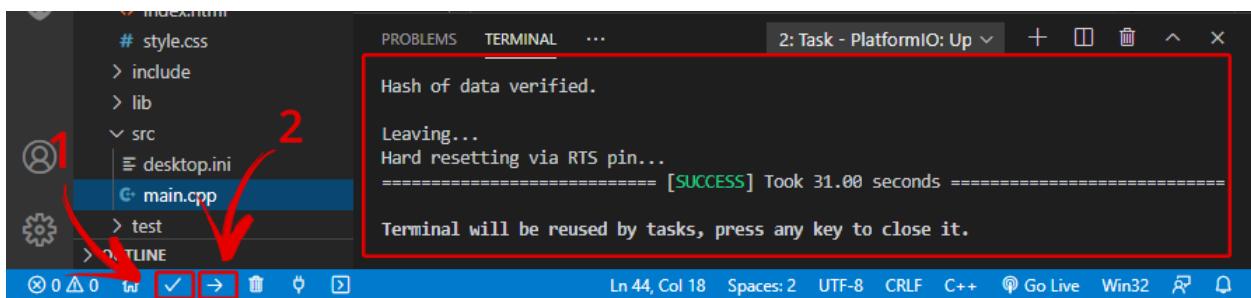
```
void loop() {
    ws.cleanupClients();
    digitalWrite(ledPin, ledState);
}
```

Note that we call the `cleanupClients()` method. Here's why (explanation from the ESPAsyncWebServer library GitHub page):

Browsers sometimes do not correctly close the WebSocket connection, even when the `close()` function is called in JavaScript. This will eventually exhaust the web server's resources and will cause the server to crash. Periodically calling the `cleanupClients()` function from the main `loop()` limits the number of clients by closing the oldest client when the maximum number of clients has been exceeded. This can be called every cycle, however, if you wish to use less power, then calling as infrequently as once per second is sufficient.

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



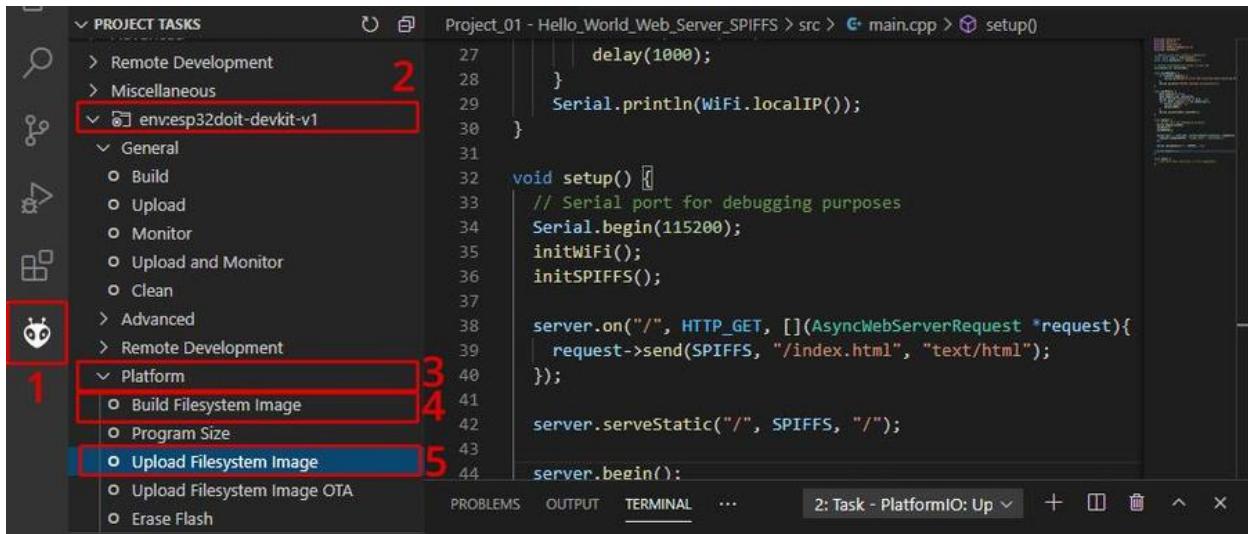
Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, upload the files (`index.html`, `style.css`, `script.js` and `favicon.png`) to the filesystem:

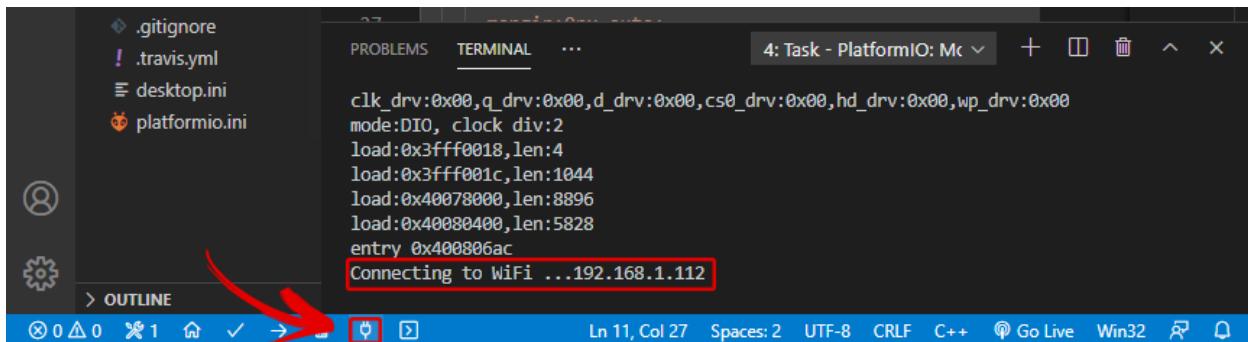
1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).

3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.

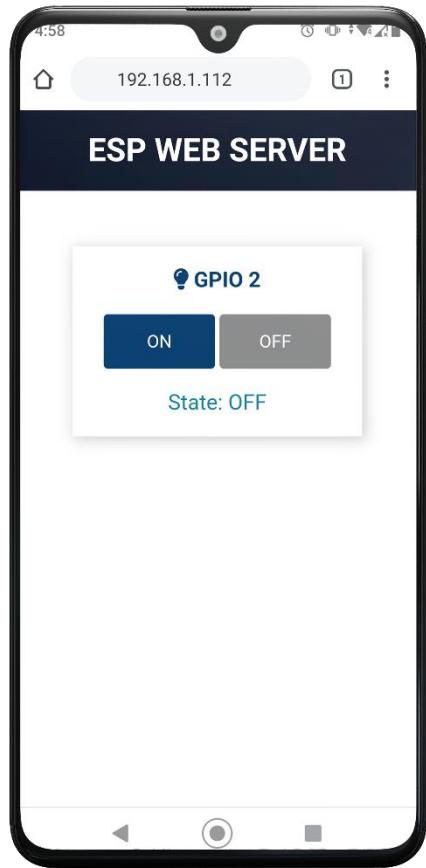


Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous example, it will probably have the same IP.



Open a browser on your local network and insert the ESP32 IP address. You should get access to the web page to control the output.



Click on the buttons to control the LED. You can open several web browser tabs simultaneously or access the web server on different devices simultaneously. The LED state is updated automatically in all clients whenever there's a change.

Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

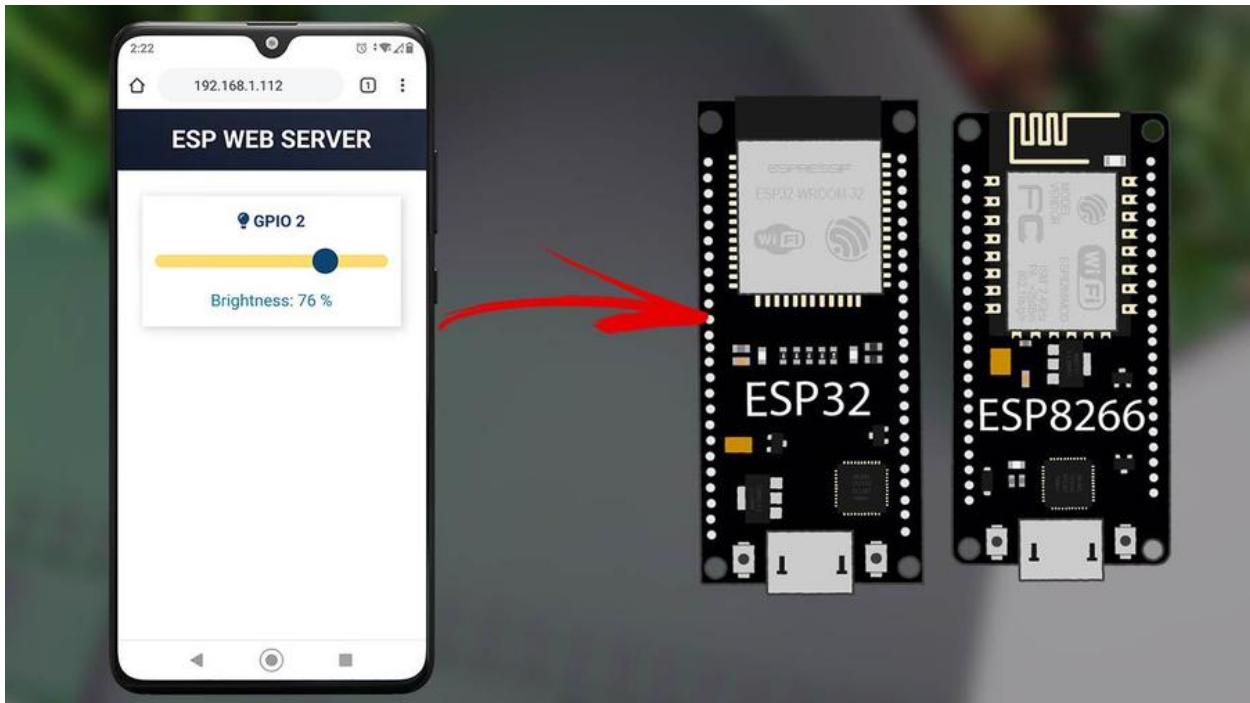
Wrapping Up

In this project, you've learned how to set up a WebSocket server with the ESP32. The WebSocket protocol allows a full duplex communication between the client and the server. After initializing, the server and the client can exchange data at any given time.

This is very useful because the server can send data to the client whenever something happens. For example, you can add a physical button to this setup that notifies all clients to update the web interface when pressed.

In this example, we've shown you how to control one GPIO of the ESP32. You can use this method to control more GPIOs. You can also use the WebSocket protocol to send sensor readings or notifications at any given time.

2.4 - Web Server with Slider: Control LED Brightness (PWM)

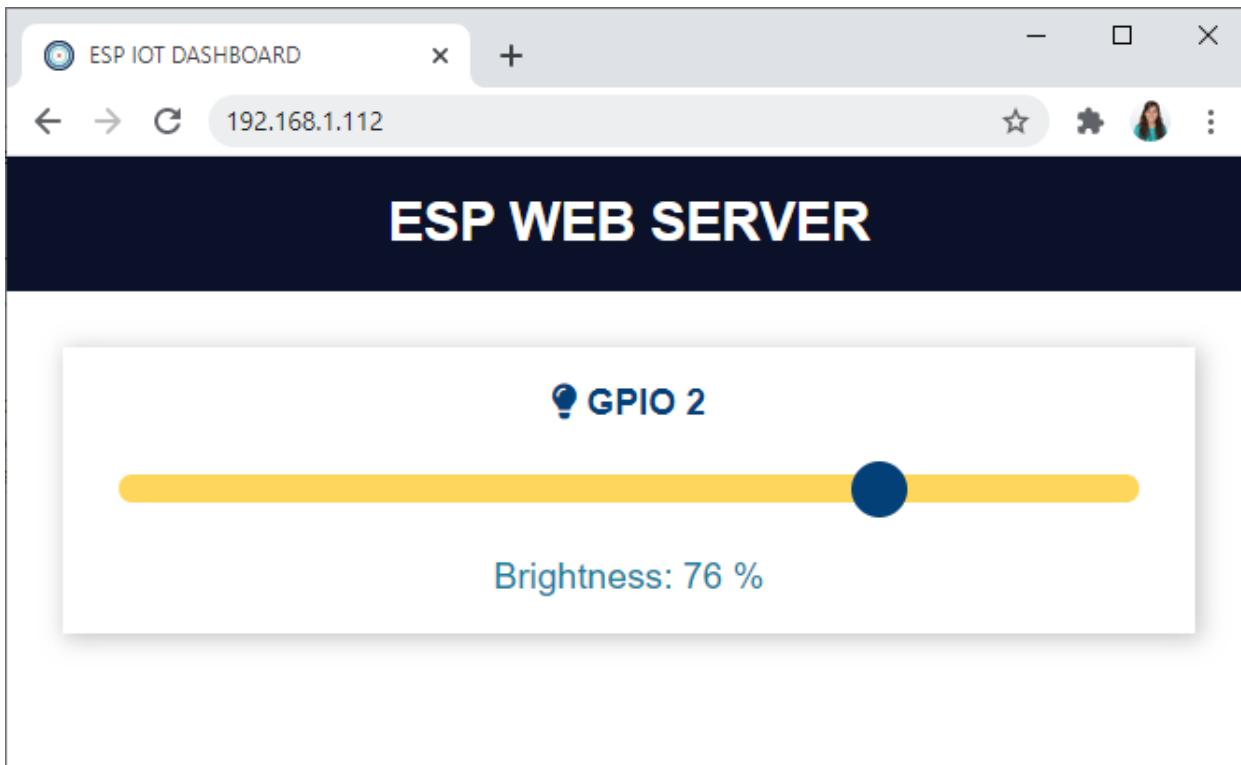


This Unit shows how to build an ESP32 or ESP8266 web server with a slider to control the LED brightness. You'll learn how to add a slider to your web server projects, get its value and save it in a variable that the ESP can use. We'll use that value to control the duty cycle of a PWM signal and change the brightness of an LED. Instead of an LED, you can control a servo motor, for example.

Additionally, you can also modify the code presented in this section to add a slider to your projects to set a threshold value or any other value you need to use in your code.

Project Overview

Here's the web page we'll build for this project.

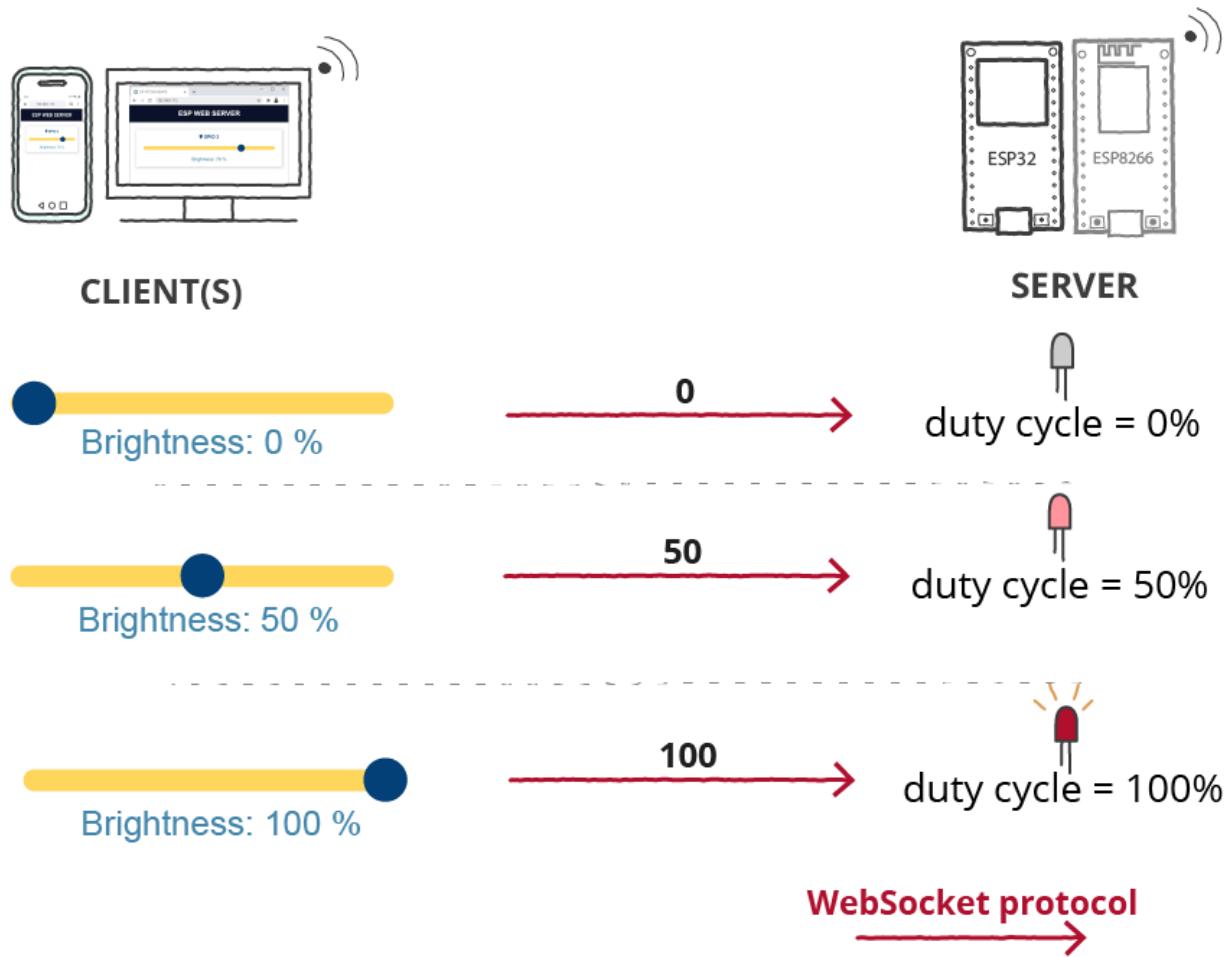


- The web page contains a card with a paragraph to display the card title;
- There's a range slider that you can drag to set the LED brightness;
- A paragraph displays the current LED brightness;
- The brightness value changes dynamically as you drag the slider;
- The actual brightness of the LED is adjusted in real-time every time you move the slider.

How it Works?

- The ESP hosts a web server that displays a web page with a slider;
- Every time you move the slider, the client sends the slider value to the server via the WebSocket protocol;

- The server (ESP) receives the slider value and adjusts the PWM duty cycle accordingly;
- The ESP outputs the PWM signal with the corresponding duty cycle to control the LED brightness. A duty cycle of 0% means the LED is completely off, a duty cycle of 50% means the LED is half lit, and a duty cycle of 100% means the LED is lit;
- This project can be useful to control the brightness of an LED (as we'll do in this example), a servo motor, setting up a threshold value, or other applications.



Building the Web Page

To build the web page for this project, place the following files in the *data* folder within your project folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*

HTML File

Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/png" href="favicon.png">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER</h1>
  </div>
  <div class="content">
    <div class="card-grid">
      <div class="card">
        <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
        <p class="switch">
          <input type="range" oninput="updateSliderPWM(this)" id="pwmSlider" min="0" max="100" step="1" value="0" class="slider">
        </p>
        <p class="state">Brightness: <span id="textSliderValue"></span> &percnt;</p>
      </div>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

Let's take a look at the card that displays the slider, the title and the value.

```
<p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
<p class="switch">
<input type="range" oninput="updateSliderPWM(this)" id="pwmSlider" min="0" max="100" step="1" value="0" class="slider">
</p>
<p class="state">Brightness: <span id="textSliderValue"></span> &percnt;</p>
```

The first paragraph displays a title for the card as in the previous projects. It contains the "GPIO 2" text. You can change the text to whatever you want.

```
<p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
```

To create a slider in HTML you use the `<input>` tag. The `<input>` tag specifies a field where the user can enter data.

There are a wide variety of input types. To define a slider, use the `type` attribute with the `range` value. In a slider, you also need to define the minimum and the maximum range using the `min` and `max` attributes (in this case, `0` and `100`, respectively).

You also need to define other attributes like:

- the `step` attribute specifies the interval between valid numbers. In our case, we set it to `1`;
- the `class` to style the slider (`class="slider"`);
- the `id` so that we can manipulate the slider value using JavaScript (`id="pwmSlider"`);
- the `oninput` attribute to call a function (`updateSliderPWM(this)`) when you move the slider. This function sends the current slider value via the WebSocket protocol to the client. The `this` keyword refers to the HTML slider element.

The slider is inside a paragraph with the `switch` class name. So, here are the tags to create the slider.

```
<p class="switch">
<input type="range" oninput="updateSliderPWM(this)" id="pwmSlider" min="0" max="100" step="1" value="0" class="slider">
</p>
```

oninput vs onchange events

The `oninput` event occurs when an element gets user input. This event occurs when the value of an `<input>` element is changed (drag the slider).

This event is similar to the `onchange` event. The difference is that the `oninput` event occurs immediately after the value of an element has changed (the event occurs while you drag the slider), while `onchange` occurs when the element loses focus, after the content has been changed (after releasing the slider). To better understand the difference, you can try both ways and see which one you like better.

Finally, there's a paragraph with a span tag ``, so that we can insert the current slider value in that paragraph.

```
<p class="state">Brightness: <span id="textSliderValue"></span> &percnt;</p>
```

CSS File

Copy the following styles to your `style.css` file.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  display: inline-block;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
p {
  font-size: 1.4rem;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
}
.content {
```

```
padding: 30px;
}
.card-grid {
  max-width: 700px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
  background-color: white;
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
  font-size: 1.2rem;
  font-weight: bold;
  color: #034078
}
.state {
  font-size: 1.2rem;
  color: #1282A2;
}
.slider {
  -webkit-appearance: none;
  margin: 0 auto;
  width: 100%;
  height: 15px;
  border-radius: 10px;
  background: #FFD65C;
  outline: none;
}
.slider::-webkit-slider-thumb {
  -webkit-appearance: none;
  appearance: none;
  width: 30px;
  height: 30px;
  border-radius: 50%;
  background: #034078;
  cursor: pointer;
}
.slider::-moz-range-thumb {
  width: 30px;
  height: 30px;
  border-radius: 50% ;
  background: #034078;
  cursor: pointer;
}
.switch {
  padding-left: 5%;
  padding-right: 5%;
}
```

We're using the same styles used in previous projects. We've just added some new instructions to format the slider range. In this example, we need to use the vendor prefixes for the `appearance` attribute.

```
.slider {  
  -webkit-appearance: none;  
  margin: 0 auto;  
  width: 100%;  
  height: 15px;  
  border-radius: 10px;  
  background: #FFD65C;  
  outline: none;  
}  
.slider::-webkit-slider-thumb {  
  -webkit-appearance: none;  
  appearance: none;  
  width: 30px;  
  height: 30px;  
  border-radius: 50%;  
  background: #034078;  
  cursor: pointer;  
}  
.slider::-moz-range-thumb {  
  width: 30px;  
  height: 30px;  
  border-radius: 50% ;  
  background: #034078;  
  cursor: pointer;  
}  
.switch {  
  padding-left: 5%;  
  padding-right: 5%;  
}
```

Let's take a look at the `.slider` selector (styles the slider itself):

```
.slider {  
  -webkit-appearance: none;  
  margin: 0 auto;  
  width: 100%;  
  height: 15px;  
  border-radius: 10px;  
  background: #FFD65C;  
  outline: none;  
}
```

Setting `-webkit-appearance` to `none` overrides the default CSS styles applied to the slider in Google Chrome, Safari, and Android browsers.

```
-webkit-appearance: none;
```

Setting the margin to `0 auto` aligns the slider inside its parent container.

```
margin: 0 auto;
```

The width of the slider is set to `100%` and the height to `15px`. The border-radius is set to `10px`.

```
margin: 0 auto;  
width: 100%;  
height: 15px;  
border-radius: 10px;
```

Set the background color for the slider and set the outline to none.

```
background: #FFD65C;  
outline: none;
```

Then, format the slider handle. Use `-webkit-` for Chrome, Opera, Safari and Edge web browsers and `-moz-` for Firefox.

```
.slider::-webkit-slider-thumb {  
    -webkit-appearance: none;  
    appearance: none;  
    width: 30px;  
    height: 30px;  
    border-radius: 50%;  
    background: #034078;  
    cursor: pointer;  
}  
.slider::-moz-range-thumb {  
    width: 30px;  
    height: 30px;  
    border-radius: 50% ;  
    background: #034078;  
    cursor: pointer;  
}
```

Set the `-webkit-appearance` and `appearance` properties to `none` to override default properties.

```
-webkit-appearance: none;  
appearance: none;
```

Set a specific width, height and border-radius for the handler. Setting the same width and height with a border-radius of 50% creates a circle.

```
width: 30px;  
height: 30px;  
border-radius: 50%;
```

Then, set a color for the background and set the cursor to a pointer.

```
background: #034078;  
cursor: pointer;
```

Feel free to play with the slider properties to give it a different look.

JavaScript File

Copy the following code to your *script.js* file. This is responsible for initializing a WebSocket connection with the server, asking the server for the current slider value as soon the web interface is fully loaded in the browser, and handling data exchange through the WebSocket protocol.

```
var gateway = `ws://${window.location.hostname}/ws`;  
var websocket;  
  
window.addEventListener('load', onload);  
  
function onload(event) {  
    initWebSocket();  
    getCurrentValue();  
}  
function initWebSocket() {  
    console.log('Trying to open a WebSocket connection...');  
    websocket = new WebSocket(gateway);  
    websocket.onopen = onOpen;  
    websocket.onclose = onClose;  
    websocket.onmessage = onMessage;  
}  
function onOpen(event) {  
    console.log('Connection opened');  
}  
function onClose(event) {  
    console.log('Connection closed');  
    setTimeout(initWebSocket, 2000);  
}  
function onMessage(event) {  
    console.log(event.data);  
}
```

```

}

function getCurrentValue() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("pwmSlider").value = this.responseText;
            document.getElementById("textSliderValue").innerHTML = this.responseText;
        }
    };
    xhr.open("GET", "/currentValue", true);
    xhr.send();
}
function updateSliderPWM(element) {
    var sliderValue = document.getElementById("pwmSlider").value;
    document.getElementById("textSliderValue").innerHTML = sliderValue;
    console.log(sliderValue);
    websocket.send(sliderValue);
}

```

Let's take a look at this JavaScript code to see how it works.

The gateway is the entry point to the WebSocket interface.

```
var gateway = `ws://${window.location.hostname}/ws`;
```

`window.location.hostname` gets the current page address (the web server IP address).

Create a new global variable called `websocket`.

```
var websocket;
```

Add an event listener that will call the `onload` function when the web page loads.

```
window.addEventListener('load', onload);
```

The `onload()` function calls the `initWebSocket()` function to initialize a WebSocket connection with the server and the `getCurrentValue()` function to get the current slider value when you first open the web browser.

```

function onload(event) {
    initWebSocket();
    getCurrentValue();
}
```

The `initWebSocket()` function initializes a WebSocket connection on the gateway defined earlier. We also assign several callback functions for when the WebSocket connection is opened, closed or when a message is received. We've already taken a look at these functions in previous projects.

```
function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}
```

getCurrentValue()

The `getCurrentValue()` function makes a request on the `/currentValue` URL and handles the response.

```
function getCurrentValue() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("pwmSlider").value = this.responseText;
            document.getElementById("textSliderValue").innerHTML = this.responseText;
        }
    };
    xhr.open("GET", "/currentValue", true);
    xhr.send();
}
```

When the server receives that request, it responds back with the value of the slider. We set the current slider position to that value. The slider id is `pwmSlider` and you can set its value by using `.value` like this:

```
document.getElementById("pwmSlider").value = this.responseText;
```

Finally, add the value to the paragraph with `textSliderValue` id:

```
document.getElementById("textSliderValue").innerHTML = this.responseText;
```

updateSliderPWM()

The `updateSliderPWM()` function is called whenever you drag the slider.

```
function updateSliderPWM(element) {
  var sliderValue = document.getElementById("pwmSlider").value;
  document.getElementById("textSliderValue").innerHTML = sliderValue;
  console.log(sliderValue);
  websocket.send(sliderValue);
}
```

The function gets the current slider value and saves it in the `sliderValue` variable.

```
var sliderValue = document.getElementById("pwmSlider").value;
```

Then, it updates the slider value on the paragraph with `textSliderValue` id:

```
document.getElementById("textSliderValue").innerHTML = this.responseText;
```

Print the slider value into the console for debugging purposes.

```
console.log(sliderValue);
```

Finally, send the slider value to the client using the WebSocket protocol

```
websocket.send(sliderValue);
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file within the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it is within the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set LED GPIO
const int ledPin = 2;

// setting PWM properties
const int freq = 5000;
const int ledChannel = 0;
const int resolution = 8;

String sliderValue = "0";
int dutyCycle;

// Initialize SPIFFS
```

```

void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        sliderValue = (char*)data;
        dutyCycle = map(sliderValue.toInt(), 0, 100, 0, 255);
        Serial.println(dutyCycle);
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type, void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
void setup() {

```

```

Serial.begin(115200);
pinMode(ledPin, OUTPUT);
initSPIFFS();
initWiFi();

    // configure LED PWM functionalitites
ledcSetup(ledChannel, freq, resolution);

    // attach the channel to the GPIO to be controlled
ledcAttachPin(ledPin, ledChannel);

initWebSocket();

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

server.serveStatic("/", SPIFFS, "/");

server.on("/currentValue", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(200, "/text/plain", String(sliderValue).c_str());
});

// Start server
server.begin();
}

void loop() {
    ledcWrite(ledChannel, dutyCycle);
    ws.cleanupClients();
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>

```

```

#include <ESPAAsyncWebServer.h>
#include "LittleFS.h"

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set LED GPIO
const int ledPin = 2;

String sliderValue = "0";
int dutyCycle;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        sliderValue = (char*)data;
        dutyCycle = map(sliderValue.toInt(), 0, 100, 0, 1023);
        Serial.println(dutyCycle);
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type, void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:

```

```

        Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
        break;
    case WS_EVT_DISCONNECT:
        Serial.printf("WebSocket client #%u disconnected\n", client->id());
        break;
    case WS_EVT_DATA:
        handleWebSocketMessage(arg, data, len);
        break;
    case WS_EVT_PONG:
    case WS_EVT_ERROR:
        break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

    // Web Server Root URL
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html");
    });

    server.serveStatic("/", LittleFS, "/");
}

server.on("/currentValue", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(200, "text/plain", String(sliderValue).c_str());
});

// Start server
server.begin();
}

void loop() {
    analogWrite(ledPin, dutyCycle);
    ws.cleanupClients();
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How The Code Works

Let's take a closer look at the code and see how it works.

Importing Libraries

Import the necessary libraries.

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

If you're using an ESP8266, you should include the following libraries instead:

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
```

Network Credentials

Insert your network credentials in the following variables:

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

AsyncWebServer and AsyncWebSocket

Create an `AsyncWebServer` object on port 80.

```
AsyncWebServer server(80);
```

The `ESPAsyncWebServer` library includes a `WebSocket` plugin that makes it easy to handle `WebSocket` connections. Create an `AsyncWebSocket` object called `ws` to handle the connections on the `/ws` path.

```
AsyncWebSocket ws("/ws");
```

PWM Output

Define the LED pin you want to control. As an example, we'll control GPIO 2, the on-board LED.

```
const int ledPin = 2;
```

Set the PWM properties to control the LED.

```
const int freq = 5000;
const int ledChannel = 0;
const int resolution = 8;
```

If you're not familiar with using PWM with the ESP32, we recommend taking a look at the following tutorial:

- [ESP32 PWM with Arduino IDE \(Analog Output\)](#)

If you're using an ESP8266, you don't need to set the PWM properties.

The `sliderValue` variable holds the value to be displayed on the web page. The `dutyCycle` variable holds the duty cycle value to control the LED brightness.

```
String sliderValue = "0";
int dutyCycle;
```

initSPIFFS() and initWiFi()

You are already familiar with the `initSPIFFS()` and `initWiFi()` function from previous projects.

```
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ...");
```

```
while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
}
Serial.println(WiFi.localIP());
```

Handle WebSocket Messages

The `handleWebSocketMessage()` function is a callback function that will run whenever we receive new data from the clients via the WebSocket protocol.

```
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info-
>opcode == WS_TEXT) {
        data[len] = 0;
        sliderValue = (char*)data;
        dutyCycle = map(sliderValue.toInt(), 0, 100, 0, 255);
        Serial.println(dutyCycle);
    }
}
```

The client sends the current slider value to the server. So, we save that value in the `sliderValue` variable.

```
sliderValue = (char*)data;
```

The `sliderValue` can be a value between 0 and 100, but the duty cycle is a value between 0 and 255. So, we use the `map()` function to convert from the 0-100 range to the 0-255 range. If you're using an ESP8266, the duty cycle range is 0 to 1023.

```
dutyCycle = map(sliderValue.toInt(), 0, 100, 0, 255);
```

Configure the WebSocket server

Now, we need to configure an event listener to handle the different asynchronous steps of the WebSocket protocol. This event handler can be implemented by defining the `onEvent()` as follows:

```
void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType t
ype, void *arg, uint8_t *data, size_t len) {
```

```
switch (type) {
    case WS_EVT_CONNECT:
        Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
        break;
    case WS_EVT_DISCONNECT:
        Serial.printf("WebSocket client #%u disconnected\n", client->id());
        break;
    case WS_EVT_DATA:
        handleWebSocketMessage(arg, data, len);
        break;
    case WS_EVT_PONG:
    case WS_EVT_ERROR:
        break;
}
```

Initialize WebSocket

Finally, the `initWebSocket()` function initializes the WebSocket protocol.

```
void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
```

setup()

In the `setup()`, initialize the Serial Monitor for debugging purposes.

```
Serial.begin(115200);
```

Set up the `ledPin` as an OUTPUT.

```
pinMode(ledPin, OUTPUT);
```

Initialize Wi-Fi and the filesystem.

```
initFS();
initWiFi();
```

Configure the PWM properties and attach a channel to the GPIO you want to control.

```
// configure LED PWM functionalitites
ledcSetup(ledChannel, freq, resolution);
// attach the channel to the GPIO to be controlled
ledcAttachPin(ledPin, ledChannel);
```

Initialize the WebSocket protocol by calling the `initWebSocket()` function created previously.

```
initWebSocket();
```

Handle Requests

Serve the text saved in the `index.html` file variable when you receive a request on the root / URL.

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

Serve the other static files (`style.css`, `JavaScript.js`, and `favicon.png`) saved in the filesystem.

```
server.serveStatic("/", SPIFFS, "/");
```

When you first open the web page, it makes a request on the `/currentValue` path to request the current `sliderValue`. You can simply send a string with that information (third argument of the `send()` function).

```
server.on("/currentValue", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "/text/plain", String(sliderValue).c_str());
});
```

Finally, start the server.

```
server.begin();
```

loop()

The LED will be physically controlled on the `loop()`. Use the `ledcWrite()` function to set the LED brightness with the current duty cycle.

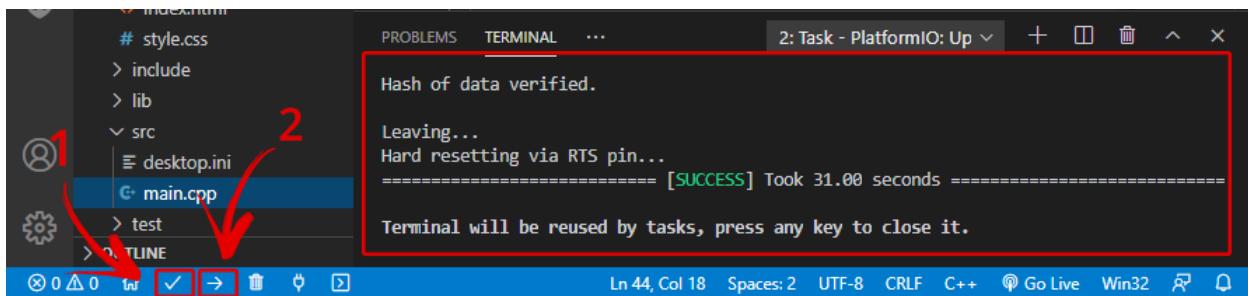
```
void loop() {
    ledcWrite(ledChannel, dutyCycle);
    ws.cleanupClients();
}
```

If you're using an ESP8266, use the `analogWrite()` function instead:

```
analogWrite(ledPin, dutyCycle);
```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



Uploading Filesystem Image

Finally, upload the files (`index.html`, `style.css`, `script.js` and `favicon.png`) to the filesystem:

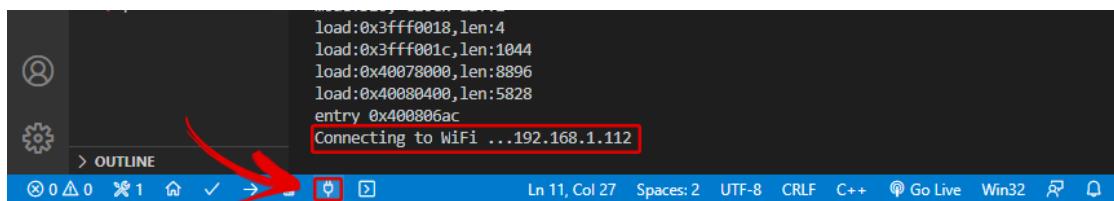
Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select `env:esp12e` or `env:esp32doit-devkit-v1` (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.

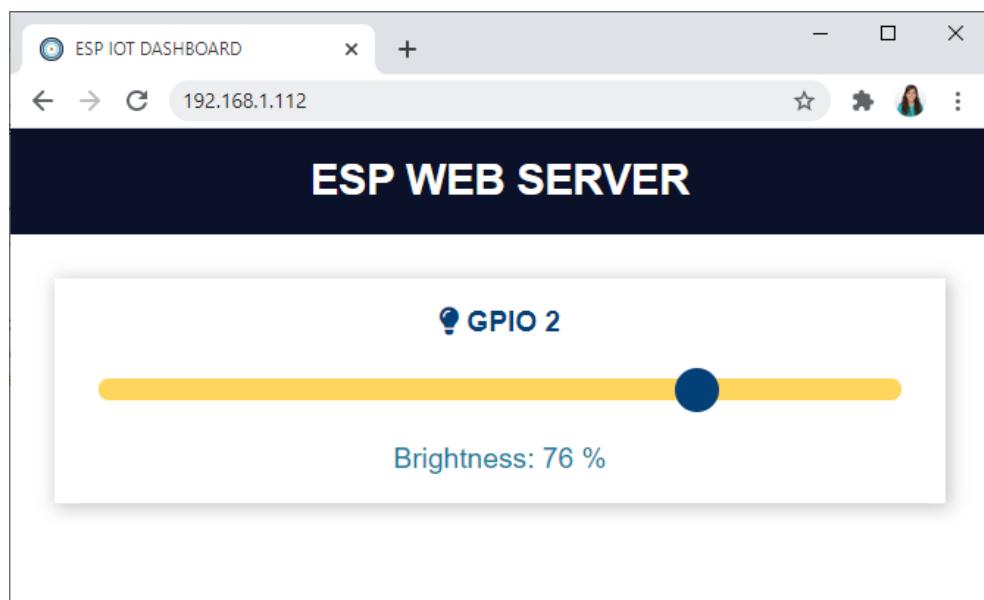


Demonstration

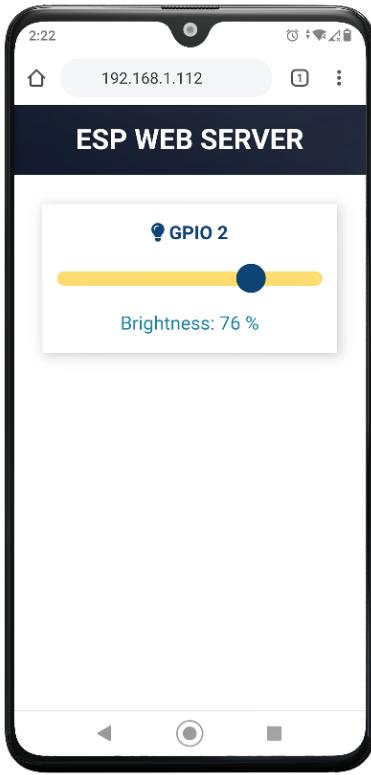
After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.



Open a browser on your local network and insert the ESP IP address. You should get access to the web page to control the LED brightness.



Here's how the web page looks on your smartphone. Move the slider, and you'll see the LED brightness changing accordingly.



Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

With this Unit you've learned how to add a slider to your web server projects, get and save its value in a variable that the ESP32 or ESP8266 can use. As an example, we're controlling a PWM signal to set the brightness of an LED. Instead of an LED, you can control a servo motor. Additionally, the slider may be used to set up a threshold or any other value that you need to use in your project.

2.5 – WebSocket Web Server: Control Multiple Outputs



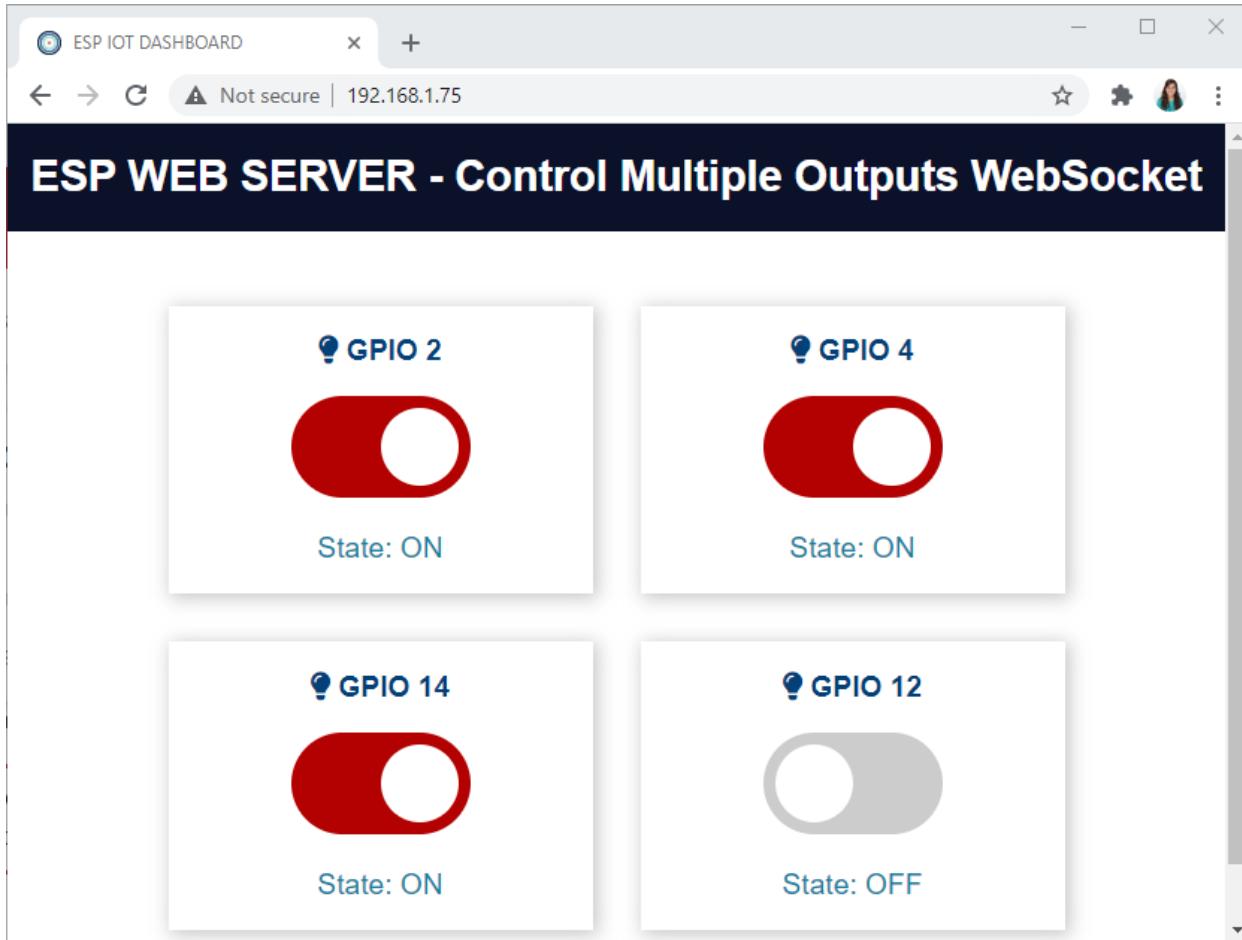
This Unit is similar to *Unit 2.2 – Web Server – Control Multiple Outputs (Toggle Switches)*. Instead of using HTTP requests to control the ESP32 and ESP8266 GPIOs, we'll use the WebSocket protocol. This protocol allows you to notify all clients whenever a change occurs in the GPIO state.

If you have several tabs open at the same time for your web server in different devices, you'll notice that with the WebSocket protocol, the web page updates the GPIO state whenever there's a change. All the information is synchronized in all clients.

To keep things as simple as possible, we'll use the same HTML and CSS files from Unit 2.2. We recommend that you follow Unit 2.2 and Unit 2.3 before proceeding.

Project Overview

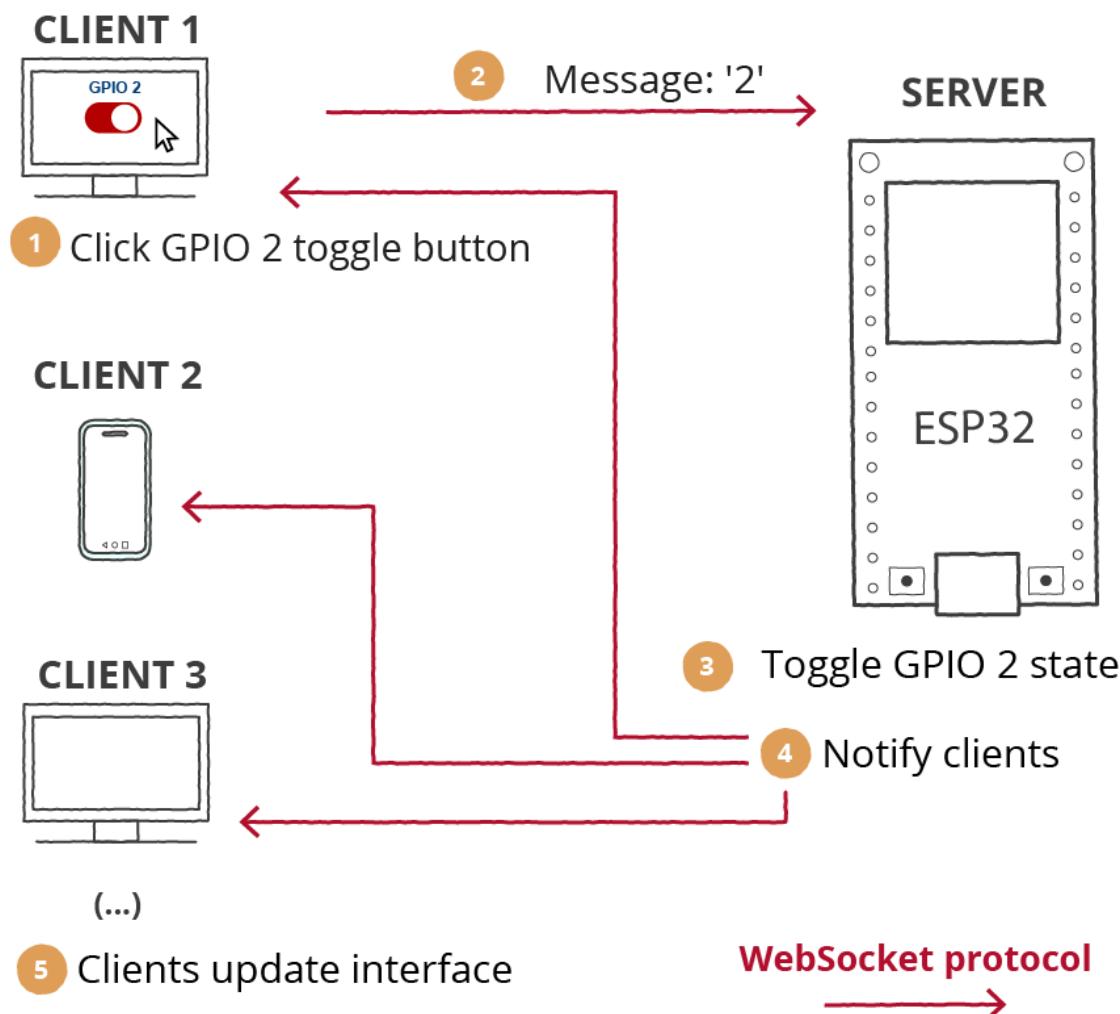
In this Unit, we'll build the same web server presented in Unit 2.2, but we'll use the WebSocket protocol.



- The web server displays four toggle switches to control four GPIOs;
- Each toggle switch has a label indicating the GPIO output pin. You can easily remove/add GPIOs. There's also a label indicating the current GPIO state;
- The toggle switches change color according to the GPIO state;
- You can have multiple web browser tabs open simultaneously, and all the information is updated automatically in all clients.

How it Works?

- 1) When you access the web page on a new client for the first time, it sends a message to the server with the content "states". When the server (ESP32 or ESP8266) receives this message, it responds with a JSON string with the current GPIO states through the WebSocket protocol. This allows us to get the current GPIO states when we access the web server for the first time.
- 2) When you press the toggle buttons, the client sends a message through the WebSocket protocol to the server with the GPIO number that we want to control. The server receives that message and toggles the GPIO state. After that, it sends a message to all clients with the new current GPIO states to have the state updated (see illustration below).



Building the Circuit

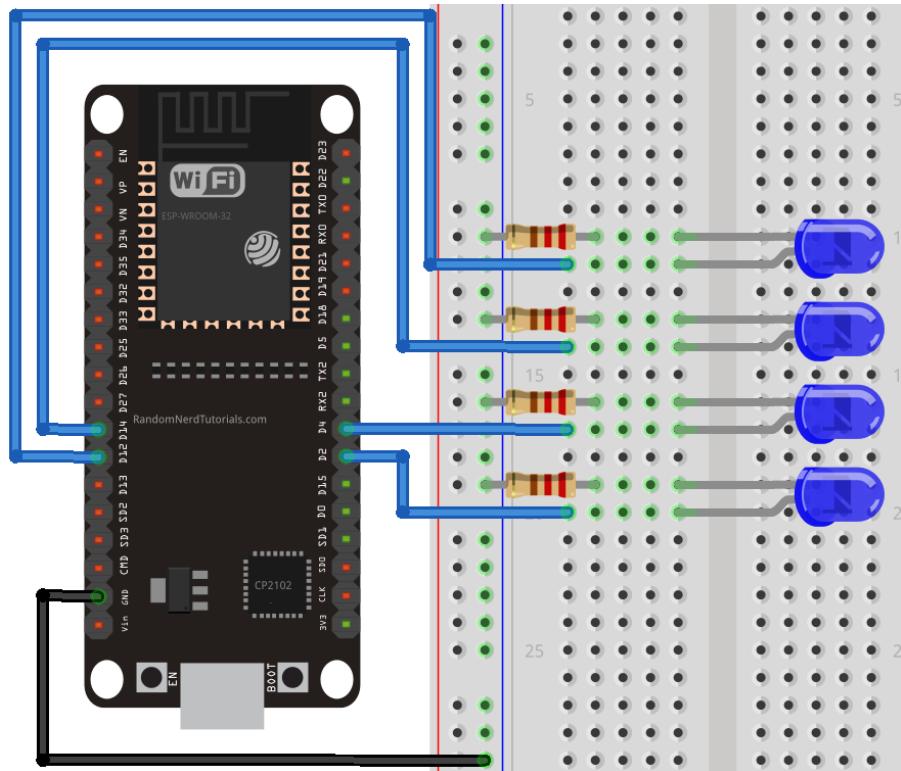
In this project, we'll control multiple ESP32/ESP8266 GPIOs. As an example, we'll control four LEDs attached to the following GPIOs: 2, 4, 12 and 14.

Parts Required:

Here's the parts needed to build the circuit:

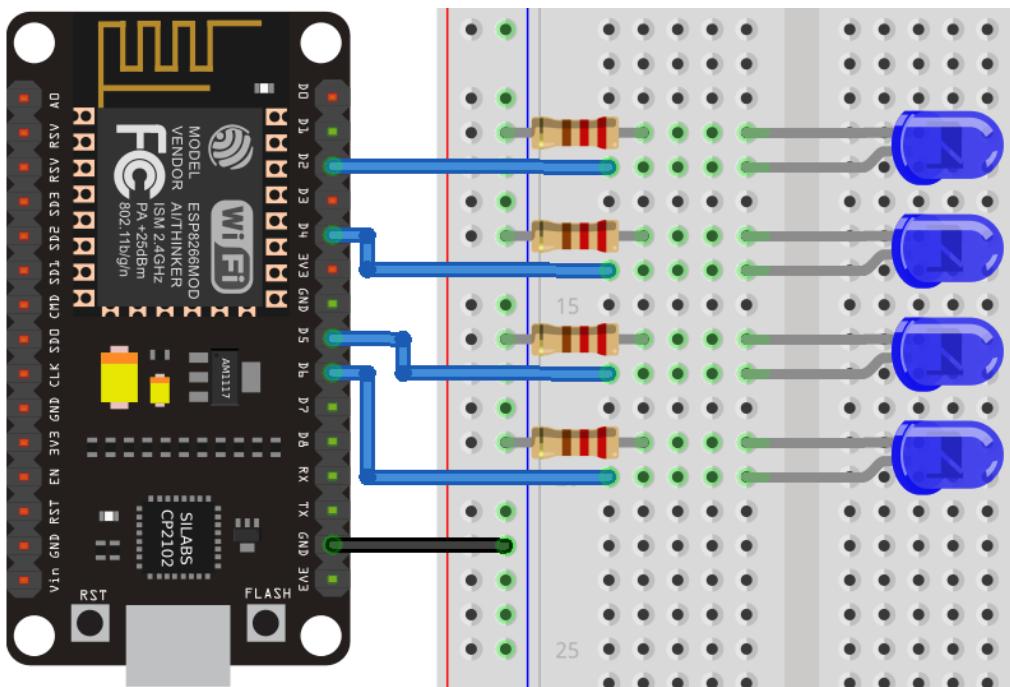
- [ESP32](#) or [ESP8266](#) board
- 4x [LEDs](#)
- 4x [220 Ω resistors](#)
- [Breadboard](#)
- [Jumper wires](#)

ESP32 – Schematic Diagram



Connect to GPIOs: 2, 4, 12 and 14.

ESP8266 – Schematic Diagram



In the ESP8266, the GPIOs are labeled on the silkscreen like this:

- GPIO 4 → D2
- GPIO 2 → D4
- GPIO 14 → D5
- GPIO 12 → D6

Building the Web Page

To build the web page for this project, place the following files inside the *data* folder within your project folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*



For the web page, we'll use the same template of Unit 2.2 (same HTML and same CSS files). Because we've already explained in great detail the HTML and CSS to build the web page, we'll just take a look at the JavaScript file and the server sketch (*main.cpp*).

HTML File

The *index.html* file for this project is the same as project in Unit 2.2.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" type="text/css" href="style.css">
  <link rel="icon" type="image/png" href="favicon.png">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUStjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER - Control Multiple Outputs</h1>
  </div>
  <div class="content">
    <div class="card-grid">
      <div class="card">
        <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
        <label class="switch">
          <input type="checkbox" onchange="toggleCheckbox(this)" id="2">
          <span class="slider"></span>
        </label>
        <p class="state">State: <span id="2s"></span></p>
      </div>
      <div class="card">
        <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 4</p>
        <label class="switch">
```

```

        <input type="checkbox" onchange="toggleCheckbox(this)" id="4">
        <span class="slider"></span>
    </label>
    <p class="state">State: <span id="4s"></span></p>
</div>
<div class="card">
    <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 14</p>
    <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="14">
        <span class="slider"></span>
    </label>
    <p class="state">State: <span id="14s"></span></p>
</div>
<div class="card">
    <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 12</p>
    <label class="switch">
        <input type="checkbox" onchange="toggleCheckbox(this)" id="12">
        <span class="slider"></span>
    </label>
    <p class="state">State: <span id="12s"></span></p>
</div>
</div>
<script src="script.js"></script>
</body>
</html>

```

CSS File

We'll also use the same CSS file of Unit 2.2.

```

html {
    font-family: Arial, Helvetica, sans-serif;
    text-align: center;
}
h1 {
    font-size: 1.8rem;
    color: white;
}
.topnav {
    overflow: hidden;
    background-color: #0A1128;
}
body {
    margin: 0;
}
.content {
    padding: 50px;
}
.card-grid {
    max-width: 600px;
    margin: 0 auto;
}

```

```

display: grid;
gap: 2rem;
grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
background-color: white;
box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
font-size: 1.2rem;
font-weight: bold;
color: #034078
}
.state {
font-size: 1.2rem;
color: #1282A2;
}
.switch {
position: relative;
display: inline-block;
width: 120px;
height: 68px
}
.switch input {
display: none
}
.slider {
position: absolute;
top: 0; left: 0; right: 0; bottom: 0;
background-color: #ccc;
border-radius: 50px
}
.slider:before {
position: absolute;
content: "";
height: 52px;
width: 52px;
left: 8px;
bottom: 8px;
background-color: #fff;
-webkit-transition: .4s;
transition: .4s;
border-radius: 50px;
}
input:checked+.slider {
background-color: #b30000;
}
input:checked+.slider:before {
-webkit-transform: translateX(52px);
-ms-transform: translateX(52px);
transform: translateX(52px);
}

```

JavaScript File

Copy the following code to your *script.js* file. This is responsible for initializing a WebSocket connection with the server as soon the web interface is fully loaded in the browser and handling data exchange through the WebSocket protocol. It is similar to Unit 2.3, but it handles multiple GPIOs.

```
var gateway = `ws://${window.location.hostname}/ws`;
var websocket;

function onLoad(event) {
    initWebSocket();
}

function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    console.log('Connection opened');
    websocket.send("states");
}

function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}

function onMessage(event) {
    var myObj = JSON.parse(event.data);
    console.log(myObj);
    for (i in myObj.gpios){
        var output = myObj.gpios[i].output;
        var state = myObj.gpios[i].state;
        console.log(output);
        console.log(state);
        if (state == "1"){
            document.getElementById(output).checked = true;
            document.getElementById(output+s).innerHTML = "ON";
        }
        else{
            document.getElementById(output).checked = false;
            document.getElementById(output+s).innerHTML = "OFF";
        }
    }
    console.log(event.data);
}
```

```

}

// Send Requests to Control GPIOs
function toggleCheckbox (element) {
    console.log(element.id);
    websocket.send(element.id);
    if (element.checked){
        document.getElementById(element.id+"s").innerHTML = "ON";
    }
    else {
        document.getElementById(element.id+"s").innerHTML = "OFF";
    }
}

// Function to get and update GPIO states on the webpage when it loads for the first time
function getStates(){
    websocket.send("states");
}

window.addEventListener('load', onLoad);

```

Let's take a closer look at how this JavaScript code works.

The gateway is the entry point to the WebSocket interface.

```
var gateway = `ws://${window.location.hostname}/ws`;
```

`window.location.hostname` gets the current page address (the web server IP address). Create a new global variable called `websocket`.

```
var websocket;
```

Add an event listener that will call the `onload` function when the web page loads.

```
window.addEventListener('load', onload);
```

The `onload()` function calls the `initWebSocket()` function to initialize a WebSocket connection with the server.

```
function onload(event) {
    initWebSocket();
}
```

The `initWebSocket()` function initializes a WebSocket connection on the gateway defined earlier. We also assign several callback functions that will be triggered when the WebSocket connection is opened, closed or when a message is received.

```
function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}
```

When the connection is opened, print a message in the console for debugging purposes and send a message saying "states", so that the server knows it needs to send the current GPIO states.

```
function onOpen(event) {
    console.log('Connection opened');
    websocket.send("states");
}
```

If for some reason the web socket connection is closed, call the `initWebSocket()` function again after 2000 milliseconds (2 seconds).

```
function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}
```

Finally, we need to handle what happens when the client receives a new message (`onMessage` event). The server (your ESP board) will send a JSON variable with the current GPIO states in the following format:

```
{
  "gpios": [
    {
      "output": "2",
      "state": "0"
    },
    {
      "output": "4",
      "state": "0"
    },
    {
      "output": "13",
      "state": "1"
    }
  ]
}
```

```
        "output": "12",
        "state": "0"
    },
{
    "output": "14",
    "state": "0"
}
]
```

To learn more about JSON syntax, go back to Unit 2.2.

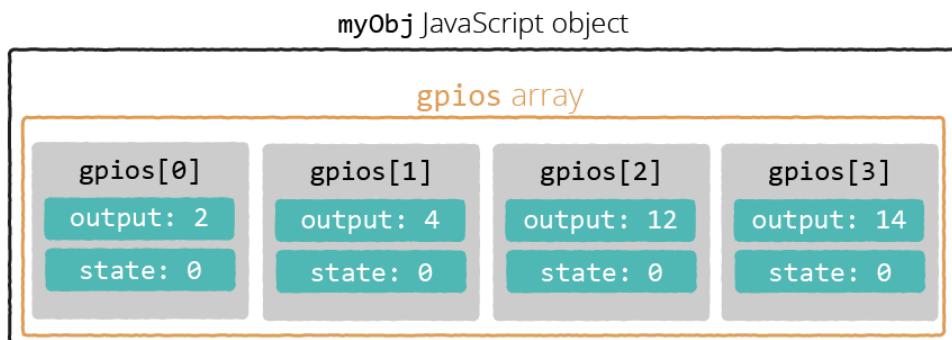
You can get the server response as a JavaScript string using the `event.data` property (because you receive the data on an event). The response comes in JSON format, so we can save the response as a JSON object using the `JSON.parse()` method like this:

```
var myObj = JSON.parse(event.data);
```

For debugging purposes, you can display the value of the `myObj` JSON variable into the console using `console.log()`.

```
console.log(myObj);
```

Now, we need a `for` loop to go through all the outputs and corresponding states.



Take a look at the previous diagram to better understand the structure of the `myObj` object and how to access the GPIOs and its states.

Here's an example on how to access GPIO 2 and its state:

- `myObj.gpios[0].output` returns 2
- `myObj.gpios[0].state` returns 0, the state for GPIO 2

The following `for` loop goes through all objects inside the `gpios` array , gets the GPIOs and corresponding states and saves them in the `output` and `state` JavaScript variables.

```
for (i in myObj.gpios) {  
    var output = myObj.gpios[i].output;  
    var state = myObj.gpios[i].state;  
    console.log(output);  
    console.log(state);
```

Let's examine what happens in the first loop, `i=0`. The `output` variable is 2, and the `state` will be whatever the current state is. In this case, it's 0.

Find the element with the `id="2"` and update the corresponding state either to ON or OFF and set the slider to `checked` or not. Like this:

```
for (i in myObj.gpios) {  
    var output = myObj.gpios[i].output;  
    var state = myObj.gpios[i].state;  
    console.log(output);  
    console.log(state);  
    if (state == "1") {  
        document.getElementById(output).checked = true;  
        document.getElementById(output+s).innerHTML = "ON";  
    }  
    else {  
        document.getElementById(output).checked = false;  
        document.getElementById(output+s).innerHTML = "OFF";  
    }  
}
```

If the state is "1", we get the element with `id="2"` (`output`) and set it to `checked`, to check the checkbox.

```
document.getElementById(output).checked = true;
```

We also need to update the state text to ON. Get the element with the `id="2s"` (`output+s`) and update the text to ON.

```
document.getElementById(output+s).innerHTML = "ON";
```

A similar process is done when the state is "0".

```
else {
    document.getElementById(output).checked = false;
    document.getElementById(output+s).innerHTML = "OFF";
}
```

The `toggleCheckBox()` function sends a message using the WebSocket connection whenever a switch is toggled on the web page. The message contains the GPIO number we want to control (`element.id` corresponds to the id of the slider switch that corresponds to the GPIO number):

```
function toggleCheckbox (element) {
    console.log(element.id);
    websocket.send(element.id);
```

Additionally, we also update the current GPIO state on the web page:

```
// Send Requests to Control GPIOs
function toggleCheckbox (element) {
    console.log(element.id);
    websocket.send(element.id);
    if (element.checked){
        document.getElementById(element.id+s).innerHTML = "ON";
    }
    else {
        document.getElementById(element.id+s).innerHTML = "OFF";
    }
}
```

Then, the ESP32 or ESP8266 should handle what happens when it receives these messages – turn the corresponding GPIOs on or off and notify all clients.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The *platformio.ini* configuration file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
arduino-libraries/Arduino_JSON @ 0.1.0
```

platformio.ini file (ESP8266)

The *platformio.ini* configuration file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
arduino-libraries/Arduino_JSON @ 0.1.0
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
// Import required libraries
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

// Create AsyncWebServer object on port 80
```

```

AsyncWebServer server(80);

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set number of outputs
#define NUM_OUTPUTS 4

// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {
            notifyClients(getOutputStates());
        }
        else{
            int gpio = atoi((char*)data);
        }
    }
}

```

```

        digitalWrite(gpio, !digitalRead(gpio));
        notifyClients(getOutputStates());
    }
}
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

void setup(){
    // Serial port for debugging purposes
    Serial.begin(115200);
    // Set GPIOs as outputs
    for (int i =0; i<NUM_OUTPUTS; i++){
        pinMode(outputGPIOs[i], OUTPUT);
    }
    initSPIFFS();
    initWiFi();
    initWebSocket();

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/index.html", "text/html", false);
    });

    server.serveStatic("/", SPIFFS, "/");
}

// Start server
server.begin();
}

void loop() {
    ws.cleanupClients();
}

```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
// Import required libraries
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>
#include <Arduino_JSON.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID ";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set number of outputs
#define NUM_OUTPUTS 4

// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize LittleFS
void initFS() {
  if (!LittleFS.begin()) {
    Serial.println("An error has occurred while mounting LittleFS");
  }
  Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
```

```

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

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {
            notifyClients(getOutputStates());
        }
        else{
            int gpio = atoi((char*)data);
            digitalWrite(gpio, !digitalRead(gpio));
            notifyClients(getOutputStates());
        }
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

```

```

    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

    // Set GPIOs as outputs
    for (int i =0; i<NUM_OUTPUTS; i++){
        pinMode(outputGPIOs[i], OUTPUT);
    }
    initFS();
    initWiFi();
    initWebSocket();

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html", false);
    });

    server.serveStatic("/", LittleFS, "/");
}

// Start server
server.begin();
}

void loop() {
    ws.cleanupClients();
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How The Code Works

The code is very similar to previous projects. So, we'll just take a look at the relevant parts for this WebSocket tutorial.

Set up Outputs

The code is prepared to control GPIOs 2, 4, 12, and 14. You can modify the `NUM_OUTPUTS` and `outputGPIOs` variables to change the number of GPIOs and which ones you want to control.

The `NUM_OUTPUTS` variable defines the number of GPIOs. The `outputGPIOs` is an array with the GPIO numbers you want to control.

```
// Set number of outputs
#define NUM_OUTPUTS 4

// Array with outputs you want to control
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};
```

Note: if you change the number of GPIOs and the GPIOs you want to control, you must also change the ids of the HTML elements in your `index.html` document.

JSON String with Current Output States

The `getOutputStates()` function checks the state of all your GPIOs and returns a JSON string variable with that information. This was already explained in Unit 2.2.

```
// Return JSON with Current Output States
String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpions"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpions"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    Serial.print(jsonString);
    return jsonString;
}
```

Notify All Clients

The `notifyClients()` function notifies all clients with a message containing whatever you pass as an argument. In this case, we want to notify all clients of the current state of all GPIOs whenever there's a change.

```
void notifyClients(String state) {
    ws.textAll(state);
}
```

The `AsyncWebSocket` class provides a `textAll()` method for sending the same message to all clients that are connected to the server at the same time.

Handle WebSocket Messages

The `handleWebSocketMessage()` function is a callback function that will run whenever we receive new data from the clients via the WebSocket protocol. As explained previously, the client will send the "states" message to request the current GPIO states or a message containing the GPIO number to change the state.

```
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info-
>opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {
            notifyClients(getOutputStates());
        }
        else{
            int gpio = atoi((char*)data);
            digitalWrite(gpio, !digitalRead(gpio));
            notifyClients(getOutputStates());
        }
    }
}
```

If we receive the "states" message, send a message to all clients with the state of all GPIOs using the `notifyClients` function. Calling the `getOutputStates()` function returns a JSON string with the GPIO states.

```
if (strcmp((char*)data, "states") == 0) {
    notifyClients(getOutputStates());
}
```

If the message is not "states", it means we've received a GPIO number and we want to toggle its state. That's what we do in the following lines.

```
else{
```

```
int gpio = atoi((char*)data);
digitalWrite(gpio, !digitalRead(gpio));
notifyClients(getOutputStates());
}
```

We save the GPIO number in the `gpio` variable.

```
int gpio = atoi((char*)data);
```

And we invert its current state:

```
digitalWrite(gpio, !digitalRead(gpio));
```

Finally, notify all clients of the change. This way, all clients are notified when there's a change and update the interface accordingly.

```
notifyClients(getOutputStates());
```

Configure the WebSocket server

Now we need to configure an event listener to handle the different asynchronous steps of the WebSocket protocol. This event handler can be implemented by defining the `onEvent()` as follows:

```
void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}
```

The `type` argument represents the event that occurs. It can take the following values:

- `WS_EVT_CONNECT` when a client has logged in;
- `WS_EVT_DISCONNECT` when a client has logged out;
- `WS_EVT_DATA` when a data packet is received from the client;
- `WS_EVT_PONG` in response to a ping request;
- `WS_EVT_ERROR` when an error is received from the client.

Initialize WebSocket

Finally, the `initWebSocket()` function initializes the WebSocket protocol.

```
void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
```

Set GPIOs as Outputs

In the `setup()`, set all your GPIOs as outputs. This will automatically get all GPIOs in the `outputGPIOs` array variable and set them as outputs.

```
for (int i =0; i<NUM_OUTPUTS; i++){
    pinMode(outputGPIOs[i], OUTPUT);
}
```

Handle Requests

The following lines handle what happens when you receive a request on the root (/) URL (ESP IP address).

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

When it receives that request, it sends the HTML text saved in the `index.html` file to build the web page. We're using SPIFFS for the ESP32 and LittleFS for the ESP8266.

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

```
request->send(LittleFS, "/index.html", "text/html", false);  
});
```

When the HTML file loads in your browser, it will make a request for the CSS, JavaScript, and favicon files. These are static files saved in the same directory (SPIFFS or LittleFS). So, we can add the following line to serve files in a directory when requested by the root URL. It will serve the CSS, JavaScript, and favicon files automatically.

```
server.serveStatic("/", SPIFFS, "/");
```

Finally, start the server.

```
server.begin();
```

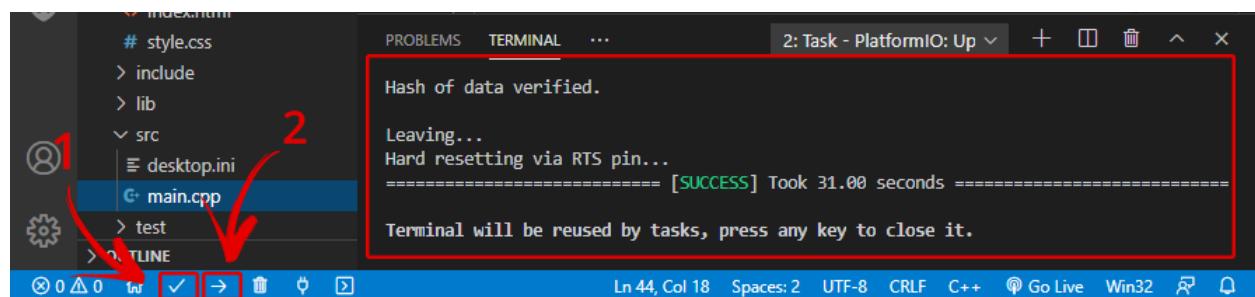
cleanupClients()

In the `loop()`, call the `cleanupClients()` method.

```
void loop() {  
    ws.cleanupClients();  
}
```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.

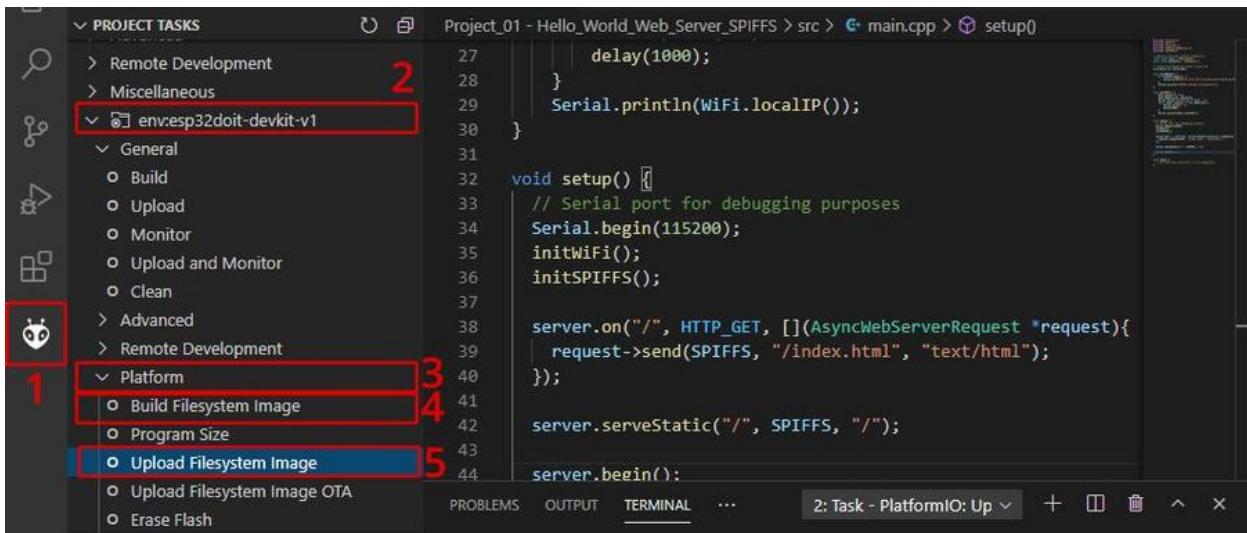


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

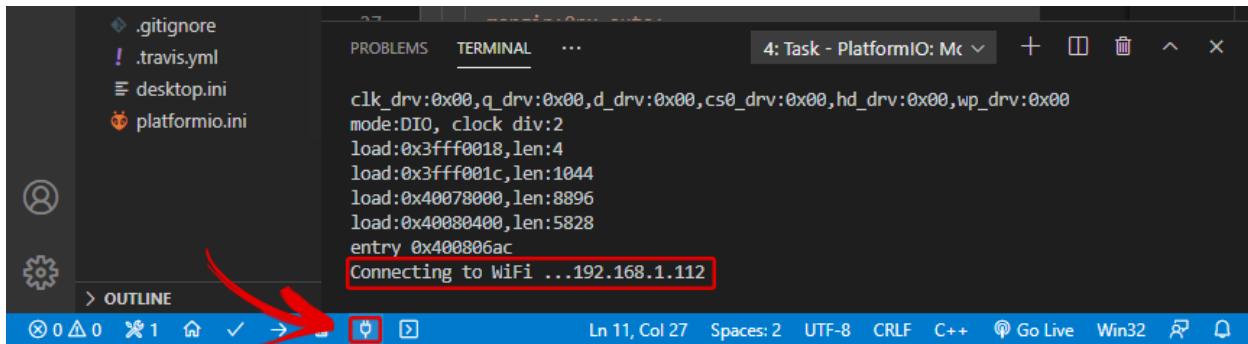
Finally, upload the files (*index.html*, *style.css*, *script.js* and *favicon.png*) to the filesystem:

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.



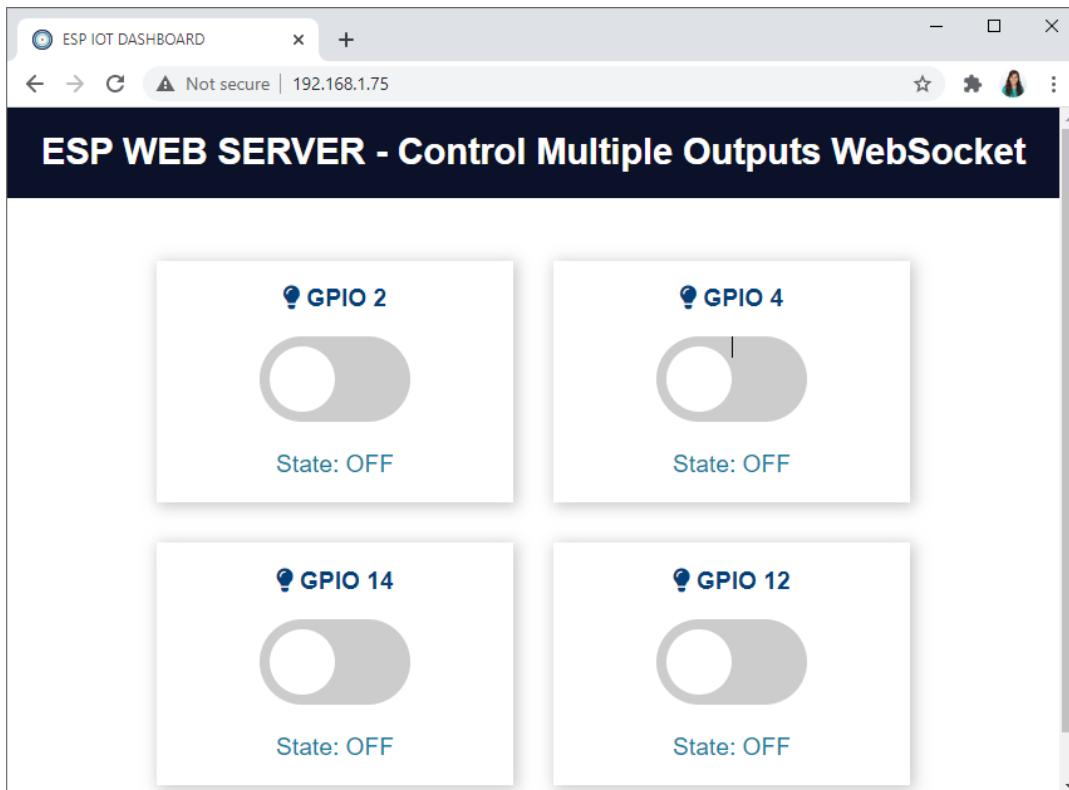
Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous example, it will probably have the same IP.



```
PROBLEMS TERMINAL ...
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5828
entry 0x400806ac
Connecting to WiFi ...192.168.1.112
```

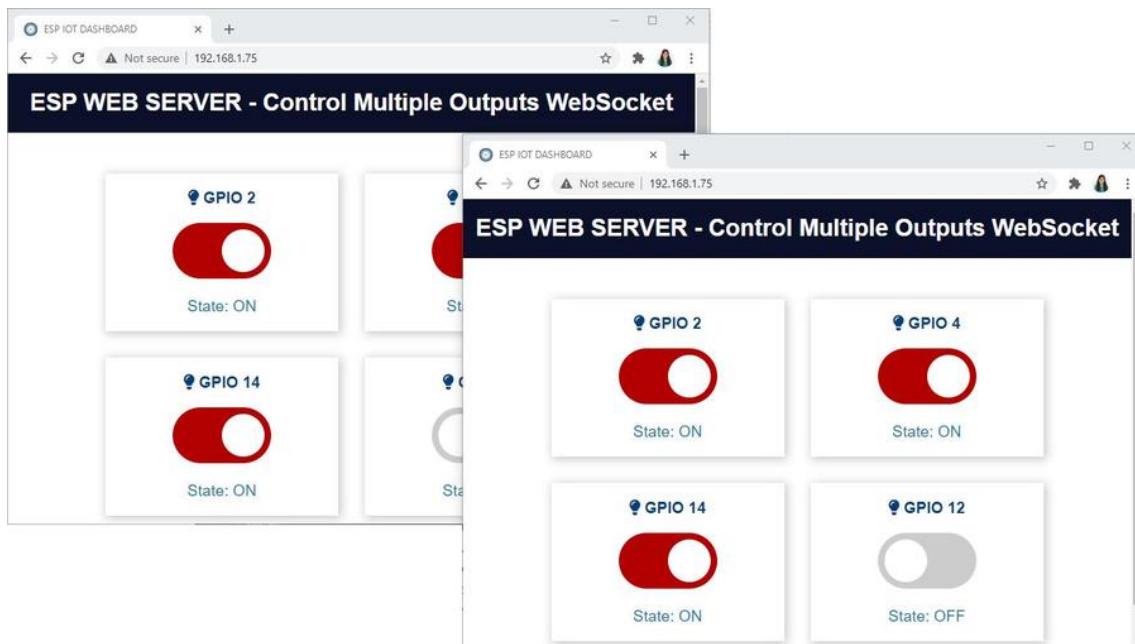
After uploading the code, access your web server by typing the ESP IP address in your browser. This is what you should see. Toggle the buttons to control the ESP32/ESP8266 GPIOs.



Here's how the web page looks like on your smartphone.



The web server looks the same as Unit 2.2, but it communicates via the WebSocket protocol. So, it has different features. Click on the buttons to control the ESP32 or ESP8266 outputs. You can open several web browser tabs at the same time or access the web server on different devices simultaneously, and the states will be updated automatically in all clients whenever there's a change—this doesn't happen with the web server of Unit 2.2.



Download Project Folder

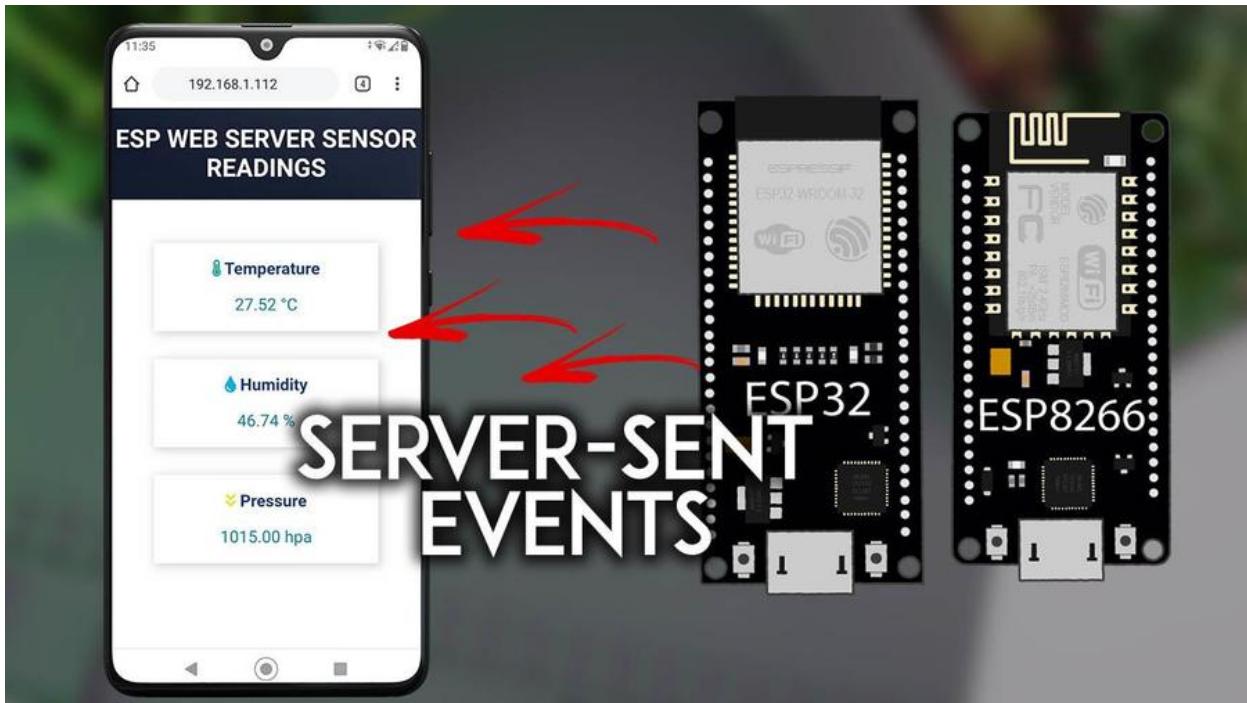
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit you've learned how to use the WebSocket protocol to control multiple ESP32 or ESP8266 GPIOs. The advantage of using the WebSocket protocol over HTTP requests is that the server can send messages to the client, and the client can send messages to the server at any given time without being requested. This was a more advanced project that combines Units 2.2 and 2.3.

3.1 - Web Server: Display Sensor Readings (SSE)

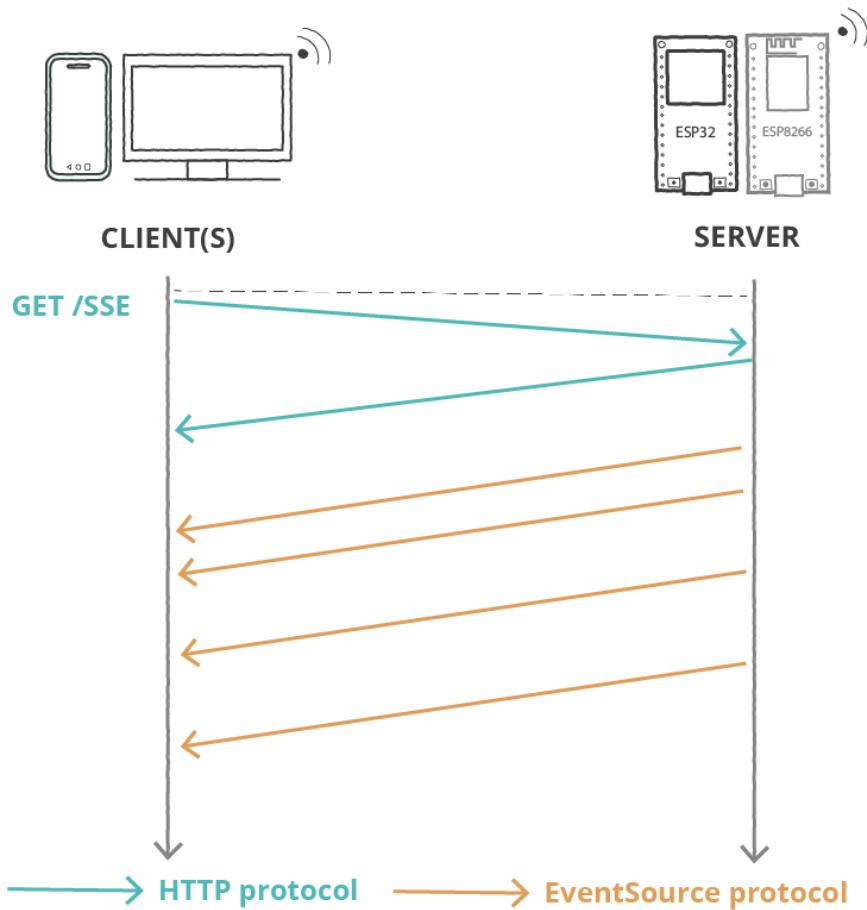


In this project, you'll learn how to create a web server to display sensor readings. As an example, we'll build a web server that displays sensor readings from a BME280 temperature, humidity, and pressure sensor. You can use any other sensor that you have. We'll use Server-Sent Events (SSE) to send the readings from the server (ESP board) to the client (browser).

SSEs allow the browser to receive automatic updates from a server via HTTP connection. This is useful to send updated sensor readings to the browser. Whenever a new reading is available, the ESP32 or ESP8266 sends it to the client, and the web page can be updated automatically without the need for further requests.

Introducing Server-Sent Events (SSE)

A Server-Sent Event (SSE) allows the client to receive automatic updates from a server via HTTP connection. The client initiates the SSE connection, and the server uses the event source API to send updates to the client. The client will receive updates from the server, but it can't send any data to the server after the initial handshake.

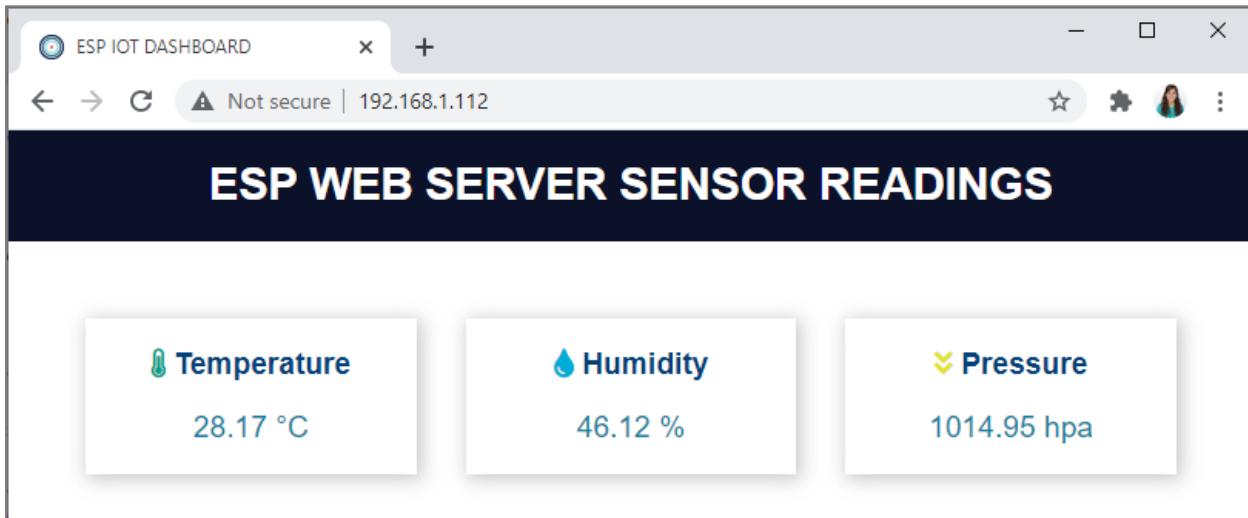


This is useful to send updated sensor readings to the browser. Whenever a new reading is available, the ESP sends it to the client, and the web page can be updated automatically without the need for further requests. Instead of sensor readings, you can send any data that might be useful for your project like GPIO states, notifications when motion is detected, etc.

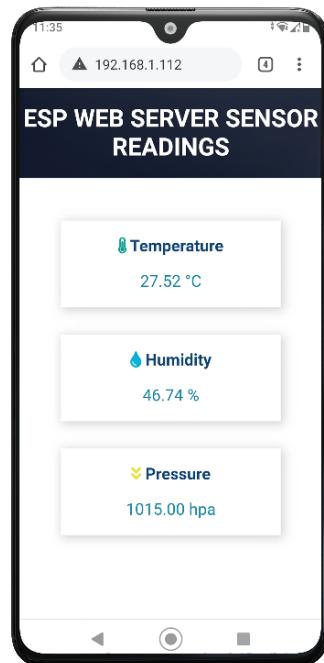
Important: Server-Sent Events (SSE) are not supported in Internet Explorer.

Project Overview

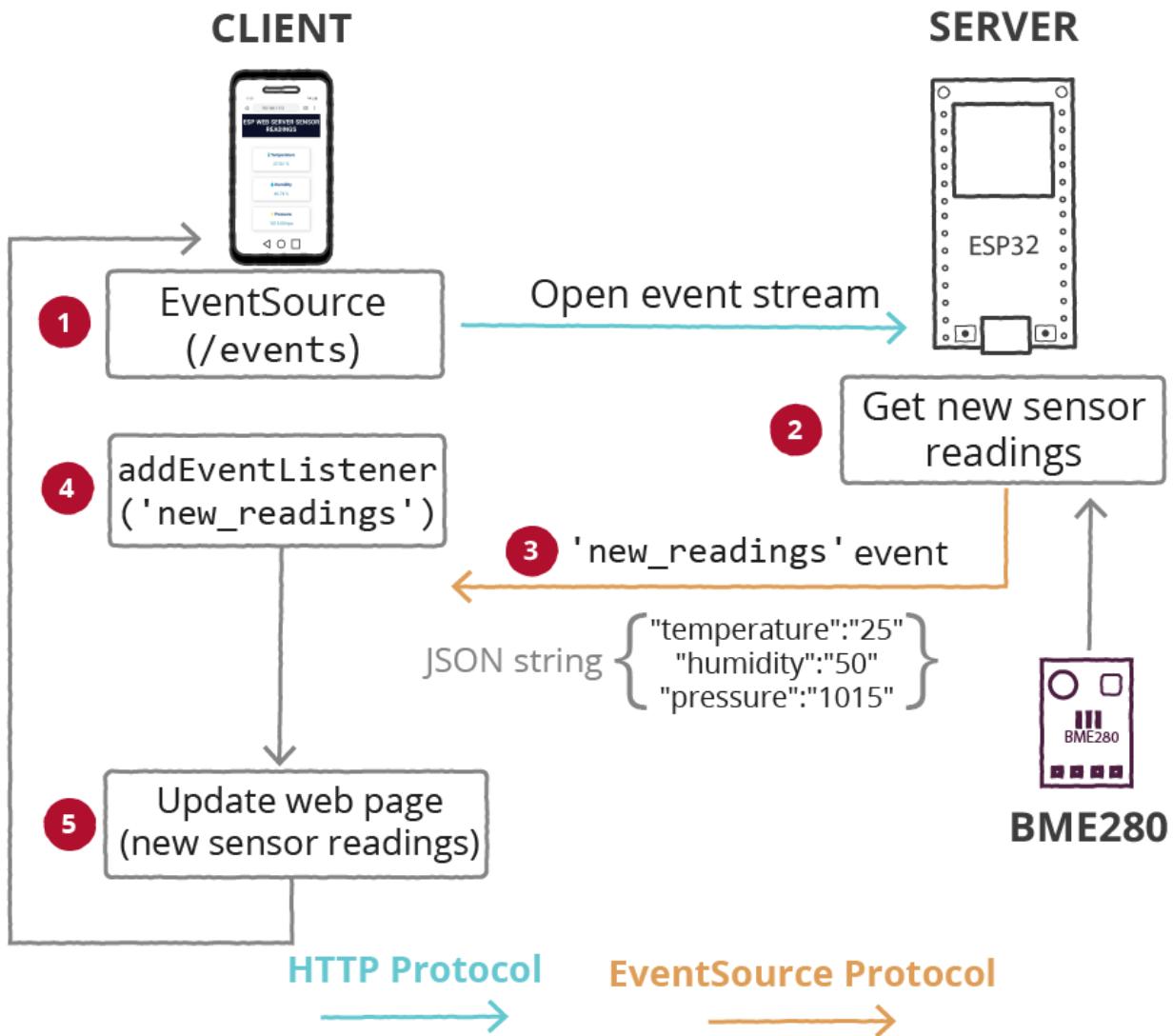
Here's the web page we'll build for this project.



The web page uses the same structure and styles from previous projects. Each card contains a paragraph to display the sensor readings—it is similar to the paragraph that displays the GPIO state in previous projects.



How it Works?



1. The client initiates the SSE connection and the server uses the event source on the `/events` URL to send updates to the client;
2. The ESP32 or ESP8266 gets new sensor readings;
3. It sends the readings as a JSON string on an event named `"new_readings"`;
4. The client has an event listener for that event and receives the updated sensor readings as a JSON string;
5. The client (browser) updates the web page with the newest readings.

Assembling the Circuit

For this example, you need to wire a BME280 sensor to your board.

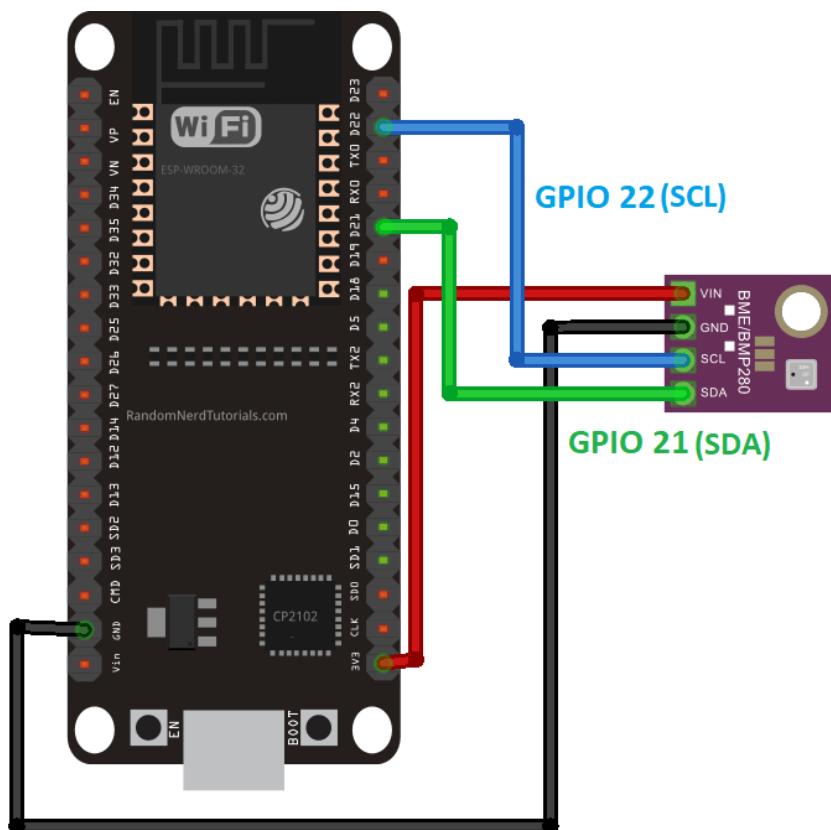
Parts Required

To complete this project you need the following parts:

- [BME280 sensor module](#)
- [ESP32 or ESP8266](#)
- [Breadboard](#)
- [Jumper wires](#)

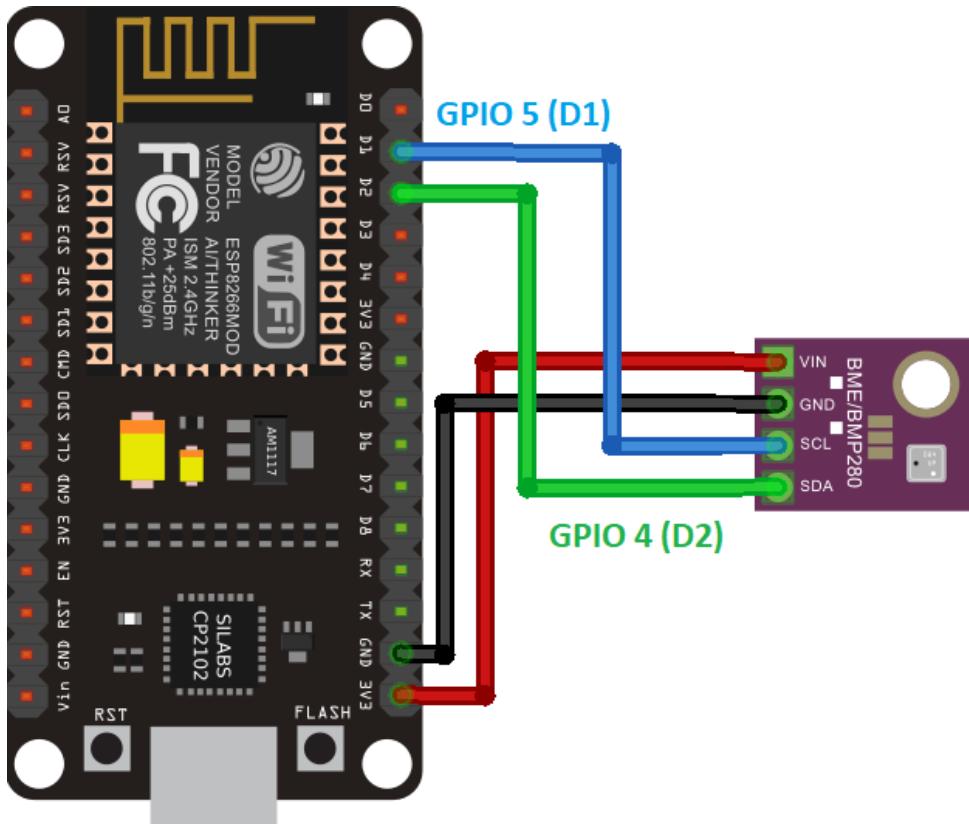
ESP32 - Schematic Diagram

Wire the sensor to the ESP32 default SDA (GPIO 21) and SCL (GPIO 22) pins, as shown in the following schematic diagram.



ESP8266 - Schematic Diagram

Follow the next schematic diagram if you're using an ESP8266. GPIO 5 and GPIO 4 are the ESP8266 I2C pins.



Building the Web Page

To build the web page for this project, place the following files inside the *data* folder within your project folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*

HTML File

Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fmnOCqbTlWIj8LyTjo7mOUSTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
    <div class="topnav">
        <h1>ESP WEB SERVER SENSOR READINGS</h1>
    </div>
    <div class="content">
        <div class="card-grid">
            <div class="card">
                <p class="card-title"><i class="fas fa-thermometer-three-quarters" style="color:#059e8a;"></i> Temperature</p>
                <p class="reading"><span id="temp"></span> &deg;C</p>
            </div>
            <div class="card">
                <p class="card-title"><i class="fas fa-tint" style="color:#00add6;"></i> Humidity</p>
                <p class="reading"><span id="hum"></span> &percnt;</p>
            </div>
            <div class="card">
                <p class="card-title"><i class="fas fa-angle-double-down" style="color:#e1e437;"></i> Pressure</p>
                <p class="reading"><span id="pres"></span> hpa</p>
            </div>
        </div>
        <script src="script.js"></script>
    </body>
</html>
```

Each card contains a paragraph to display the sensor readings.

```
<div class="card">
    <p class="card-title"><i class="fas fa-thermometer-three-quarters" style="color:#059e8a;"></i> Temperature</p>
    <p class="reading"><span id="temp"></span> &deg;C</p>
</div>
```

The second paragraph is where we'll display the actual sensor readings.

```
<p class="reading"><span id="temp"></span> &deg;C</p>
```

The readings change dynamically, so we need to add a `` tag with a specific `id`, so that we're able to manipulate that part of the paragraph to display the readings.

The id for the temperature is "temp".

```
<span id="temp"></span>
```

For the humidity, the id is "hum":

```
<span id="hum"></span>
```

Finally, the id for the pressure is "pres".

```
<span id="pres"></span>
```

Then, we'll add the sensor readings between those ` ` tags using JavaScript later on.

CSS File

Copy the following styles to your `style.css` file.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  display: inline-block;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
}
.content {
  padding: 50px;
}
```

```

.card-grid {
  max-width: 800px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
  background-color: white;
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
  font-size: 1.2rem;
  font-weight: bold;
  color: #034078
}
.reading {
  font-size: 1.2rem;
  color: #1282A2;
}

```

We're formatting the web page with the same styles from previous projects. So, if you want to learn how these styles work, go back to previous Units.

JavaScript File

Copy the following code to your *script.js* file. This is responsible for: initializing the event source stream; adding an event listener for the `new_readings` event; getting the latest sensor readings from the `new_readings` event and placing them in the right places on the web page; making an HTTP request for the current sensor readings when you access the web page for the first time.

```

// Get current sensor readings when the page loads
window.addEventListener('load', getReadings);

// Function to get current readings on the web page when it loads for the first time
function getReadings() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var myObj = JSON.parse(this.responseText);
      console.log(myObj);
      document.getElementById("temp").innerHTML = myObj.temperature;
      document.getElementById("hum").innerHTML = myObj.humidity;
      document.getElementById("pres").innerHTML = myObj.pressure;
    }
  }
}

```

```
};

xhr.open("GET", "/readings", true);
xhr.send();
}

// Create an Event Source to listen for events
if (!!window.EventSource) {
  var source = new EventSource('/events');

  source.addEventListener('open', function(e) {
    console.log("Events Connected");
  }, false);

  source.addEventListener('error', function(e) {
    if (e.target.readyState != EventSource.OPEN) {
      console.log("Events Disconnected");
    }
  }, false);

  source.addEventListener('new_readings', function(e) {
    console.log("new_readings", e.data);
    var obj = JSON.parse(e.data);
    document.getElementById("temp").innerHTML = obj.temperature;
    document.getElementById("hum").innerHTML = obj.humidity;
    document.getElementById("pres").innerHTML = obj.pressure;
  }, false);
}
```

Let's take a look at this JavaScript code to see how it works.

Get Readings

When you access the web page for the first time, we'll make a request to the server to get the current sensor readings. Otherwise, we would have to wait for new sensor readings to arrive (via Server-Sent Events), which can take some time depending on the interval that you set on the server.

Add an event listener that calls the `getReadings` function when the web page loads.

```
// Get current sensor readings when the page loads
window.addEventListener('load', getReadings);
```

The `window` object represents an open window in a browser. The `addEventListener()` method sets up a function to be called when a particular event happens. In this case, we'll call the `getReadings` function when the pages loads ('`load`') to get the current sensor readings.

Now, let's take a look at the `getReadings` function. You already know how to make requests using JavaScript. Create a new `XMLHttpRequest` object. Then, send a GET request to the server on the `/readings` URL using the `open()` and `send()` methods.

```
function getReadings() {  
  var xhr = new XMLHttpRequest();  
  xhr.open("GET", "/readings", true);  
  xhr.send();  
}
```

When we send that request, the ESP will send a response with the required information. So, we need to be able to handle what happens when we receive the response. We'll use the `onreadystatechange` property that defines a function to be executed when the `readyState` property changes. The `readyState` property holds the status of the `XMLHttpRequest`. The response of the request is ready when the `readyState` is 4 and the `status` is 200.

- `readyState = 4` means that the request finished and the response is ready;
- `status = 200` means "OK"

So, the request should look something like this:

```
function getReadings(){  
  var xhr = new XMLHttpRequest();  
  xhr.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
      ... DO WHATEVER YOU WANT WITH THE RESPONSE ...  
    }  
  };  
  xhr.open("GET", "/readings", true);  
  xhr.send();  
}
```

The response to the request is the following text (or similar) in JSON format.

```
{  
  "temperature" : "25",  
  "humidity" : "50",  
  "pressure" : "1015"  
}
```

We need to convert the JSON string into a JSON object using the `parse()` method. The result is saved in the `myObj` variable.

```
var myObj = JSON.parse(this.responseText);
```

So, we can get the temperature with `myObj.temperature`, the humidity with `myObj.humidity`, and the pressure with `myObj.pressure`.

The following lines put the received data into the elements with the corresponding ids ("temp", "hum" and "pres") on the web page.

```
document.getElementById("temp").innerHTML = myObj.temperature;
document.getElementById("hum").innerHTML = myObj.humidity;
document.getElementById("pres").innerHTML = myObj.pressure;
```

Here's the complete `getReadings()` function.

```
// Function to get current readings on the web page when it loads for the first time
function getReadings(){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            document.getElementById("temp").innerHTML = myObj.temperature;
            document.getElementById("hum").innerHTML = myObj.humidity;
            document.getElementById("pres").innerHTML = myObj.pressure;
        }
    };
    xhr.open("GET", "/readings", true);
    xhr.send();
}
```

Handle Events

Now, we need to handle the events sent by the server (Server-Sent Events).

Create a new `EventSource` object and specify the URL of the page sending the updates. In our case, it's `/events`.

```
if (!!window.EventSource) {
    var source = new EventSource('/events');
```

Once you've instantiated an event source, you can start listening for messages from the server with `addEventListener()`.

These are the default event listeners, as shown in the [AsyncWebServer documentation](#).

```
source.addEventListener('open', function(e) {
    console.log("Events Connected");
}, false);
source.addEventListener('error', function(e) {
    if (e.target.readyState != EventSource.OPEN) {
        console.log("Events Disconnected");
    }
}, false);
```

Then, add an event listener for the '`'new_readings'`' event.

```
source.addEventListener('new_readings', function(e) {
    console.log("new_readings", e.data);
    var obj = JSON.parse(e.data);
    document.getElementById("temp").innerHTML = obj.temperature;
    document.getElementById("hum").innerHTML = obj.humidity;
    document.getElementById("pres").innerHTML = obj.pressure;
}, false);
```

When new readings are available, the ESP sends an event (`'new_readings'`) to the client with a JSON string that contains the sensor readings.

The following line prints the content of the message on the console:

```
console.log("new_readings", e.data);
```

Then, convert the data into a JSON object with the `parse()` method and save it in the `obj` variable.

```
var obj = JSON.parse(e.data);
```

The JSON string comes in the following format:

```
{
  "temperature" : "25",
  "humidity" : "50",
  "pressure" : "1015"
}
```

You can get the temperature with `obj.temperature`, the humidity with `obj.humidity` and the pressure with `obj.pressure`.

The following lines put the received data into the elements with the corresponding ids ("temp", "hum" and "pres") on the web page.

```
document.getElementById("temp").innerHTML = obj.temperature;
document.getElementById("hum").innerHTML = obj.humidity;
document.getElementById("pres").innerHTML = obj.pressure;
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` file for the ESP32 should be like this. You need to include the Arduino JSON library to handle JSON strings; and the Adafruit BME280 and Adafruit Unified sensor libraries to interface with the BME280 sensor.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit_BME280_Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
```

platformio.ini file (ESP8266)

The `platformio.ini` file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
```

```
lib_deps = ESP Async WebServer
  arduino-libraries/Arduino_JSON @ 0.1.0
  adafruit/Adafruit BME280 Library @ ^2.1.0
  adafruit/Adafruit Unified Sensor @ ^1.1.4
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;
// Create a sensor object
Adafruit_BME280 bme; //BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
  if (!bme.begin(0x76)) {
    Serial.println("Could not find a valid BME280 sensor, check wiring!");
    while (1);
  }
}
```

```

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    Serial.begin(115200);
    initBME();
    initWiFi();
    initSPIFFS();

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

    server.serveStatic("/", SPIFFS, "/");
}

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

events.onConnect([] (AsyncEventSourceClient *client){
    if (client->lastId()) {
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
});

```

```

    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
);
server.addHandler(&events);

// Start server
server.begin();
}

void loop() {
  if ((millis() - lastTime) > timerDelay) {
    // Send Events to the client with the Sensor Readings Every 30 seconds
    events.send("ping",NULL,millis());
    events.send(getSensorReadings().c_str(),"new_readings" ,millis());
    lastTime = millis();
}
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

```

```

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

// Create a sensor object
Adafruit_BME280 bme; //BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

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

```

```

initFS();
// Web Server Root URL
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/index.html", "text/html");
});

server.serveStatic("/", LittleFS, "/");

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);
// Start server
server.begin();
}
void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 30 seconds
        events.send("ping",NULL,millis());
        events.send(getSensorReadings().c_str(),"new_readings" ,millis());
        lastTime = millis();
    }
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How The Code Works

Let's take a look at the code and see how it works to send readings to the client using the event source protocol.

Including Libraries

The `Adafruit_Sensor` and `Adafruit_BME280` libraries are needed to interface with the BME280 sensor.

```
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
```

The `WiFi`, `ESPAsyncWebServer` and `AsyncTCP` libraries are used to create the web server.

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

If you're using an ESP8266, you need to include the `ESP8266WiFi` library and the `ESPAsyncTCP` library.

You also need to include the `Arduino_JSON` library to make it easier to handle JSON strings.

```
#include <Arduino_JSON.h>
```

Network Credentials

Insert your network credentials in the following variables so that the ESP32 or ESP8266 can connect to your local network using Wi-Fi.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

AsyncWebServer and AsyncEventSource

Create an `AsyncWebServer` object on port 80.

```
AsyncWebServer server(80);
```

The following line creates a new event source on `/events`.

```
AsyncEventSource events("/events");
```

Declaring Variables

The `readings` variable is a JSON variable to hold the sensor readings in JSON format.

```
JSONVar readings;
```

The `lastTime` and the `timerDelay` variables will be used to update sensor readings every X number of seconds. As an example, we'll get new sensor readings every 30 seconds (30000 milliseconds).

You can change that delay time in the `timerDelay` variable.

```
unsigned long lastTime = 0;  
unsigned long timerDelay = 30000;
```

Create an `Adafruit_BME280` object called `bme` on the default ESP I2C pins.

```
Adafruit_BME280 bme;
```

Initialize BME280 Sensor

The following function can be called to initialize the BME280 sensor.

```
void initBME(){  
    if (!bme.begin(0x76)) {  
        Serial.println("Could not find a valid BME280 sensor, check wiring!");  
        while (1);  
    }  
}
```

Usually, the I2C address for the BME280 sensor is `0x76`. However, you may have a slightly different sensor. So, it is a good idea to check the I2C address. You can run the following I2C scanner sketch with the sensor connected to your board to find its I2C address: https://raw.githubusercontent.com/RuiSantosdotme/Random-Nerd-Tutorials/master/Projects/LCD_I2C/I2C_Scanner.ino

Get BME280 Readings

To get readings from the BME280 temperature, humidity and pressure sensor, use the following methods on the `bme` object: `bme.readTemperature()`; `bme.readHumidity()` and `bme.readPressure()`.

The `getSensorReadings()` function gets the sensor readings and saves them in the `readings` JSON array.

```
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}
```

The `readings` array is then converted into a JSON string variable using the `stringify()` method and saved in the `jsonString` variable.

The function returns the `jsonString` variable with the current sensor readings. The JSON string has the following format (the values are just arbitrary numbers for explanation purposes).

```
{
  "temperature" : "25",
  "humidity" : "50",
  "pressure" : "1015"
}
```

setup()

In the `setup()`, initialize the Serial Monitor, Wi-Fi, filesystem, and the BME280 sensor.

```
void setup() {
  Serial.begin(115200);
  initBME();
  initWiFi();
  initSPIFFS();
```

If you're using an ESP8266, initialize LittleFS instead of SPIFFS.

Handle Requests

When you access the ESP32 IP address on the root / URL, send the text that is stored in the `index.html` file to build the web page.

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

Serve the other static files requested by the client (`style.css`, `script.js`, and `favicon.png`).

```
server.serveStatic("/", SPIFFS, "/");
```

Send the JSON string with the current sensor readings when you receive a request on the `/readings` URL.

```
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});
```

The `json` variable holds the return from the `getSensorReadings()` function. To send a JSON string as response, the `send()` method accepts as first argument the response code (200), the second is the content type ("application/json") and finally the content (`json` variable).

Server Event Source

Set up the event source on the server.

```
events.onConnect([](AsyncEventSourceClient *client) {
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);
```

Finally, start the server.

```
server.begin();
```

loop()

In the `loop()`, send events to the browser with the latest sensor readings to update the web page every 30 seconds.

```
events.send("ping", NULL, millis());  
events.send(getSensorReadings().c_str(), "new_readings" , millis());
```

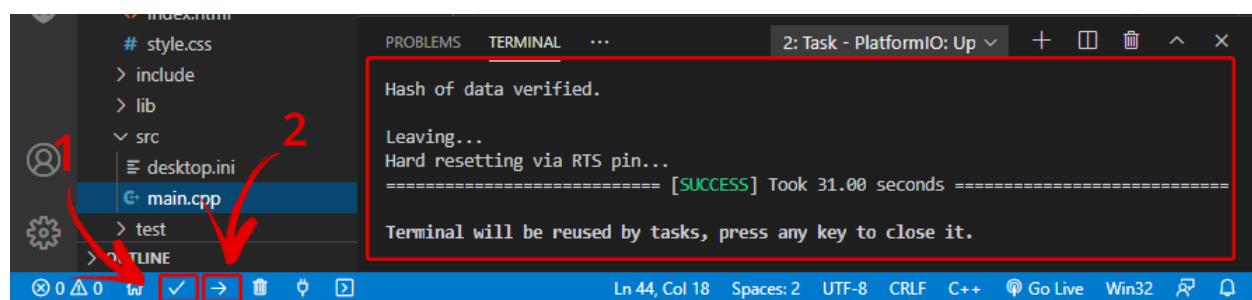
Use the `send()` method on the `events` object and pass as argument the content you want to send and the name of the event. In this case, we want to send the JSON string returned by the `getSensorReadings()` function. The `send()` method accepts a variable of type `char`, so we need to use the `c_str()` method to convert the variable. The name of the events is `new_readings`.

Usually, we also send a `ping` message every X number of seconds. That line is not mandatory. It is used to check on the client side that the server is alive.

```
events.send("ping", NULL, millis());
```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload code to your board.

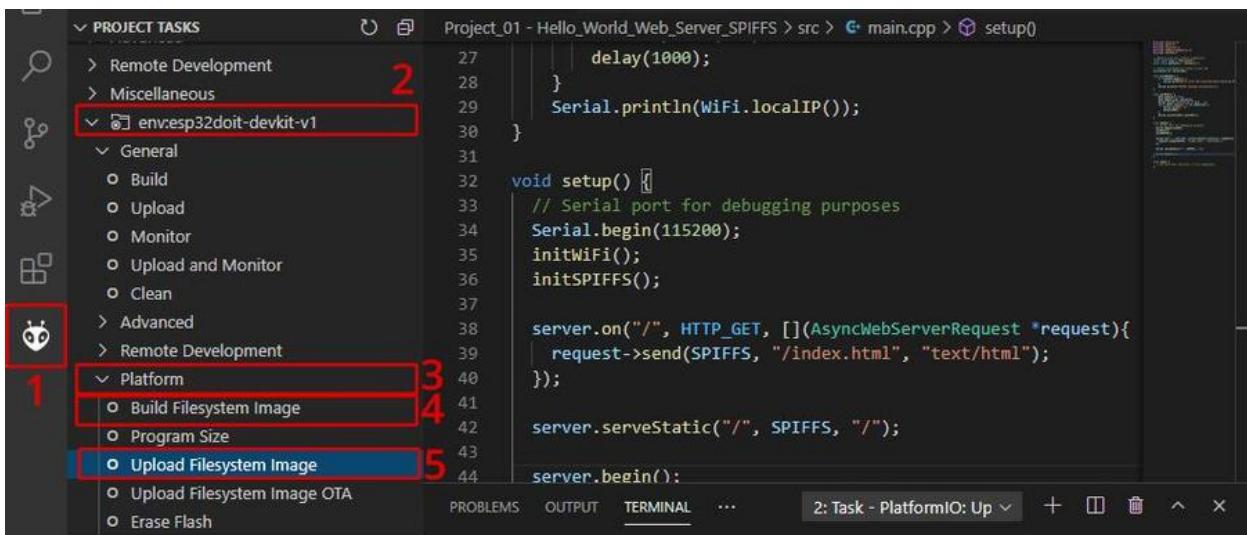


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

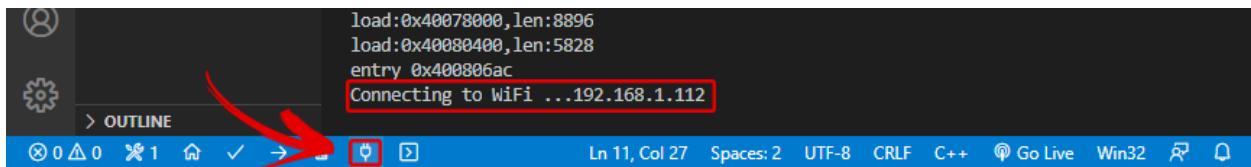
Finally, upload the files (*index.html*, *style.css*, *script.js*, and *favicon.png*) to the filesystem:

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.

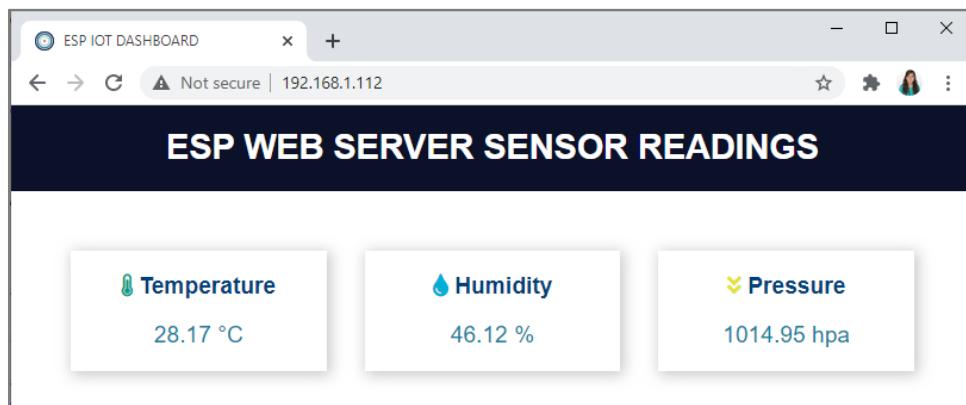


Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.

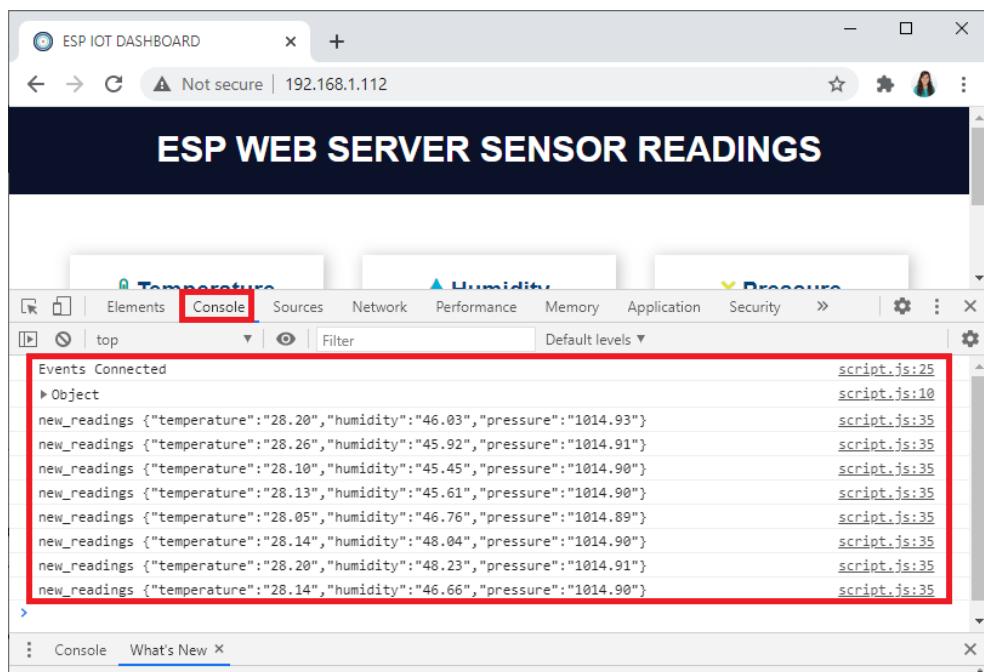


Open a browser on your local network and insert the ESP IP address. You should get access to the web page to monitor the sensor readings.



The readings are updated automatically every 30 seconds.

You can check if the client is receiving the events. On your browser, open the console by pressing **Ctrl+Shift+J** (Windows) or **Option + ⌘ + C** (Mac OS).



You should see the readings being received every 30 seconds as a JSON string.

Download Project Folder

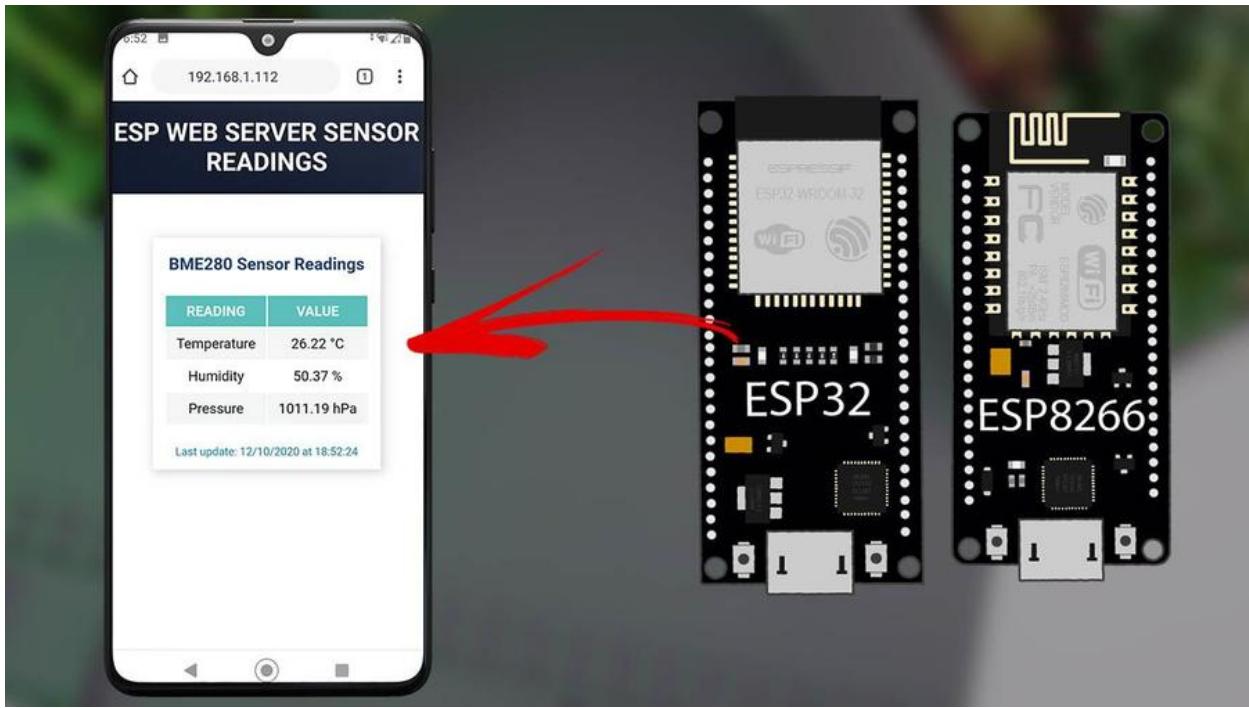
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to use Server-Sent Events with the ESP32 and ESP8266. Server-Sent Events allow a web page (client) to get updates from a server. This can automatically display new sensor readings on the web page as soon as they are available.

3.2 - Web Server: Display Sensor Readings (Table)



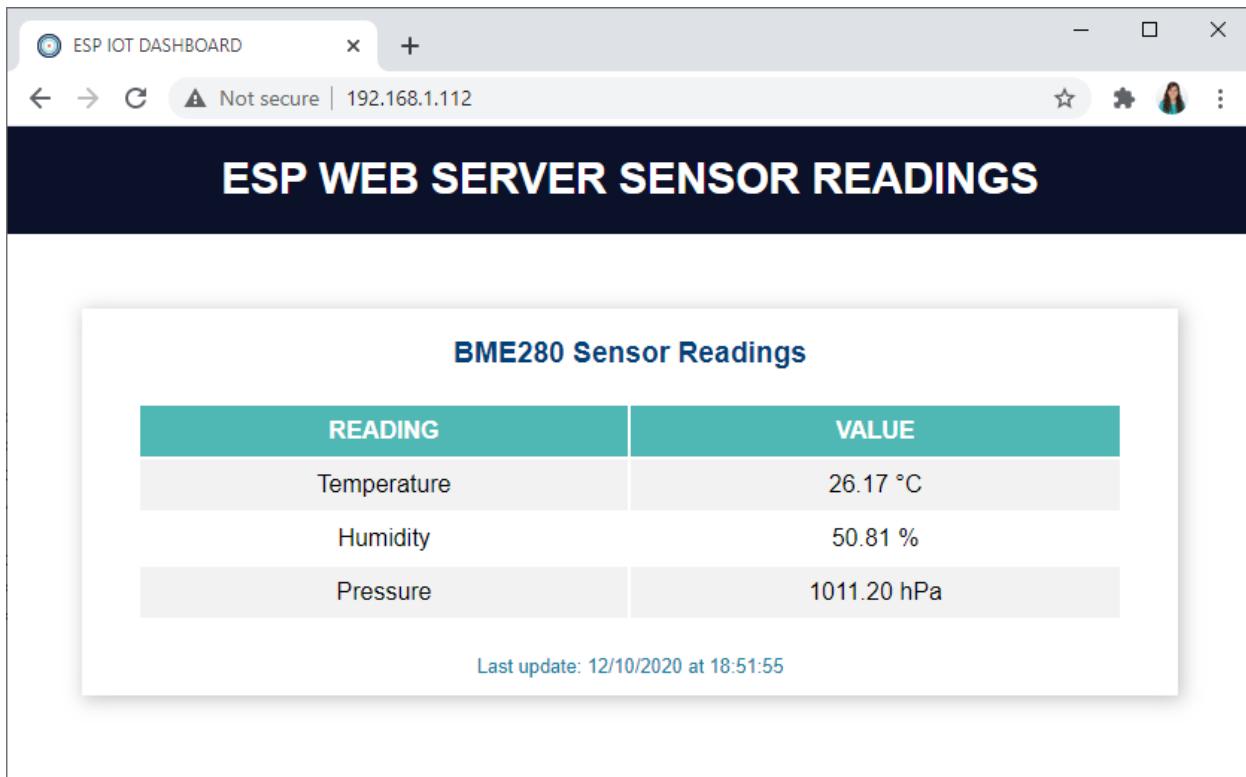
This Unit is very similar to the previous project, but you'll display the sensor readings in a table. Additionally, we'll add a paragraph that registers the last time the readings were updated (date and time).

The web page displays readings from the BME280 temperature, humidity, and pressure sensor, and we'll use Server-Sent Events to send updated readings to the client every 30 seconds. The web server code is the same as the previous Unit. By modifying the HTML, CSS, and JavaScript files, you can build a completely different interface.

With this project, you'll learn how to get and display date and time using JavaScript and style tables using CSS.

Project Overview

Here's the web page we'll build for this project.



The web page for this project uses the same structure as in previous Units. Inside the card, there's a card title, a table and a paragraph with the last time the readings were updated on the interface. We get the date and time using JavaScript.

Every time new readings are received, call a JavaScript function that gets the current date and time and places it in the right place on the web page.

Assembling the Circuit

For this example, you need to wire a BME280 sensor to your board.

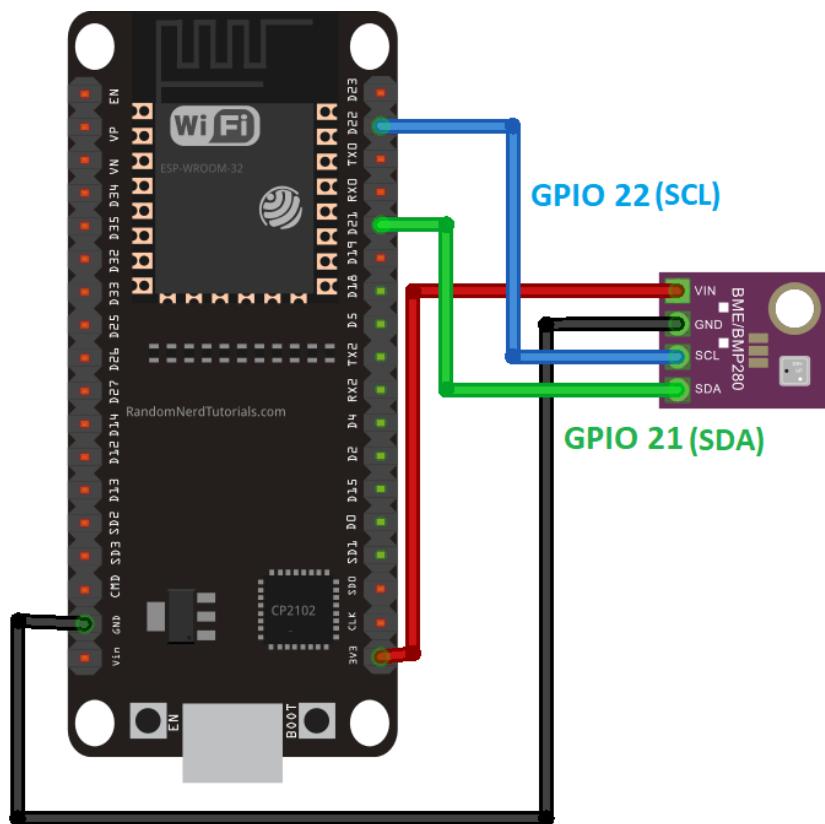
Parts Required

To complete this Unit, you need the following parts:

- [BME280 sensor module](#)
- [ESP32 or ESP8266](#)
- [Breadboard](#)
- [Jumper wires](#)

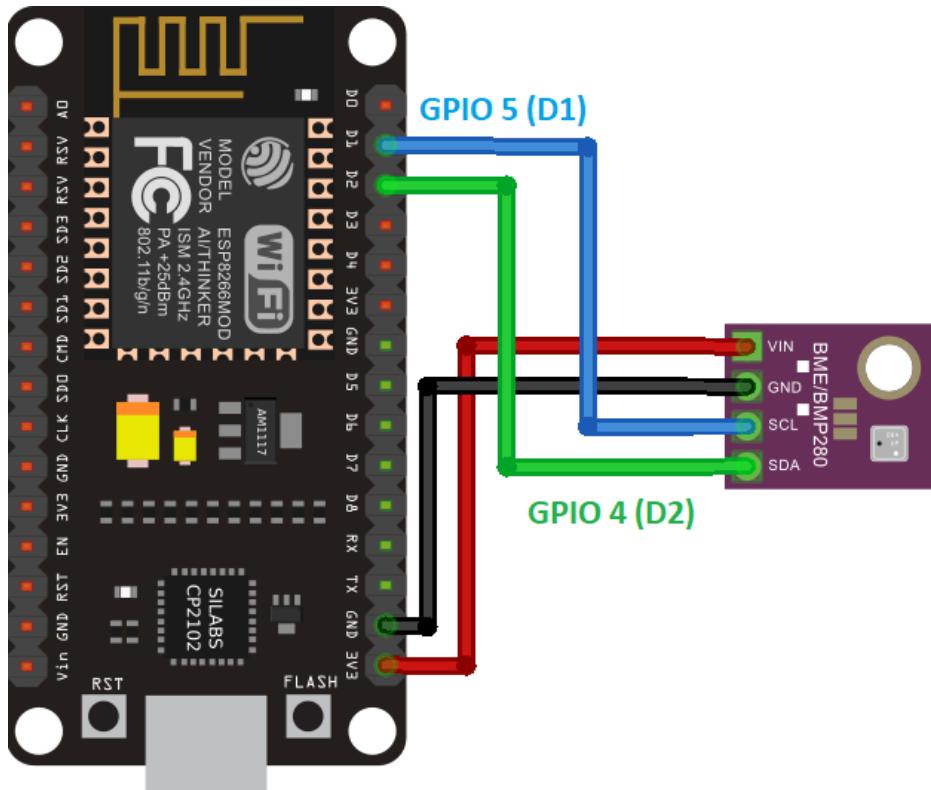
ESP32 - Schematic Diagram

Wire the sensor to the ESP32 default SDA (GPIO 21) and SCL (GPIO 22) pins, as shown in the following schematic diagram.



ESP8266 - Schematic Diagram

Follow the next schematic diagram if you're using an ESP8266. GPIO 5 and GPIO 4 are the ESP8266 I2C pins.



Building the Web Page

To build the web page for this project, place the following files in your project's *data* folder:

- *index.html*
- *style.css*
- *script.js*
- *favicon.png*

HTML File

Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/png" href="favicon.png">
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUStjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER SENSOR READINGS</h1>
  </div>
  <div class="content">
    <div class="card-grid">
      <div class="card">
        <p class="card-title">BME280 Sensor Readings</p>
        <p>
          <table>
            <tr>
              <th>READING</th>
              <th>VALUE</th>
            </tr>
            <tr>
              <td>Temperature</td>
              <td><span id="temp"></span> &deg;C</td>
            </tr>
            <tr>
              <td>Humidity</td>
              <td><span id="hum"></span> &percnt;</td>
            </tr>
            <tr>
              <td>Pressure</td>
              <td><span id="pres"></span> hPa</td>
            </tr>
          </table>
        </p>
        <p class="update-time">Last update: <span id="update-time"></span></p>
      </div>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

Let's take a look at the HTML tags to build the card with the sensor readings.

The following line displays the card title. You can change the text to whatever you want. We gave it the same class name (`card-title`) used in previous projects, so it will be styled similarly using CSS.

```
<p class="card-title">BME280 Sensor Readings</p>
```

Then, we have a paragraph with a table.

```
<table>
  <tr>
    <th>READING</th>
    <th>VALUE</th>
  </tr>
  <tr>
    <td>Temperature</td>
    <td><span id="temp"></span> &deg;C</td>
  </tr>
  <tr>
    <td>Humidity</td>
    <td><span id="hum"></span> &percnt;</td>
  </tr>
  <tr>
    <td>Pressure</td>
    <td><span id="pres"></span> hPa</td>
  </tr>
</table>
```

To create a table in HTML, start with the `<table>` and `</table>` tags. This encloses the entire table. To create a row, use the `<tr>` and `</tr>` tags. The table is defined with a series of rows. Use the `<tr></tr>` pair to enclose each row of data. The table heading is defined using the `<th>` and `</th>` tags, and each table cell is defined using the `<td>` and `</td>` tags.

Notice that the cells to display the sensor readings have `` tags with specific ids, so that we can manipulate them later using JavaScript to insert the updated readings.

Finally, there's a paragraph to display the last time the readings were updated:

```
<p class="update-time">Last update: <span id="update-time"></span></p>
```

There's a `` tag with the `update-time` id. This will be used later to insert the date and time using JavaScript.

CSS File

Copy the following styles to your `style.css` file.

```
html {
    font-family: Arial, Helvetica, sans-serif;
    display: inline-block;
    text-align: center;
}
h1 {
    font-size: 1.8rem;
    color: white;
}
p {
    font-size: 1.4rem;
}
.topnav {
    overflow: hidden;
    background-color: #0A1128;
}
body {
    margin: 0;
}
.content {
    padding: 50px;
}
.card-grid {
    max-width: 800px;
    margin: 0 auto;
    display: grid;
    grid-gap: 2rem;
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
    background-color: white;
    box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
    font-size: 1.2rem;
    font-weight: bold;
    color: #034078
}
th, td {
    text-align: center;
    padding: 8px;
}
tr:nth-child(even) {
```

```
background-color: #f2f2f2
}
tr:hover {
  background-color: #ddd;
}
th {
  background-color: #50b8b4;
  color: white;
}
table {
  margin: 0 auto;
  width: 90%
}
.update-time {
  font-size: 0.8rem;
  color: #1282A2;
}
```

We are using the same styles used in previous projects but added the necessary instructions to style the table. Let's take a quick look at them.

```
th, td {
  text-align: center;
  padding: 8px;
}
tr:nth-child(even) {
  background-color: #f2f2f2
}
tr:hover {
  background-color: #ddd;
}
th {
  background-color: #50b8b4;
  color: white;
}
table {
  margin: 0 auto;
  width: 90%
}
```

Without styles, the table looks like this:

BME280 Sensor Readings

READING	VALUE
Temperature	°C
Humidity	%
Pressure	hPa

Last update:

The following instructions set the background color for the table headings and the text color to white.

```
th {  
    background-color: #50b8b4;  
    color: white;  
}
```

BME280 Sensor Readings

READING	VALUE
Temperature	°C
Humidity	%
Pressure	hPa

Last update:

The next lines align the content of all cells, including the heading cells to the center and with 8px of padding.

```
th, td {  
    text-align: center;  
    padding: 8px;  
}
```

Set the background color of the even rows to a light shade of gray:

```
tr:nth-child(even) {  
    background-color: #f2f2f2  
}
```

The `:nth-child(n)` selector matches every element that is the *n*th child, regardless of type, of its parent. It can be a number, a keyword, or a formula. In this case, we're getting all even rows.

The `:hover` selector will change the background color of the rows when you hover your mouse over them to a slightly darker tone of gray.

```
tr:hover {  
    background-color: #ddd;  
}
```

BME280 Sensor Readings	
READING	VALUE
Temperature	°C
Humidity	%
Pressure	hPa

Last update:

To change the table width and align it at the center of its parent, set the following instructions:

```
table {  
    margin: 0 auto;  
    width: 90%  
}
```

Setting the margin to `0px auto`, sets the bottom and upper margin to `0` and `auto` to the left and right margins. Setting the left and right margins to `auto` centers the table—the browser will automatically distribute the right amount of margin to either

side. Setting the `width` to `90%` means that the table width is 90% of its parent container.

BME280 Sensor Readings	
READING	VALUE
Temperature	°C
Humidity	%
Pressure	hPa

Last update:

Finally, format the last paragraph (whose class name is `update-time`) with a slightly smaller font and with a different color:

```
.update-time {  
    font-size: 0.8rem;  
    color: #1282A2;  
}
```

Here's the final result:

BME280 Sensor Readings	
READING	VALUE
Temperature	°C
Humidity	%
Pressure	hPa

Last update:

JavaScript File

Copy the following code to your *script.js* file. This is responsible for:

- initializing the event source protocol;
- adding an event listener for the `new_readings` event;
- getting the latest sensor readings from the `new_readings` event and placing them in the right places on the web page;
- making an HTTP request for the current sensor readings when you access the web page for the first time;
- getting date and time when new readings are available.

```
// Get current sensor readings when the page loads
window.addEventListener('load', getReadings);

//Function to add date and time of last update
function updateDateTime() {
    var currdate = new Date();
    var datetime = currdate.getDate() + "/"
    + (currdate.getMonth()+1) + "/"
    + currdate.getFullYear() + " at "
    + currdate.getHours() + ":"
    + currdate.getMinutes() + ":"
    + currdate.getSeconds();
    document.getElementById("update-time").innerHTML = datetime;
    console.log(datetime);
}

// Function to get current readings on the web page when it loads at first
function getReadings() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            document.getElementById("temp").innerHTML = myObj.temperature;
            document.getElementById("hum").innerHTML = myObj.humidity;
            document.getElementById("pres").innerHTML = myObj.pressure;
            updateDateTime();
        }
    };
    xhr.open("GET", "/readings", true);
    xhr.send();
}
// Create an Event Source to listen for events
if (!!window.EventSource) {
```

```

var source = new EventSource('/events');

source.addEventListener('open', function(e) {
  console.log("Events Connected");
}, false);

source.addEventListener('error', function(e) {
  if (e.target.readyState != EventSource.OPEN) {
    console.log("Events Disconnected");
  }
}, false);

source.addEventListener('new_readings', function(e) {
  console.log("new_readings", e.data);
  var obj = JSON.parse(e.data);
  document.getElementById("temp").innerHTML = obj.temperature;
  document.getElementById("hum").innerHTML = obj.humidity;
  document.getElementById("pres").innerHTML = obj.pressure;
  updateDateTime();
}, false);
}

```

This JavaScript code is very similar to the one used in the previous Unit, but adds the lines to get date and time when new readings are available.

Notice that we call the `updateDateTime()` function after receiving the HTTP response with the current sensor readings:

```

function getReadings() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var myObj = JSON.parse(this.responseText);
      console.log(myObj);
      document.getElementById("temp").innerHTML = myObj.temperature;
      document.getElementById("hum").innerHTML = myObj.humidity;
      document.getElementById("pres").innerHTML = myObj.pressure;
      updateDateTime();
    }
  };
  xhr.open("GET", "/readings", true);
  xhr.send();
}

```

And it is also called when the `new_readings` event occurs:

```

source.addEventListener('new_readings', function(e) {
  console.log("new_readings", e.data);
  var obj = JSON.parse(e.data);
  document.getElementById("temp").innerHTML = obj.temperature;
}

```

```
document.getElementById("hum").innerHTML = obj.humidity;
document.getElementById("pres").innerHTML = obj.pressure;
updateDateTime();
}, false);
```

The `updateDateTime()` function gets the current date and time and places it in the HTML element with the `update-time` id.

```
function updateDateTime() {
    var currentdate = new Date();
    var datetime = currentdate.getDate() + "/"
+ (currentdate.getMonth()+1) + "/"
+ currentdate.getFullYear() + " at "
+ currentdate.getHours() + ":"
+ currentdate.getMinutes() + ":"
+ currentdate.getSeconds();
    document.getElementById("update-time").innerHTML = datetime;
    console.log(datetime);
}
```

First, create a JavaScript `Date` object. Date objects are created with the `new Date()` constructor. The following line creates a `Date` object called `currentdate` with the current date and time.

```
var currentdate = new Date();
```

By default, JavaScript will use the browser's time zone to display a date as a full-text string like this:

```
Tue Oct 13 2020 11:34:48 GMT+0100 (Western European Summer Time)
```

There are methods to get information from a `Date` object. We use the following methods in our example:

- `getDate()` get the day as a number (0 to 31);
- `getMonth()` get the month as a number (0 to 11). 0 is January and 11 is December. So, if you want to get the current month as a number you must add 1.
- `getFullYear()` get the year as a four digit number;
- `getHours()` get the hour (0 to 23);

- `getMinutes()` get the minute (0 to 59);
- `getSeconds()` get the seconds (0 to 59).

The following lines create a new variable called `datetime` with the current date and time in this format: DD/MM/YY at HH:MM:SS.

```
var datetime = currentdate.getDate() + "/"
+ (currentdate.getMonth()+1) + "/"
+ currentdate.getFullYear() + " at "
+ currentdate.getHours() + ":"
+ currentdate.getMinutes() + ":"
+ currentdate.getSeconds();
```

Finally, place the content of the `datetime` variable in the HTML element with the `update-time` id. This places the date and time in the right place on the HTML page.

```
document.getElementById("update-time").innerHTML = datetime;
```

For debugging purposes, you can print the date and time in the browser console.

```
console.log(datetime);
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` file for the ESP32 should be like this. You need to include the Arduino JSON library to handle JSON strings; and the Adafruit BME280 and Adafruit Unified sensor libraries to interface with the BME280 sensor.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
```

```
adafruit/Adafruit_BME280_Library @ ^2.1.0
adafruit/Adafruit Unified Sensor @ ^1.1.4
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit_BME280_Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;
```

```

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

// Create a sensor object
Adafruit_BME280 bme;           // BME280 connect to ESP32 I2C (GPIO 21 = , GPIO 22
= SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize LittleFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    initBME();
    initWiFi();
    initSPIFFS();

    // Web Server Root URL
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){

```

```

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

server.serveStatic("/", SPIFFS, "/");

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

events.onConnect([] (AsyncEventSourceClient *client){
    if (client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);

// Start server
server.begin();
}

void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 30 seconds
        events.send("ping", NULL, millis());
        events.send(getSensorReadings().c_str(), "new_readings", millis());
        lastTime = millis();
    }
}

```

This is the same code we used in the previous project. Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

// Create a sensor object
Adafruit_BME280 bme;           // BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO
22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
```

```

        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    initBME();
    initWiFi();
    initFS();

    // Web Server Root URL
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html");
    });

    server.serveStatic("/", LittleFS, "/");
}

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

events.onConnect([] (AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);

// Start server
server.begin();
}

void loop() {

```

```

if ((millis() - lastTime) > timerDelay) {
    // Send Events to the client with the Sensor Readings Every 30 seconds
    events.send("ping", NULL, millis());
    events.send(getSensorReadings().c_str(), "new_readings" , millis());
    lastTime = millis();
}

```

This is the same code we used in the previous project. Modify the code to include your network credentials, and it will work straight away.

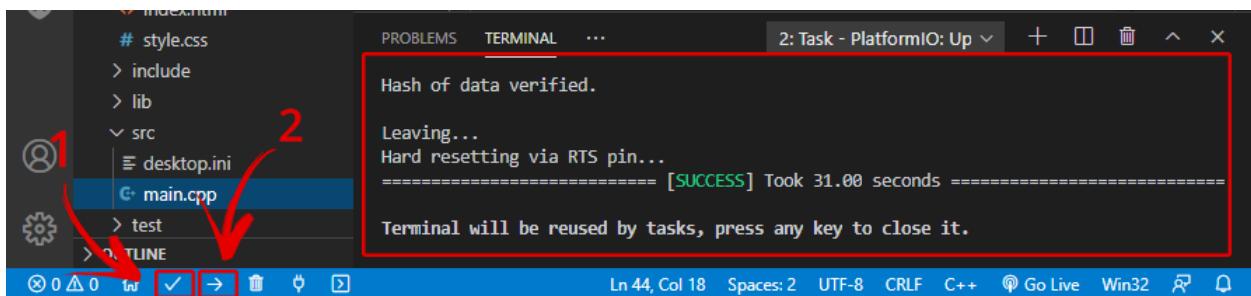
```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



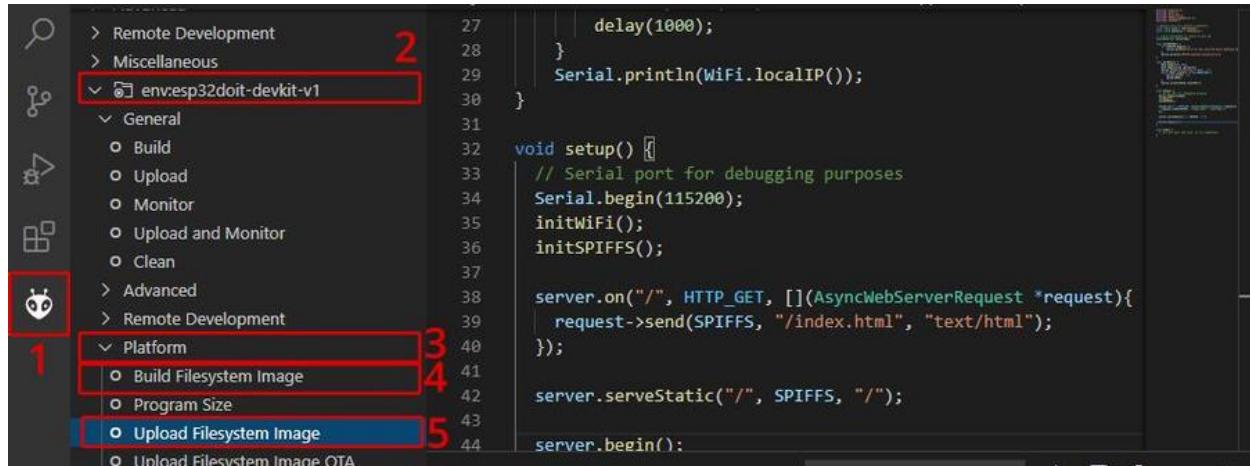
Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, upload the files (*index.html*, *style.css*, *script.js*, and *favicon.png*) to the filesystem:

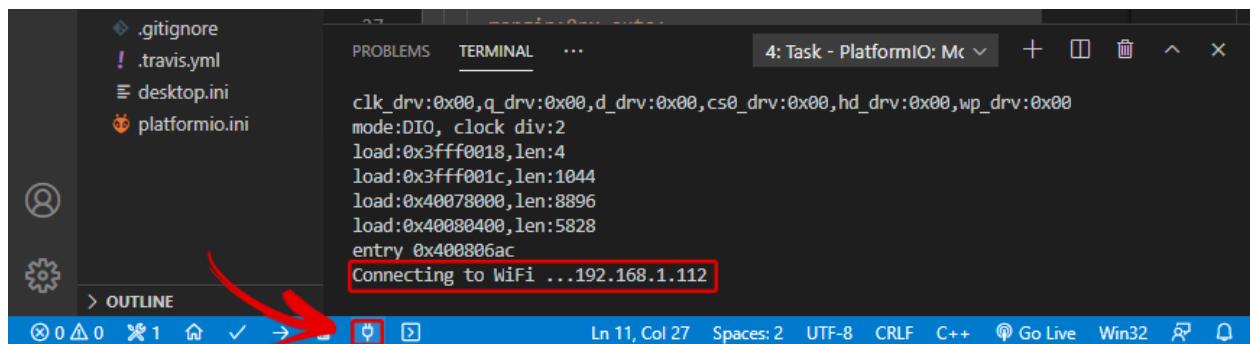
1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).

3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.



Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.



Open a browser on your local network and insert the ESP IP address. You should get access to the web page to monitor the sensor readings. Here's how the web page looks on your computer.

The screenshot shows a web browser window titled "ESP IOT DASHBOARD" with the URL "192.168.1.112". The main title is "ESP WEB SERVER SENSOR READINGS". Below it is a section titled "BME280 Sensor Readings" containing a table with three rows: Temperature (26.17 °C), Humidity (50.81 %), and Pressure (1011.20 hPa). At the bottom of this section is a timestamp: "Last update: 12/10/2020 at 18:51:55".

READING	VALUE
Temperature	26.17 °C
Humidity	50.81 %
Pressure	1011.20 hPa

You should receive new sensor readings every 30 seconds, and the last update time should change accordingly. You can press **Ctrl+Shift+J** (Windows) or **Option + ⌘ + C** (Mac OS) to monitor what is happening in the background.

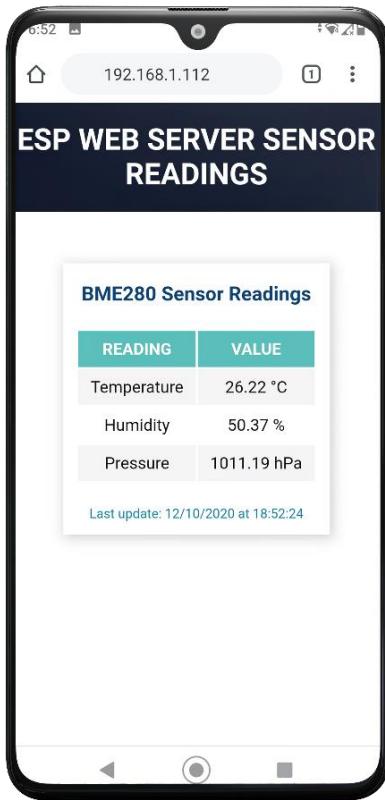
The screenshot shows the Chrome DevTools Console tab. The console output displays sensor readings from a BME280 sensor. The readings are logged at different times, showing slight changes over time. The log entries are highlighted with a red box.

```

BME280 Sensor Readings
READING VALUE
Events Connected
Object
  humidity: "60.49"
  pressure: "1009.17"
  temperature: "23.94"
  > __proto__: Object
13/10/2020 at 10:40:41
new_readings {"temperature": "23.78", "humidity": "59.32", "pressure": "1009.17"}
13/10/2020 at 10:40:59
new_readings {"temperature": "23.62", "humidity": "60.24", "pressure": "1009.16"}
13/10/2020 at 10:41:29

```

Here's what the web page looks like on your smartphone:



Download Project Folder

You can download the complete project folder for this project using the links below.

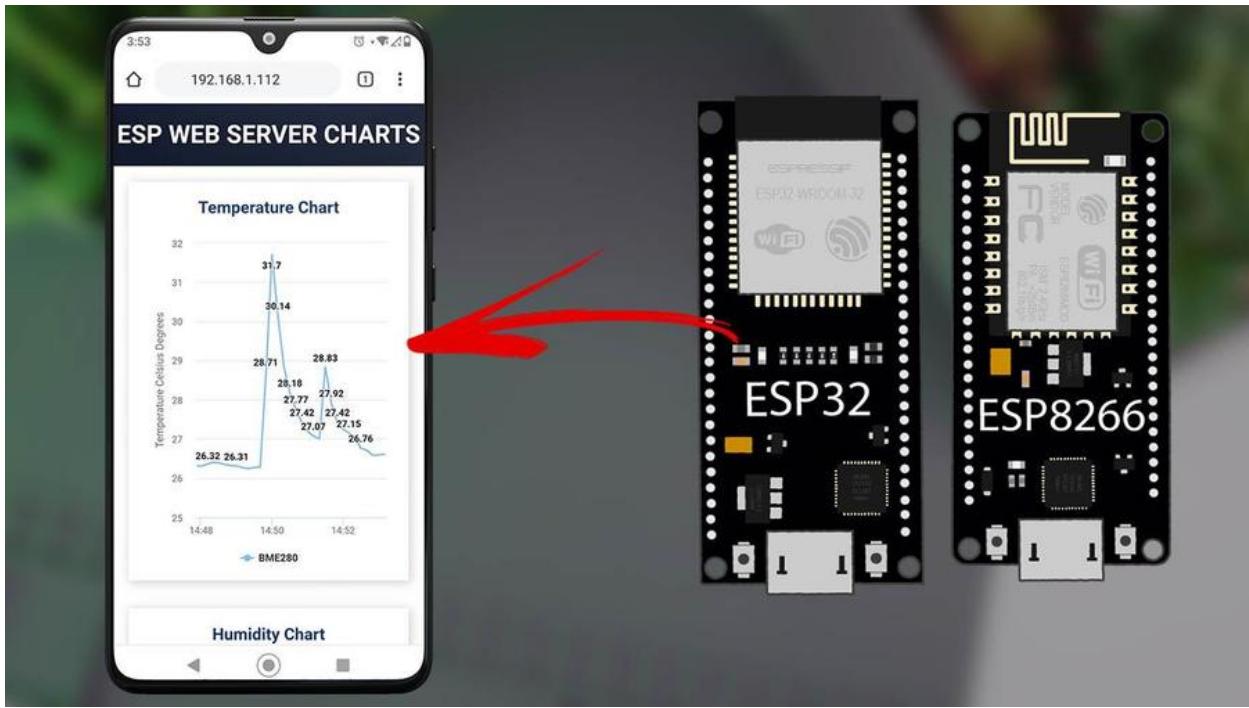
- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to display sensor readings in a table and how to use JavaScript to get the current date and time.

With the same web server code, if you change the HTML, CSS, and JavaScript files, you can have a completely different interface. That's why it is good to know some HTML, CSS, and JavaScript—so that you can change the web interface to make it look as you wish.

3.3 - Web Server: Display Sensor Readings (Charts)

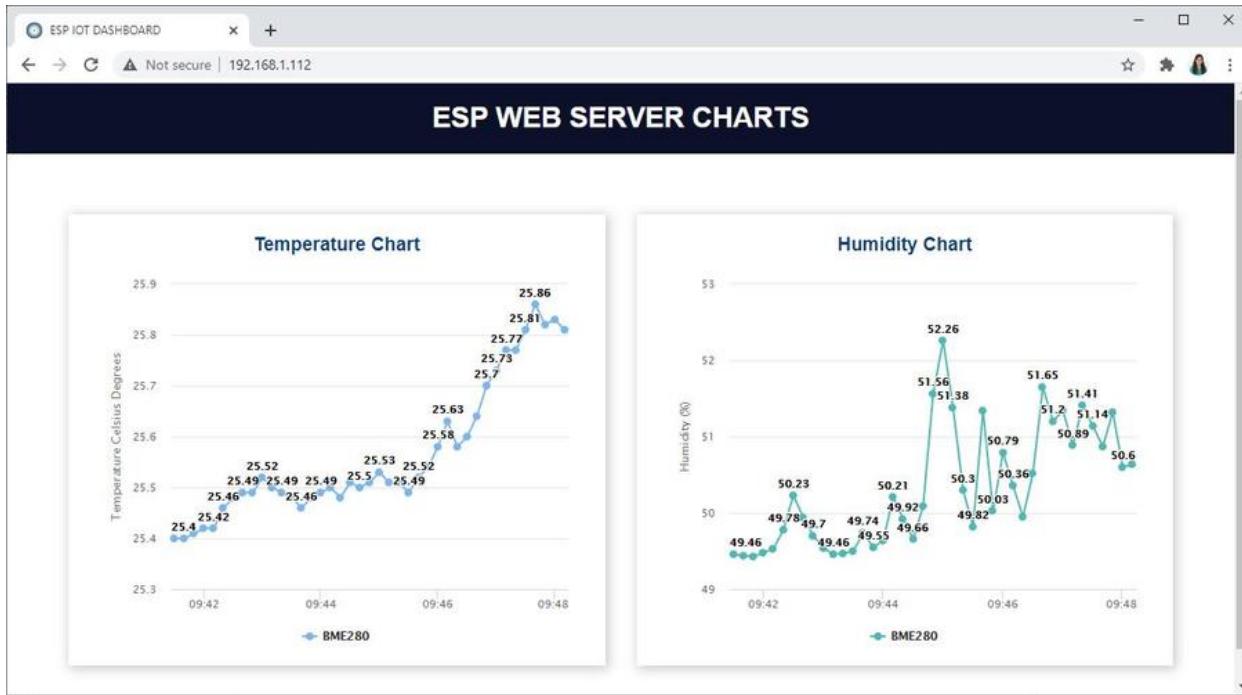


Learn how to plot sensor readings (temperature and humidity) on a web server using the ESP32 or ESP8266. The ESP will host a web page with two real-time charts with new readings added every 30 seconds.

To build the charts, we'll use the [Highcharts JavaScript library](#). We'll create two charts: temperature over time and humidity over time. The charts display a maximum of 40 data points, and a new reading is added every 30 seconds. You can change these values in your code.

Project Overview

Here's the web page you'll build for this project.



The chart on the left displays temperature readings over time. The chart on the right displays humidity readings over time. New readings are added every 30 seconds. The charts are built using the [Highcharts JavaScript library](#).

The web page for this project uses the same structure as in previous Units. Inside the cards, there's a card title and the chart.

How it Works?

1. When the page first loads, the client: calls the JavaScript functions to create the charts; makes an HTTP GET request to get the current sensor readings and plots the readings in the charts; initializes the Event Source protocol, so that it can receive new sensor readings as events.
2. When new readings are available, the server sends an event to the client.

3. The client receives an event with the readings and plots the readings in the corresponding charts.

Assembling the Circuit

For this example, you need to wire a BME280 sensor to your board.

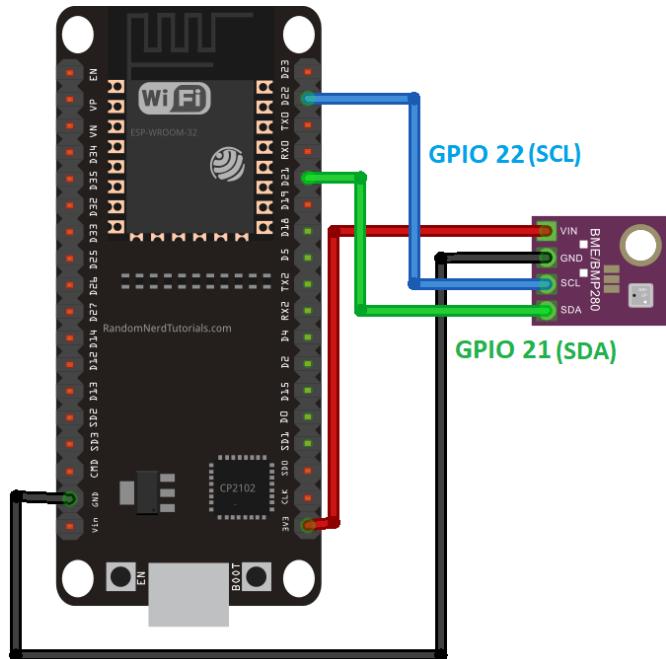
Parts Required

To complete this tutorial, you need the following parts:

- [BME280 sensor module](#)
- [ESP32 or ESP8266](#)
- [Breadboard](#)
- [Jumper wires](#)

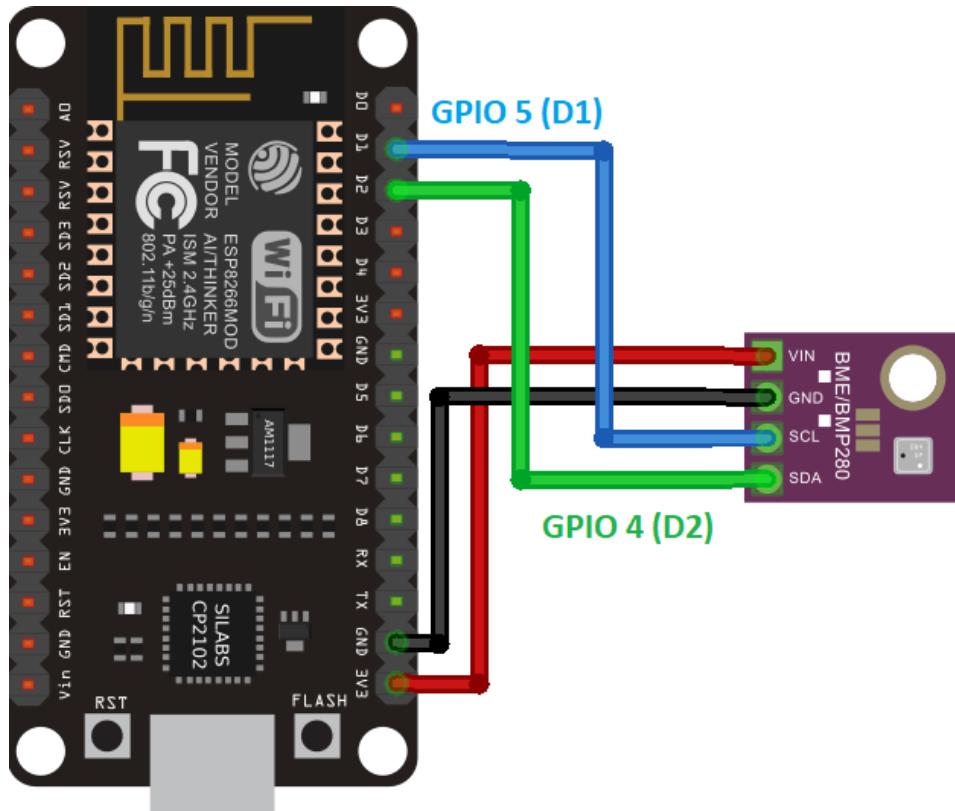
ESP32 - Schematic Diagram

Wire the sensor to the ESP32 default SDA (GPIO 21) and SCL (GPIO 22) pins, as shown in the following schematic diagram.



ESP8266 - Schematic Diagram

Follow the next schematic diagram if you're using an ESP8266. GPIO 5 and GPIO 4 are the ESP8266 I2C pins.



Building the Web Page

To create the web page for this project, place the following files inside the *data* folder within your project folder:*index.html*

- *style.css*
- *script.js*
- *favicon.png*

HTML File

Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fmnOCqbTlWIj8LyTjo7mOUSTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css">
    <script src="https://code.highcharts.com/highcharts.js"></script>
</head>
<body>
    <div class="topnav">
        <h1>ESP WEB SERVER CHARTS</h1>
    </div>
    <div class="content">
        <div class="card-grid">
            <div class="card">
                <p class="card-title">Temperature Chart</p>
                <div id="chart-temperature" class="chart-container"></div>
            </div>
            <div class="card">
                <p class="card-title">Humidity Chart</p>
                <div id="chart-humidity" class="chart-container"></div>
            </div>
        </div>
        <script src="script.js"></script>
    </body>
</html>
```

The structure of the HTML file is similar to previous projects. Note that we need to include the JavaScript Highcharts library in the head of the HTML file like this:

```
<script src="https://code.highcharts.com/highcharts.js"></script>
```

In the HTML body, there are two cards. Here's the card for the temperature:

```
<div class="card">
    <p class="card-title">Temperature Chart</p>
    <div id="chart-temperature" class="chart-container"></div>
</div>
```

Inside the card, there's a paragraph with the title like in previous projects.

```
<p class="card-title">Temperature Chart</p>
```

The chart should be placed inside a `<div>` tag with a specific `id`. For the temperature chart, we created a `<div>` with the `chart-temperature` id.

```
<div id="chart-temperature" class="chart-container"></div>
```

The chart will be placed in that `<div>` element using JavaScript.

The HTML to build the card for the humidity is similar. The id for the `<div>` tag is `chart-humidity`:

```
<div class="card">
  <p class="card-title">Humidity Chart</p>
  <div id="chart-humidity" class="chart-container"></div>
</div>
```

CSS File

Copy the following styles to your `style.css` file.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  display: inline-block;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
p {
  font-size: 1.4rem;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
}
.content {
  padding: 5%;
}
.card-grid {
  max-width: 1200px;
```

```
margin: 0 auto;
display: grid;
grid-gap: 2rem;
grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}
.card {
background-color: white;
box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
font-size: 1.2rem;
font-weight: bold;
color: #034078
}
.chart-container {
padding-right: 5%;
padding-left: 5%;
}
```

The styles are the same used in the previous projects with slight differences (explained below).

Format the `chart-container` with some padding at the left and right.

```
.chart-container {
padding-right: 5%;
padding-left: 5%;
}
```

Additionally, to have more space in the web browser window for the charts, we decreased the padding of the `content`:

```
.content {
padding: 5%;
}
```

And increased the `max-width` of the `card-grid`.

```
.card-grid {
max-width: 1200px;
margin: 0 auto;
display: grid;
grid-gap: 2rem;
grid-template-columns: repeat(auto-fit, minmax(200px, 1fr)));
}
```

If you prefer, you can keep exactly the same styles as previous projects.

JavaScript File

Copy the following code to your `script.js` file. This is responsible for:

- initializing the event source protocol;
- adding an event listener for the `new_readings` event;
- creating the charts;
- getting the latest sensor readings from the `new_readings` event and plot them in the charts;
- making an HTTP GET request for the current sensor readings when you access the web page for the first time;

```
// Get current sensor readings when the page loads
window.addEventListener('load', getReadings);

// Create Temperature Chart
var chartT = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-temperature'
    },
    series: [
        {
            name: 'BME280'
        }
    ],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Temperature Celsius Degrees'
        }
    },
    credits: {
```

```

        enabled: true
    }
});

// Create Humidity Chart
var chartH = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-humidity'
    },
    series: [{
        name: 'BME280'
    }],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        },
        series: {
            color: '#50b8b4'
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Humidity (%)'
        }
    },
    credits: {
        enabled: false
    }
});

//Plot temperature in the temperature chart
function plotTemperature(value) {
    var x = (new Date()).getTime()
    var y = Number(value);
    if(chartT.series[0].data.length > 40) {
        chartT.series[0].addPoint([x, y], true, true, true);
    } else {
        chartT.series[0].addPoint([x, y], true, false, true);
    }
}

//Plot humidity in the humidity chart
function plotHumidity(value) {
    var x = (new Date()).getTime()

```

```

var y = Number(value);
if(chartH.series[0].data.length > 40) {
    chartH.series[0].addPoint([x, y], true, true, true);
} else {
    chartH.series[0].addPoint([x, y], true, false, true);
}
}

// Function to get current readings on the web page when it loads
function getReadings() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            var temp = myObj.temperature;
            var hum = myObj.humidity;
            plotTemperature(temp);
            plotHumidity(hum);
        }
    };
    xhr.open("GET", "/readings", true);
    xhr.send();
}

if (!!window.EventSource) {
    var source = new EventSource('/events');

    source.addEventListener('open', function(e) {
        console.log("Events Connected");
    }, false);

    source.addEventListener('error', function(e) {
        if (e.target.readyState != EventSource.OPEN) {
            console.log("Events Disconnected");
        }
    }, false);

    source.addEventListener('message', function(e) {
        console.log("message", e.data);
    }, false);

    source.addEventListener('new_readings', function(e) {
        console.log("new_readings", e.data);
        var myObj = JSON.parse(e.data);
        console.log(myObj);
        plotTemperature(myObj.temperature);
        plotHumidity(myObj.humidity);
    }, false);
}

```

This JavaScript code is similar to the code used in previous projects but adds the necessary functions to build the charts and plot the readings in the charts. Let's take a look at the relevant parts of this project.

Time Zone

If after building the project, the charts are not showing the right time zone, add the following lines to the JavaScript file after the second line:

```
Highcharts.setOptions({
  time: {
    timezoneOffset: -60 //Add your time zone offset here in seconds
  }
});
```

Add the time zone offset in seconds.

Creating the Charts

The following lines create the chart for the temperature:

```
var chartT = new Highcharts.Chart({
  chart: {
    renderTo: 'chart-temperature'
  },
  series: [
    {
      name: 'BME280'
    }
  ],
  title: {
    text: undefined
  },
  plotOptions: {
    line: {
      animation: false,
      dataLabels: {
        enabled: true
      }
    }
  },
  xAxis: {
    type: 'datetimestamp',
    dateTimeLabelFormats: { second: '%H:%M:%S' }
  },
  y-axis: {
    title: {
      text: 'Temperature (°C)'
    }
  }
});
```

```
yAxis: {
  title: {
    text: 'Temperature Celsius Degrees'
  }
},
credits: {
  enabled: false
}
});
```

To create a new chart use the `new Highcharts.Chart()` method and pass as argument the chart properties.

In the next line, define where you want to put the chart. In our example, we want to place it in the `<div>` tag with the `chart-temperature` id.

```
chart: {
  renderTo: 'chart-temperature'
},
```

Define the series name.

```
series: [
  {
    name: 'BME280'
  }
],
```

You can add a title and subtitle to the chart. For example:

```
title: {
  text: 'My custom title'
},
subtitle: {
  text: 'My custom subtitle'
}
```

However, because we're already displaying the card title, we don't want to display any title or subtitle. The title is displayed by default, so we must set it to `undefined`.

```
title: {
  text: undefined
},
```

General options that apply to the series are defined in `plotOptions`.

```
plotOptions: {  
    line: {  
        animation: false,  
        dataLabels: {  
            enabled: true  
        }  
    }  
},
```

- `animation`: allows disabling or altering the characteristics of the initial animation of a series. Animation is enabled by default, so we set it to `false`. You can set it to `true` to see how it works.
- `dataLabels`: allows data labels to be displayed for each data point in a series on the chart.

There are many other properties you can set in `plotOption` to change how the series looks. We recommend taking a look at the documentation and test the properties to see how it works:

- <https://www.highcharts.com/docs/chart-concepts/series>

Define the x axis and the y axis properties:

```
xAxis: {  
    type: 'datetime',  
    dateTimeLabelFormats: { second: '%H:%M:%S' }  
},  
yAxis: {  
    title: {  
        text: 'Temperature Celsius Degrees'  
    }  
},
```

Finally, set the `credits` option to `false` to hide the credits of the Highcharts library.

```
credits: {  
    enabled: false  
}
```

Creating the chart for the humidity is similar.

```
// Create Humidity Chart
var chartH = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-humidity'
    },
    series: [{
        name: 'BME280'
    }],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        },
        series: {
            color: '#50b8b4'
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Humidity (%)'
        }
    },
    credits: {
        enabled: false
    }
});
```

The chart is placed in the HTML element with the `chart-humidity` id.

```
chart: {
    renderTo: 'chart-humidity'
},
```

We set a different color for the series:

```
series: {
    color: '#50b8b4'
}
```

And give a different name to the `yAxis`:

```
yAxis: {
  title: {
    text: 'Humidity (%)'
  }
},
```

Plot Temperature and Humidity

To plot temperature and humidity in the charts, we created two functions: `plotTemperature()` and `plotHumidity()` that accept as argument the value we want to plot.

```
function plotTemperature(value){
  var x = (new Date()).getTime()
  var y = Number(value);
  if(chartT.series[0].data.length > 40) {
    chartT.series[0].addPoint([x, y], true, true, true);
  } else {
    chartT.series[0].addPoint([x, y], true, false, true);
  }
}
```

The `x` value for the chart is the timestamp.

```
var x = (new Date()).getTime()
```

The `y` value is passed as an argument. We need to convert it to a number.

```
var y = Number(value);
```

Our charts only have one series (index is 0). So, we can access the series in the temperature chart using: `chartT.series[0]`.

First, we check the series data length:

- If the series has more than 40 points: append and shift a new point;
- Or if the series has less than 40 points: append a new point.

To add a new point, use the `addPoint()` method that accepts the following arguments:

- The value to be plotted. If it is a single number, a point with that y value is appended to the series. If it is an array, it will be interpreted as x and y values. In our case, we pass an array with the x and y values;
- Redraw option (boolean): set to true to redraw the chart after the point is added.
- Shift option (boolean): If true, a point is shifted off the start of the series as one is appended to the end. When the chart length is bigger than 40, we set the shift option to true.
- withEvent option (boolean): Used internally, to fire the series addPoint event.

So, to add a point to the chart, we use the following lines:

```
if(chartT.series[0].data.length > 40) {
  chartT.series[0].addPoint([x, y], true, true, true);
} else {
  chartT.series[0].addPoint([x, y], true, false, true);
}
```

There's something similar to plot the humidity. We use the chartH instead:

```
function plotHumidity(value){
  var x = (new Date()).getTime()
  var y = Number(value);
  if(chartH.series[0].data.length > 40) {
    chartH.series[0].addPoint([x, y], true, true, true);
  } else {
    chartH.series[0].addPoint([x, y], true, false, true);
  }
}
```

getReadings()

When the page loads in your browser, it calls the getReadings() function to get the current sensor readings. That function makes an HTTP request on the /readings URL. The server responds with a JSON string containing the latest sensor readings.

Plot the sensor readings by calling the `plotTemperature()` and `plotHumidity()` functions and pass the temperature (`myObj.temperature`) and humidity (`myObj.humidity`) as arguments.

```
function getReadings() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var myObj = JSON.parse(this.responseText);
      console.log(myObj);
      plotTemperature(myObj.temperature);
      plotHumidity(myObj.humidity);
    }
  };
  xhr.open("GET", "/readings", true);
  xhr.send();
}
```

Handle events

Plot the readings on the charts when the client receives them in the `new_readings` event.

```
source.addEventListener('new_readings', function(e) {
  console.log("new_readings", e.data);
  var myObj = JSON.parse(e.data);
  console.log(myObj);
  plotTemperature(myObj.temperature);
  plotHumidity(myObj.humidity);
}, false);
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The *platformio.ini* file for the ESP32 should be like this (same as previous Unit).

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit BME280 Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this (same as previous Unit).

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit BME280 Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
```

```

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

// Create a sensor object
Adafruit_BME280 bme;           // BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO
22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
}

```

```

    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    initBME();
    initWiFi();
    initSPIFFS();

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

    server.serveStatic("/", SPIFFS, "/");

    // Request for the latest sensor readings
    server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
        String json = getSensorReadings();
        request->send(200, "application/json", json);
        json = String();
    });

    events.onConnect([] (AsyncEventSourceClient *client){
        if(client->lastId()){
            Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
        }
        // send event with message "hello!", id current millis
        // and set reconnect delay to 1 second
        client->send("hello!", NULL, millis(), 10000);
    });
    server.addHandler(&events);

    // Start server
    server.begin();
}

void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 10 seconds
        events.send("ping", NULL, millis());
        events.send(getSensorReadings().c_str(), "new_readings", millis());
        lastTime = millis();
    }
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

The code is exactly the same as that used in the previous project. If you want to learn how it works, go back to Project 3.1

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 30000;

// Create a sensor object
Adafruit_BME280 bme;           // BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO
22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}
```

```

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    initBME();
    initWiFi();
    initFS();
    // Web Server Root URL
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html");
    });
    server.serveStatic("/", LittleFS, "/");
}

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});
events.onConnect([] (AsyncEventSourceClient *client){
    if (client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
});

```

```

    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});

server.addHandler(&events);

// Start server
server.begin();
}

void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 10 seconds
        events.send("ping",NULL,millis());
        events.send(getSensorReadings().c_str(),"new_readings" ,millis());
        lastTime = millis();
    }
}

```

Modify the code to include your network credentials, and it will work straight away.

```

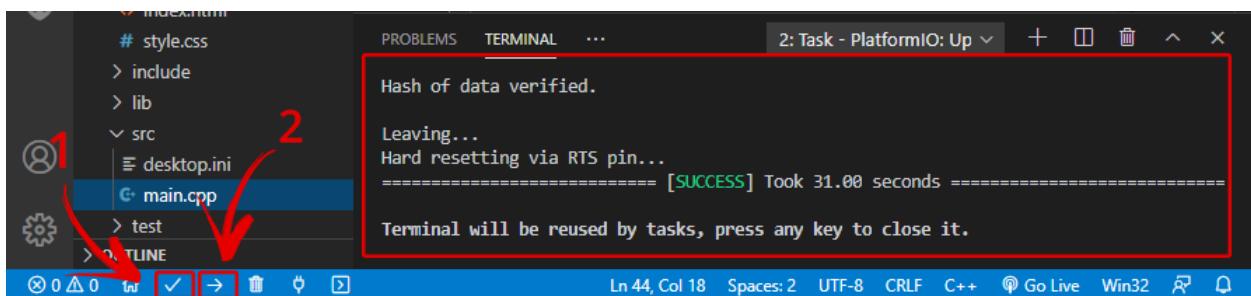
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

The code is the same as that used in the previous projects. If you want to learn how it works, go back to Project 3.1.

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

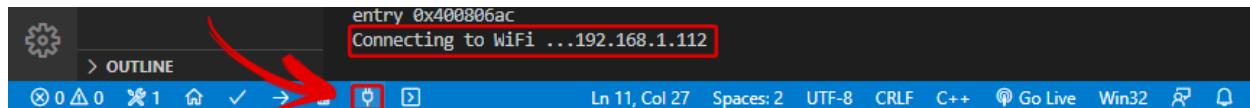
Uploading Filesystem Image

Finally, upload the files (*index.html*, *style.css*, *script.js* and *favicon.png*) to the filesystem:

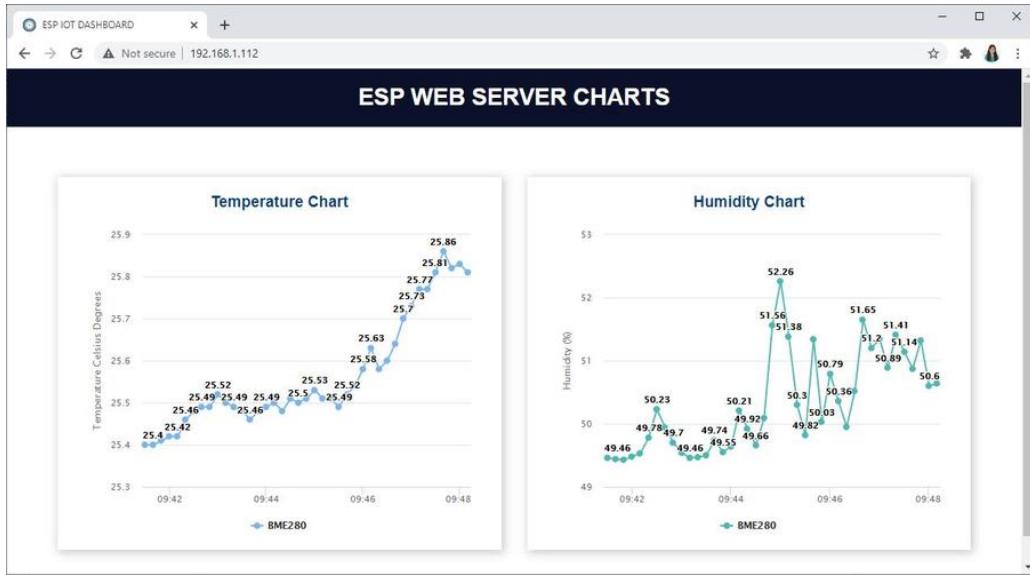
1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.
5. Finally, click **Upload Filesystem Image**.

Demonstration

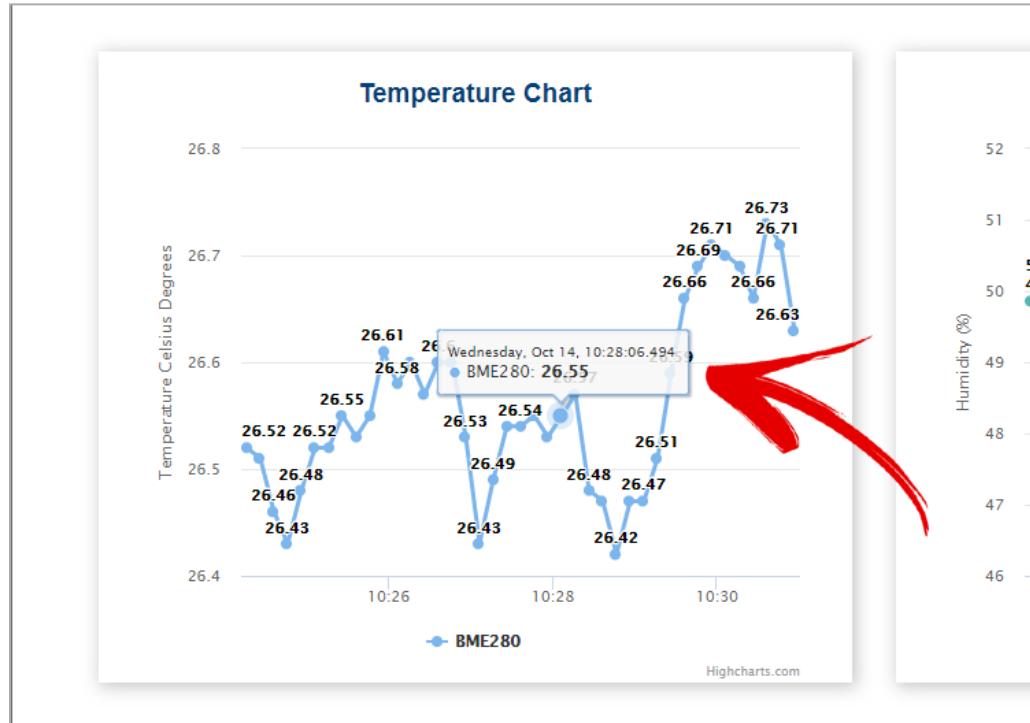
After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.



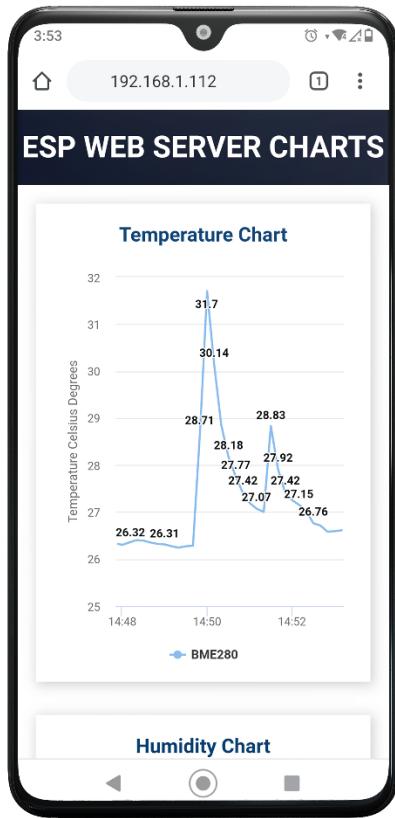
Open a browser on your local network and insert the ESP IP address. You should get access to the web page to monitor the sensor readings. The page displays two charts. A new data point is added every 30 seconds to a total of 40 points. New data keeps being displayed on the charts as long as you have your web browser tab opened.



You can select each point to see the exact timestamp.



Here's how the web page looks like on your smartphone.



Download Project Folder

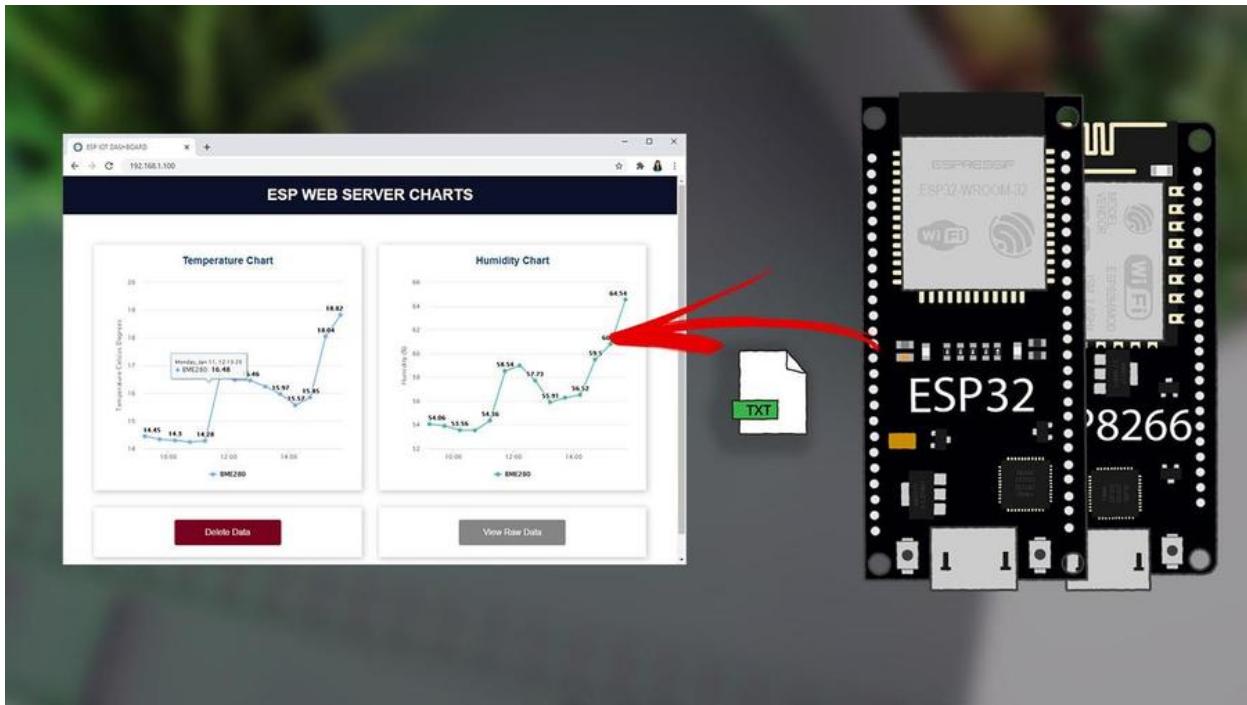
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to create charts to display data on your web page. You can modify this project to create as many charts as you want and use any other sensors. You can also play with the chart properties to change its appearance. We recommend taking a look at the [HighCharts documentation](#).

3.4 - Web Server: Display Sensor Readings from File (Charts)



In the previous Unit, you've learned how to display sensor readings on charts using the Highcharts JavaScript Library. In that example, you can only access the readings while you have a client connection (your web browser open). If you close the window and open it again, it will have lost all previous readings. In this Unit, we'll modify the previous project so that your charts always load the last 40 data points.

To always display the last 40 readings whenever you open a client connection (even if you reset or restart your ESP board), we need to save the readings in the ESP's flash memory. We'll create a file in the filesystem (SPIFFS for the ESP32 and LittleFS for the ESP8266) to save the readings.

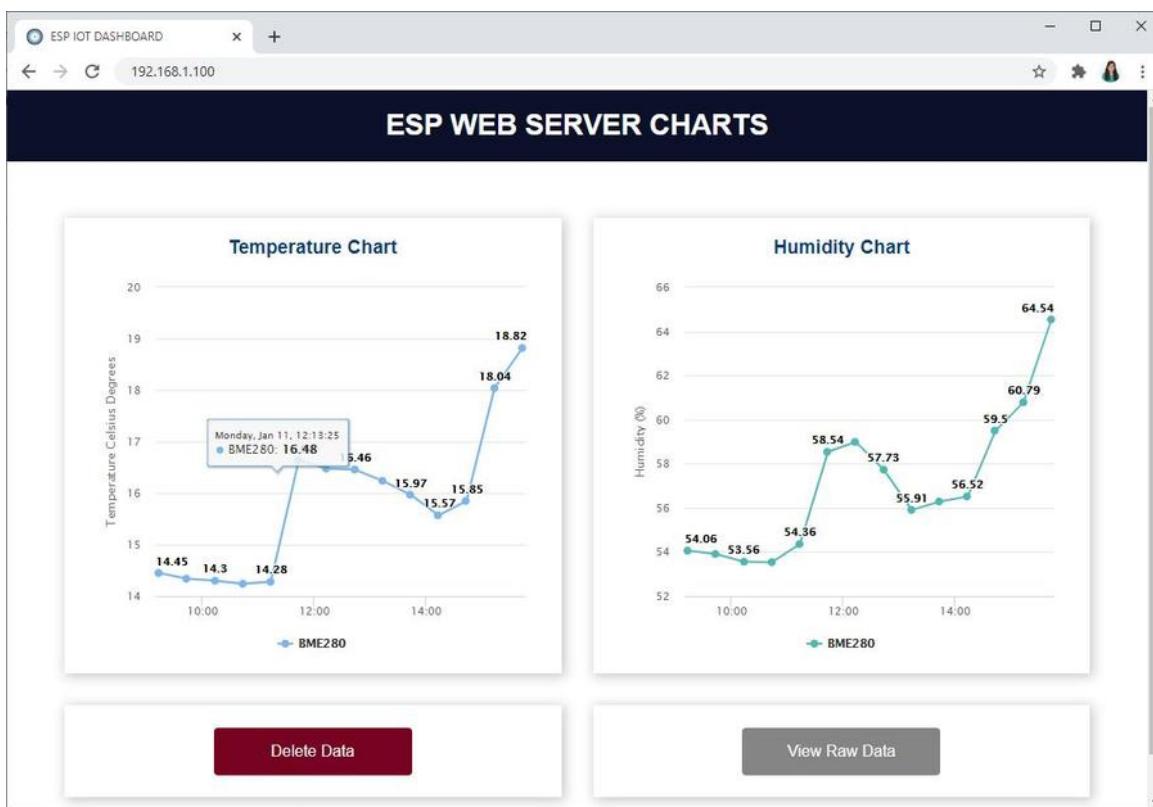
The ESP filesystem has a limited size, so we'll check the file size every time new readings are saved. When the file size is bigger than a predetermined size, we'll delete the contents of that file. To save significant amounts of data for prolonged periods,

we recommend using a microSD card, or logging your readings into a database (not covered in this eBook).

To learn about managing files using a microSD card, we recommend reading the following tutorial: [ESP32: Guide for MicroSD Card Module using Arduino IDE](#).

Project Overview

Here's the web page you'll build for this project.



The chart on the left displays temperature readings over time. The chart on the right displays humidity readings over time. New readings are added every 30 minutes. The charts are built using the [Highcharts JavaScript library](#).

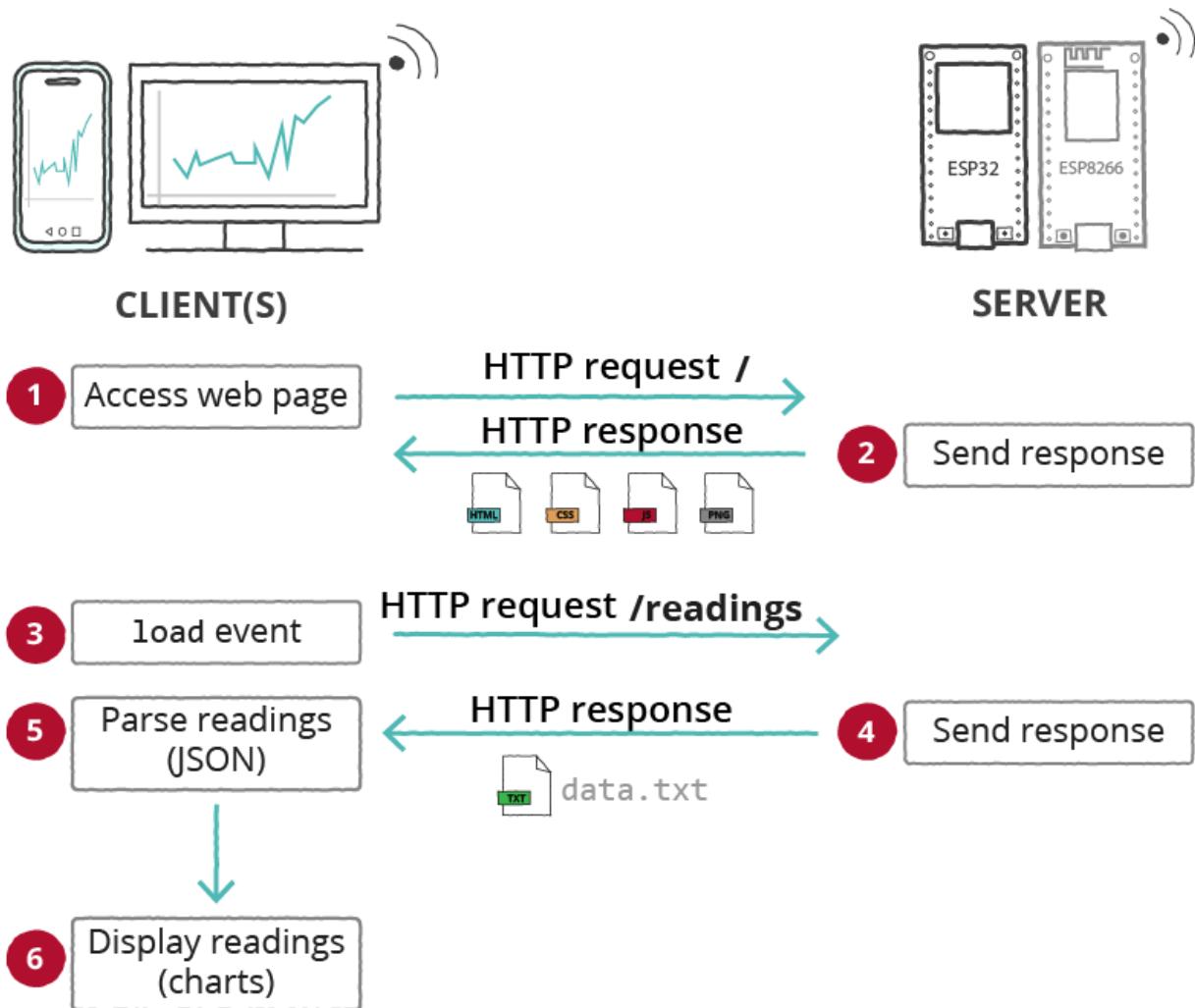
When you open a client connection, it will request the last available readings (to a maximum of 40 readings) and display them on the chart.

There are two additional buttons on the web page:

- **Delete Data:** deletes the content of the file that saves the readings. All data points on the chart are also deleted;
- **View Raw Data:** opens a new web page with all the raw data saved in the file.

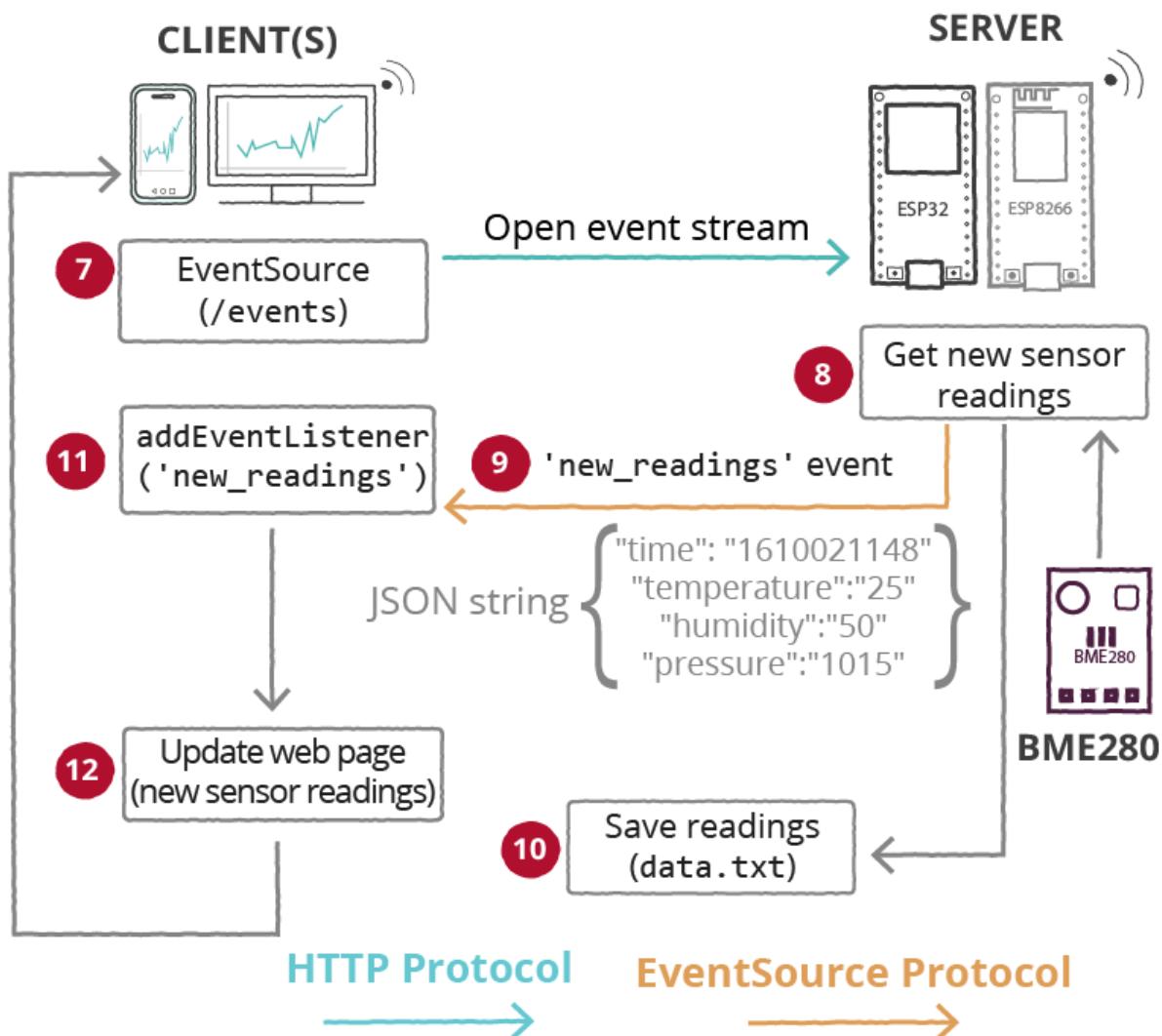
The web page for this project uses the same structure and styles as previous Units.

How it Works?



1. When you open a client connection, it will make a request to the server on the root ("/") URL.
2. The server responds with the HTML, CSS, JavaScript, and PNG (favicon) files.

3. When the page first loads (`load` event), the client calls a JavaScript function that makes an HTTP GET request on the `/readings` URL to get the last sensor readings saved in the file.
4. The server responds with the content of the `data.txt` file that contains the readings.
5. The client parses the content of the `data.txt` file into a JSON object using the `parse()` method.
6. Finally, it displays the last sensor readings on charts.

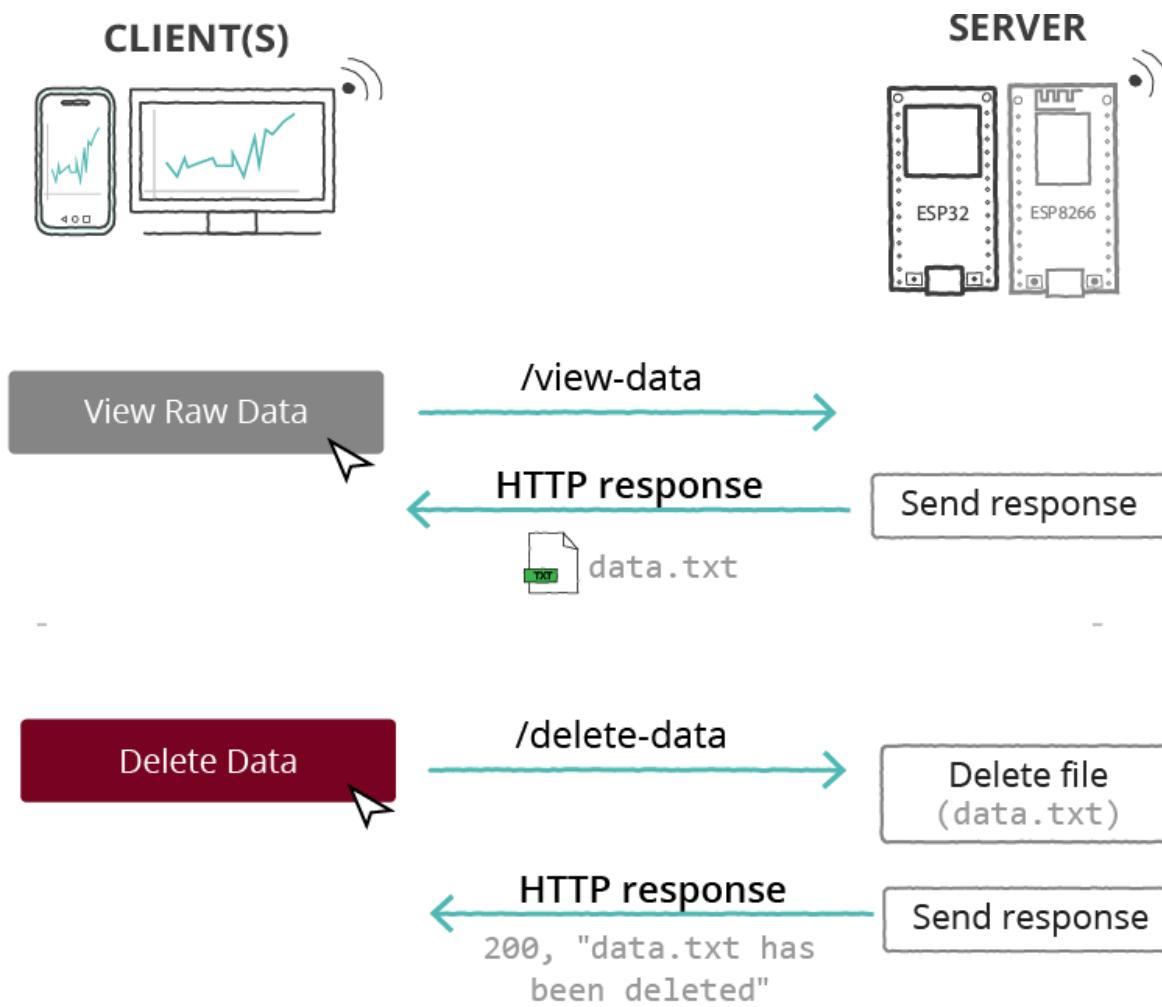


7. After that, it initializes the Event Source protocol to receive new sensor readings as events.

8. The server gets new sensor readings from the BME280. When new readings are available...
9. ... the server sends an event (`new_readings`) to the client.
10. The server saves the readings on a file (`data.txt`).
11. The client receives an event with the readings...
12. ... and plots the readings to the corresponding charts.

On the web page, there are also the **View Raw Data** and the **Delete Data** buttons.

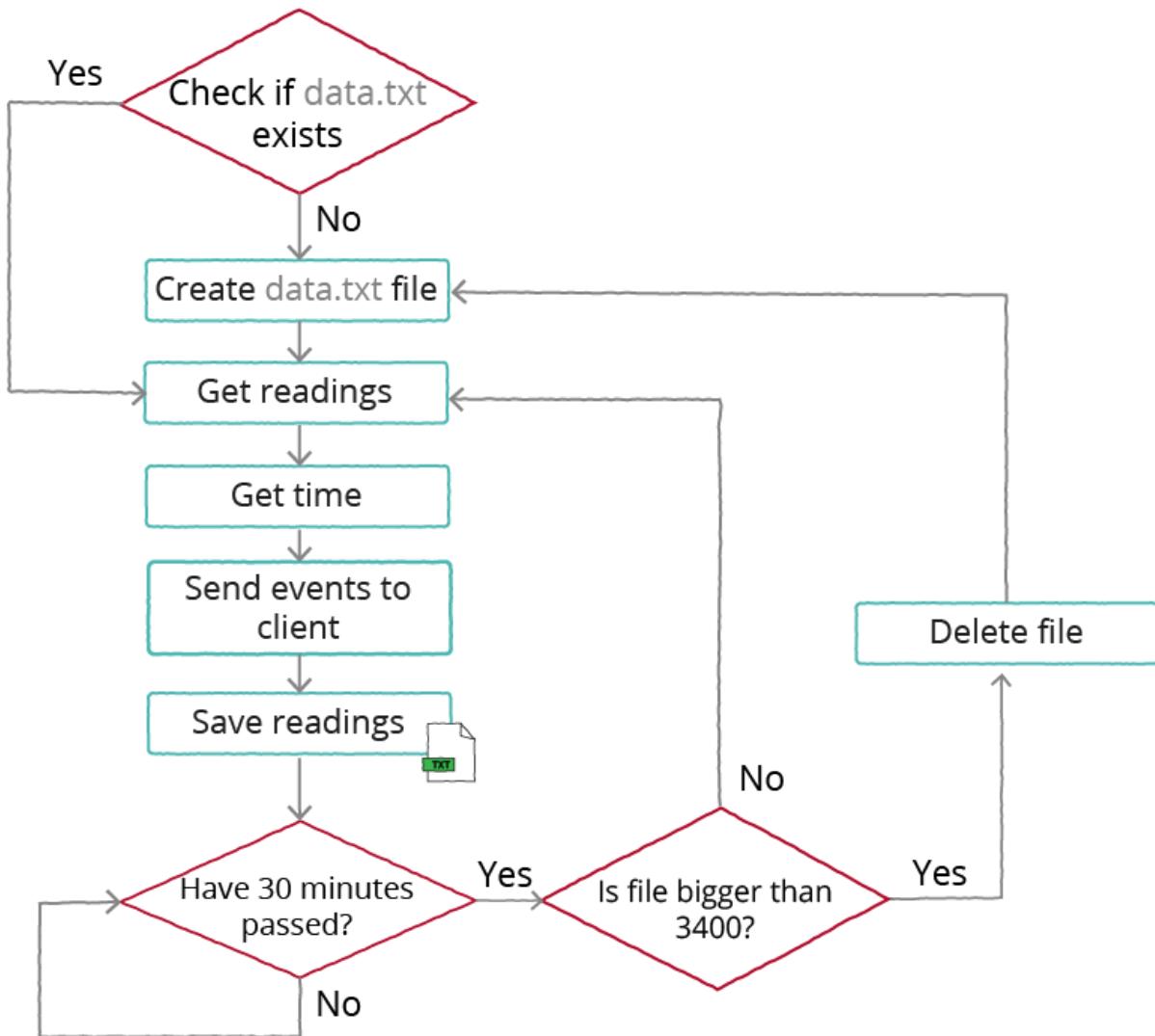
The following diagram shows how they work:



- When you click on the **View Raw Data** button, it makes a request on the `/view-data` URL. The server receives that request and sends a response with the `data.txt` file.

- When you click on the **Delete Data** button, it makes a request on the `/delete-data` URL. The server receives that request, deletes the file, and responds with a text message.

Here's what happens on the server side:



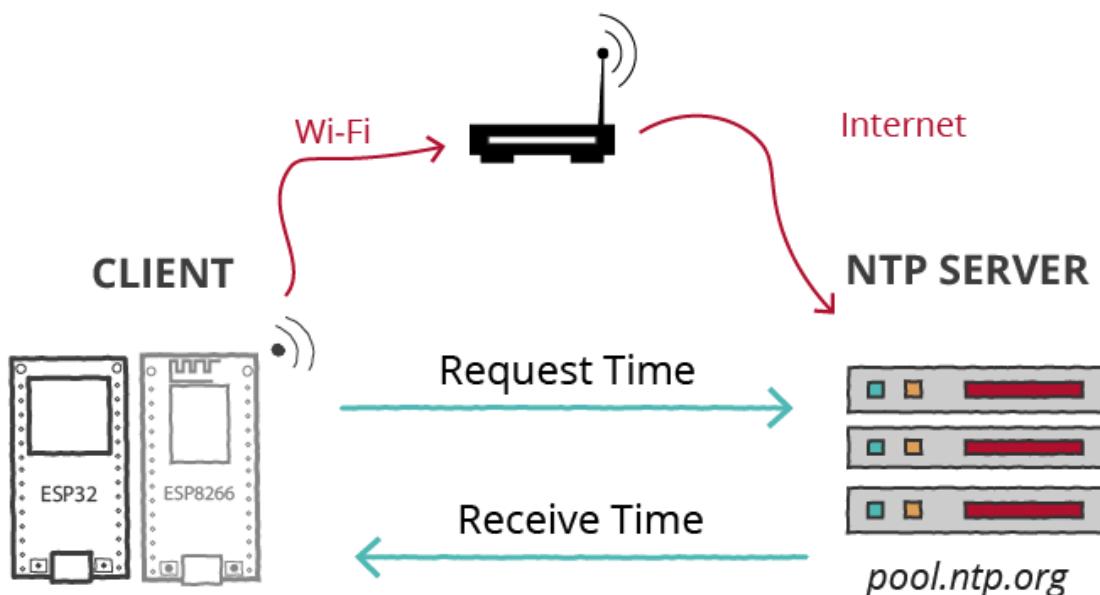
- When the server first runs, it checks if the `data.txt` file has already been created. If not, it creates the file. This file is used to log sensor readings;
- It gets new sensor readings from the BME280 sensor;
- It connects to an NTP server to timestamp the readings;
- It sends the new readings to the client as an event;
- Saves the new readings in the `data.txt` file;

6. In the `loop()`, the ESP gets new sensor readings (as well as time) from the BME280 sensor every 30 minutes;
7. If the file is bigger than a predetermined size set on the code (in this case 3400 bytes—saves approximately 40 data points), the file is deleted;
8. Otherwise, new readings are appended to the `data.txt` file.

NTP Server and Client

NTP stands for Network Time Protocol, and it is a networking protocol for clock synchronization between computer systems. In other words, it is used to synchronize computer clock times in a network.

There are NTP servers like `pool.ntp.org` that anyone can use to request time as a client. In this case, the ESP32/ESP8266 is an NTP Client that requests time from an NTP Server (`pool.ntp.org`).



- To get date and time with the ESP32, you don't need to install any libraries. You simply need to include the `time.h` library in your code.
- With the ESP8266, we'll use the `NTPClient` library, as you'll see later on.

Assembling the Circuit

For this example, you need to wire a BME280 sensor to your board.

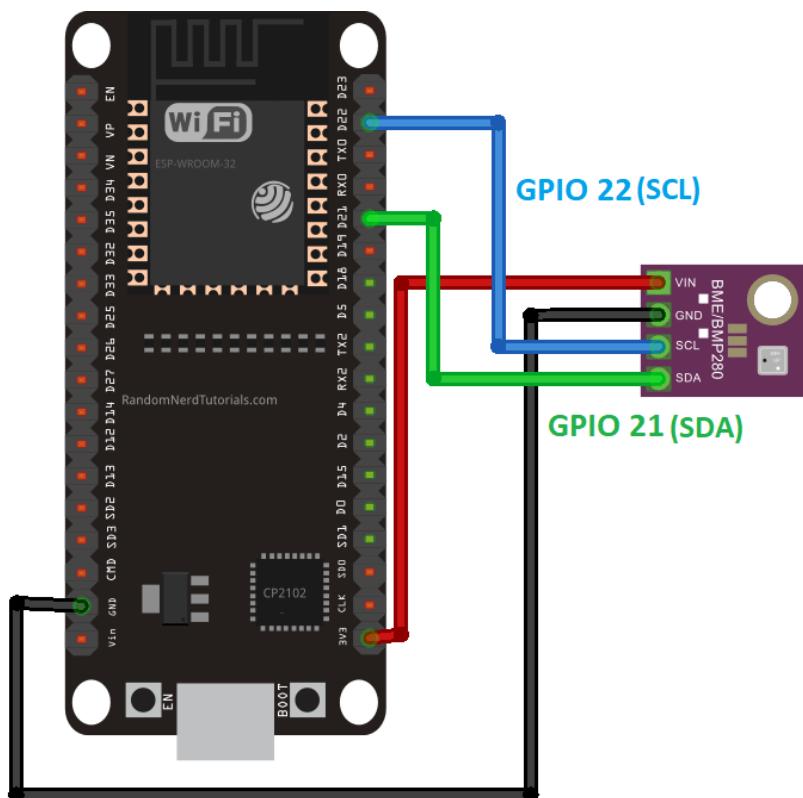
Parts Required

To complete this tutorial, you need the following parts:

- [BME280 sensor module](#)
- [ESP32 or ESP8266](#)
- [Breadboard](#)
- [Jumper wires](#)

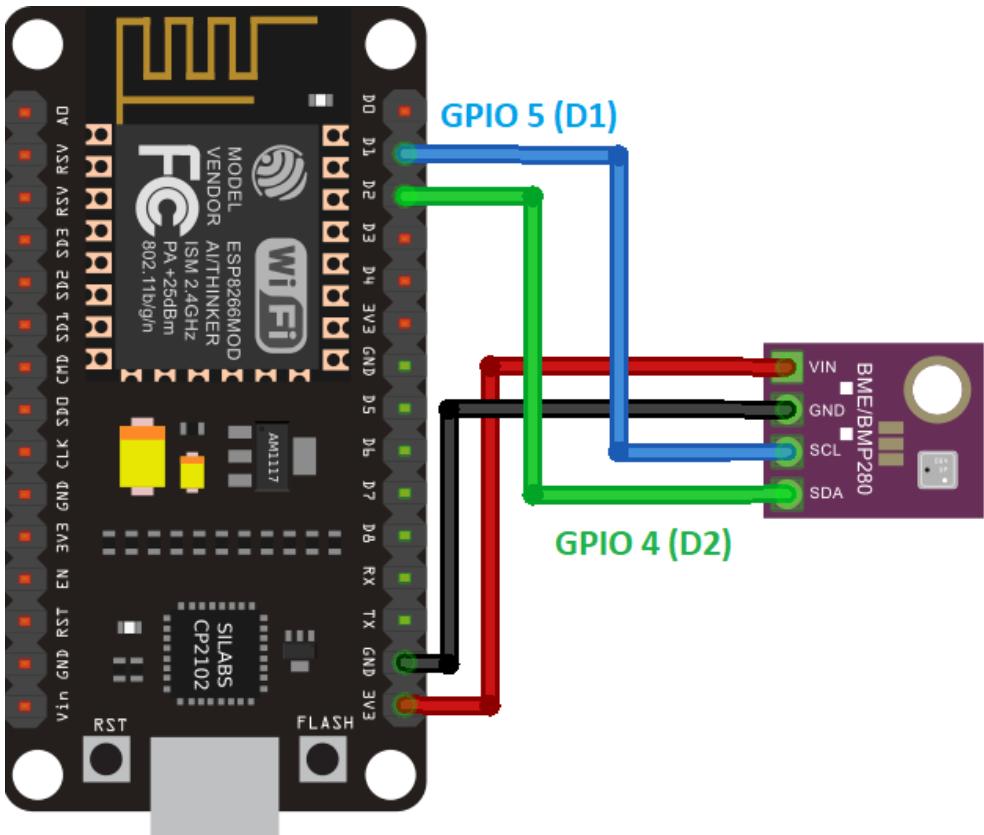
ESP32 - Schematic Diagram

Wire the sensor to the ESP32 default SDA (GPIO 21) and SCL (GPIO 22) pins, as shown in the following schematic diagram.



ESP8266 - Schematic Diagram

Follow the next schematic diagram if you're using an ESP8266. GPIO 5 and GPIO 4 are the ESP8266 I2C pins.



Building the Web Page

To build the web page for this project, place the following files inside the *data* folder within your project folder:*index.html*

- *style.css*
- *script.js*
- *favicon.png*

HTML File

Here's the text you should copy to your *index.html* file.

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7mOUStjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css">
    <script src="https://code.highcharts.com/highcharts.js"></script>
</head>
<body>
    <div class="topnav">
        <h1>ESP WEB SERVER CHARTS</h1>
    </div>
    <div class="content">
        <div class="card-grid">
            <div class="card">
                <p class="card-title">Temperature Chart</p>
                <div id="chart-temperature" class="chart-container"></div>
            </div>
            <div class="card">
                <p class="card-title">Humidity Chart</p>
                <div id="chart-humidity" class="chart-container"></div>
            </div>
            <div class="card">
                <p>
                    <a href="delete-data"><button class="button-delete">Delete Data</button></a>
                </p>
            </div>
            <div class="card">
                <p>
                    <a href="view-data"><button class="button-data">View Raw Data</button></a>
                </p>
            </div>
        </div>
    </div>
    <script src="script.js"></script>
</html>
```

The HTML file is very similar to the previous project but adds the **Delete Data** and **View Raw Data** buttons as shown in the following tags:

```
<div class="card">
  <p>
    <a href="delete-data"><button class="button-delete">Delete Data</button></a>
  </p>
</div>
<div class="card">
  <p>
    <a href="view-data"><button class="button-data">View Raw Data</button></a>
  </p>
</div>
```

The **Delete Data** button redirects to the `/delete-data` URL when clicked (`href="delete-data"`) and its class name is `button-delete`. The **View Raw Data** button redirects to the `/view-data` URL when clicked (`href="view-data"`) and its class name is `button-data`.

CSS File

Copy the following styles to your `style.css` file.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  display: inline-block;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
p {
  font-size: 1.4rem;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
}
.content {
  padding: 5%;
}
.card-grid {
  max-width: 1200px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(500px, 1fr));
```

```
}

.card {
    background-color: white;
    box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}

.card-title {
    font-size: 1.2rem;
    font-weight: bold;
    color: #034078
}

.chart-container{
    padding-right: 5%;
    padding-left: 5%;
}

button {
    border: none;
    color: #FEFCFB;
    padding: 15px 32px;
    text-align: center;
    display: inline-block;
    font-size: 16px;
    width: 200px;
    border-radius: 4px;
    transition-duration: 0.4s;
}

.button-delete {
    background-color: #780320;
}

.button-data {
    background-color: #858585;
}
```

The styles are the same as those used in the previous project, but we add some styles for the new buttons.

```
button {
    border: none;
    color: #FEFCFB;
    padding: 15px 32px;
    text-align: center;
    display: inline-block;
    font-size: 16px;
    width: 200px;
    border-radius: 4px;
    transition-duration: 0.4s; }

.button-delete {
    background-color: #780320; }

.button-data {
    background-color: #858585; }
```

JavaScript File

Copy the following code to your *script.js* file. This is responsible for:

- initializing the event source protocol;
- adding an event listener for the `new_readings` event;
- creating the charts;
- getting the latest sensor readings from the `new_readings` event and plotting them in the charts;
- making an HTTP GET request to get the last sensor readings saved in the `data.txt` file when you access the web page for the first time.

```
// Get current sensor readings when the page loads
window.addEventListener('load', getReadings);

// Create Temperature Chart
var chartT = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-temperature'
    },
    series: [
        {
            name: 'BME280'
        }
    ],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Temperature Celsius Degrees'
        }
    },
    credits: {
        enabled: false
    }
});
```

```

// Create Humidity Chart
var chartH = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-humidity'
    },
    series: [
        {
            name: 'BME280'
        }],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        },
        series: {
            color: '#50b8b4'
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Humidity (%)'
        }
    },
    credits: {
        enabled: false
    }
});

// Plot temperature in the temperature chart
function plotTemperature(timeValue, value){
    console.log(timeValue);
    var x = new Date(timeValue*1000).getTime();
    console.log(x);
    var y = Number(value);
    if(chartT.series[0].data.length > 40) {
        chartT.series[0].addPoint([x, y], true, true, true);
    } else {
        chartT.series[0].addPoint([x, y], true, false, true);
    }
}

// Plot humidity in the humidity chart
function plotHumidity(timeValue, value){
    console.log(timeValue);
    var x = new Date(timeValue*1000).getTime();

```

```

        console.log(x);
        var y = Number(value);
        if(chartH.series[0].data.length > 40) {
            chartH.series[0].addPoint([x, y], true, true, true);
        } else {
            chartH.series[0].addPoint([x, y], true, false, true);
        }
    }

// Function to get current readings on the webpage when it loads for the first time
function getReadings(){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            //console.log("[ "+this.responseText.slice(0, -1)+" ]");
            var myObj = JSON.parse("[ "+this.responseText.slice(0, -1)+" ]");
            //console.log(myObj);
            var len = myObj.length;
            if(len > 40) {
                for(var i = len-40; i<len; i++){
                    plotTemperature(myObj[i].time, myObj[i].temperature);
                    plotHumidity(myObj[i].time, myObj[i].humidity);
                }
            } else {
                for(var i = 0; i<len; i++){
                    plotTemperature(myObj[i].time, myObj[i].temperature);
                    plotHumidity(myObj[i].time, myObj[i].humidity);
                }
            }
        }
    };
    xhr.open("GET", "/readings", true);
    xhr.send();
}

if (!!window.EventSource) {
    var source = new EventSource('/events');

    source.addEventListener('open', function(e) {
        console.log("Events Connected");
    }, false);

    source.addEventListener('error', function(e) {
        if (e.target.readyState != EventSource.OPEN) {
            console.log("Events Disconnected");
        }
    }, false);

    source.addEventListener('message', function(e) {
        console.log("message", e.data);
    }, false);
    source.addEventListener('new_readings', function(e) {
}

```

```
    console.log("new_readings", e.data);
    var myObj = JSON.parse(e.data);
    console.log(myObj);
    plotTemperature(myObj.time, myObj.temperature);
    plotHumidity(myObj.time, myObj.humidity);
}, false);
}
```

Let's take a quick look at the code to see how it works.

Creating the Charts

The following lines create the Temperature chart.

```
// Create Temperature Chart
var chartT = new Highcharts.Chart({
  chart: {
    renderTo: 'chart-temperature'
  },
  series: [
    {
      name: 'BME280'
    }
  ],
  title: {
    text: undefined
  },
  plotOptions: {
    line: {
      animation: false,
      dataLabels: {
        enabled: true
      }
    }
  },
  xAxis: {
    type: 'datetime',
    dateTimeLabelFormats: { second: '%H:%M:%S' }
  },
  yAxis: {
    title: {
      text: 'Temperature Celsius Degrees'
    }
  },
  credits: {
    enabled: false
  }
});
```

These are exactly the same lines of code used in the previous project. So, if you want to learn how these lines work, just go back to that project.

We use a similar approach to build the Humidity chart.

```
// Create Humidity Chart
var chartH = new Highcharts.Chart({
    chart: {
        renderTo: 'chart-humidity'
    },
    series: [{
        name: 'BME280'
    }],
    title: {
        text: undefined
    },
    plotOptions: {
        line: {
            animation: false,
            dataLabels: {
                enabled: true
            }
        },
        series: {
            color: '#50b8b4'
        }
    },
    xAxis: {
        type: 'datetime',
        dateTimeLabelFormats: { second: '%H:%M:%S' }
    },
    yAxis: {
        title: {
            text: 'Humidity (%)'
        }
    },
    credits: {
        enabled: false
    }
});
```

Plot Temperature and Humidity

The remaining JavaScript code is only slightly different from that in the previous project. This time, the JSON variable sent by the server contains the timestamp in epoch time (in the previous project we got the time using a JavaScript function).

```

// Plot temperature in the temperature chart
function plotTemperature(timeValue, value){
    console.log(timeValue);
    var x = new Date(timeValue*1000).getTime();
    console.log(x);
    var y = Number(value);
    if(chartT.series[0].data.length > 40) {
        chartT.series[0].addPoint([x, y], true, true, true);
    } else {
        chartT.series[0].addPoint([x, y], true, false, true);
    }
}

// Plot humidity in the humidity chart
function plotHumidity(timeValue, value){
    console.log(timeValue);
    var x = new Date(timeValue*1000).getTime();
    console.log(x);
    var y = Number(value);
    if(chartH.series[0].data.length > 40) {
        chartH.series[0].addPoint([x, y], true, true, true);
    } else {
        chartH.series[0].addPoint([x, y], true, false, true);
    }
}

```

The `plotTemperature()` and `plotHumidity()` functions accept as argument, the time (epoch time) and the temperature or humidity value.

```
function plotTemperature(timeValue, value){
```

Then, we need to convert the epoch time to human readable time as follows (the time is saved in the `x` variable):

```
var x = new Date(timeValue*1000).getTime();
```

Then, use the same lines of code used previously to plot the readings on the charts (`x` is the timestamp and `y` is the reading):

```

var y = Number(value);
if(chartT.series[0].data.length > 40) {
    chartT.series[0].addPoint([x, y], true, true, true);
} else {
    chartT.series[0].addPoint([x, y], true, false, true);
}

```

getReadings()

When the page loads in your browser, it calls the `getReadings()` function to get the last 40 data points saved in the `data.txt` file.

```
// Function to get current readings on the webpage when it loads for the first time
function getReadings(){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            //console.log("[ "+this.responseText.slice(0, -1)+" ]");
            var myObj = JSON.parse("[ "+this.responseText.slice(0, -1)+" ]");
            //console.log(myObj);
            var len = myObj.length;
            if(len > 40) {
                for(var i = len-40; i<len; i++){
                    plotTemperature(myObj[i].time, myObj[i].temperature);
                    plotHumidity(myObj[i].time, myObj[i].humidity);
                }
            } else {
                for(var i = 0; i<len; i++){
                    plotTemperature(myObj[i].time, myObj[i].temperature);
                    plotHumidity(myObj[i].time, myObj[i].humidity);
                }
            }
        };
        xhr.open("GET", "/readings", true);
        xhr.send();
    }
}
```

This function makes an HTTP request on the `/readings` URL. The server responds with the content saved in the `data.txt` file. That file contains multiple JSON variables separated by commas. For example, if the file contained the last three readings, it would look as follows:

```
{
  "time" : "1610021148",
  "temperature" : "25",
  "humidity" : "50",
  "pressure" : "1015"
},
{
  "time" : "1610021557",
  "temperature" : "26",
  "humidity" : "51",
  "pressure" : "1014"
```

```
},
{
  "time" : "1610021775",
  "temperature" : "23",
  "humidity" : "50",
  "pressure" : "1013"
},
```

Our goal is to convert the content of that file into a JSON object. There are two things we need to do to convert it into valid JSON format:

- Delete the last comma (,) – for that we'll use the `slice()` JavaScript method;
- Add square brackets [] to the beginning and to the end of the file to convert it into a JSON array;
- Use the JavaScript `parse()` method to convert a JSON string into a JSON object.

```
var myObj = JSON.parse("[ "+this.responseText.slice(0, -1)+" ]");
```

The `slice(start, end)` method selects the elements starting at the given `start` argument and ends at the given `end` argument:

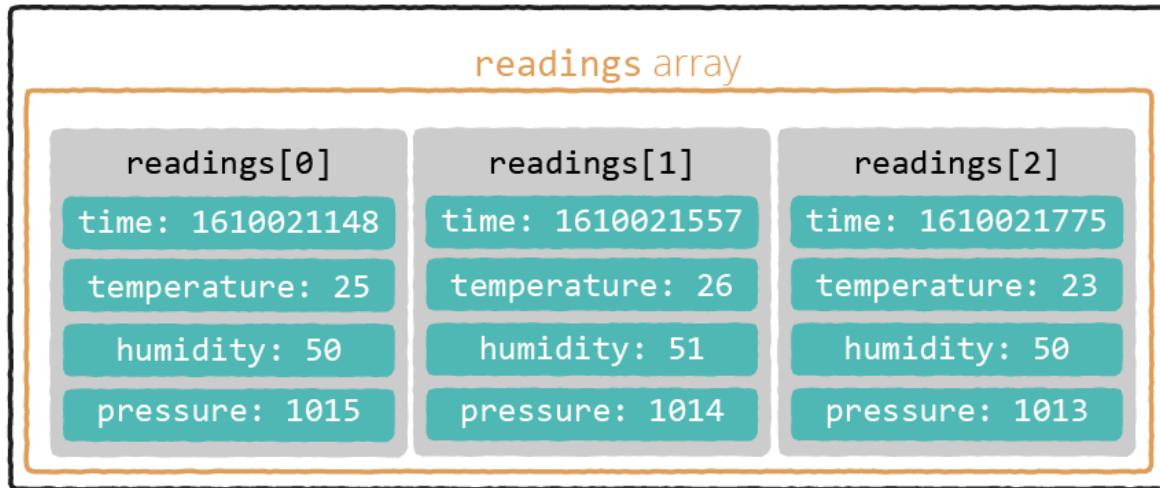
- `start`: an integer that specifies where to start the selection (the first element has an index of 0);
- `end`: an integer that specifies where to end the selection (use negative numbers to select from the end of an array).

After this previous line, we have a valid JSON object called `myObj`. Here are some examples on how to access the readings:

- Temperature readings of `readings[0]`: `myObj[0].temperature`
- Humidity readings of `readings[1]`: `myObj[1].humidity`

To better understand how it works, take a look at the following picture that shows a representation of the `myObj` variable.

myObj JavaScript object



Then, we need to access all the information inside the object and place it in the charts. We can do it using a `for` loop.

```
var len = myObj.length;
if(len > 40) {
  for(var i = len-40; i<len; i++){
    plotTemperature(myObj[i].time, myObj[i].temperature);
    plotHumidity(myObj[i].time, myObj[i].humidity);
  }
}
else {
  for(var i = 0; i<len; i++){
    plotTemperature(myObj[i].time, myObj[i].temperature);
    plotHumidity(myObj[i].time, myObj[i].humidity);
  }
}
```

Handle Events

Plot the readings on the charts when the client receives the readings on the `new_readings` event.

```
source.addEventListener('new_readings', function(e) {
  console.log("new_readings", e.data);
  var myObj = JSON.parse(e.data);
  console.log(myObj);
  plotTemperature(myObj.time, myObj.temperature);
  plotHumidity(myObj.time, myObj.humidity);
}, false);
```

This time, the server sends a JSON variable in the following format:

```
{  
  "time" : "1610021148",  
  "temperature" : "25",  
  "humidity" : "50",  
  "pressure" : "1015"  
}
```

So, when plotting the temperature and humidity, we also need to pass the time as an argument. You access the time with `myObj.time`:

```
plotTemperature(myObj.time, myObj.temperature);  
plotHumidity(myObj.time, myObj.humidity);
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` file for the ESP32 should be like this (same as previous Unit).

```
[env:esp32doit-devkit-v1]  
platform = espressif32  
board = esp32doit-devkit-v1  
framework = arduino  
monitor_speed = 115200  
lib_deps = ESP Async WebServer  
  arduino-libraries/Arduino_JSON @ 0.1.0  
  adafruit/Adafruit BME280 Library @ ^2.1.0  
  adafruit/Adafruit Unified Sensor @ ^1.1.4
```

platformio.ini file (ESP8266)

The `platformio.ini` file for the ESP8266 should be like this. We need to include the `NTPClient` library.

```
[env:esp12e]  
platform = espressif8266  
board = esp12e  
framework = arduino  
monitor_speed = 115200
```

```
lib_deps      = ESP Async WebServer
  arduino-libraries/Arduino_JSON @ 0.1.0
  adafruit/Adafruit_BME280_Library @ ^2.1.0
  adafruit/Adafruit_Unified_Sensor @ ^1.1.4
  arduino-libraries/NTPClient @ ^3.1.0
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include "time.h"
#include <WiFiUdp.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// NTP server to request epoch time
const char* ntpServer = "pool.ntp.org";

// Json Variable to Hold Sensor Readings
JSONVar readings;

// File name where readings will be saved
const char* dataPath = "/data.txt";

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 1800000;

// Create a sensor object
```

```

Adafruit_BME280 bme;// BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Function that gets current epoch time
unsigned long getTime() {
    time_t now;
    struct tm timeinfo;
    if (!getLocalTime(&timeinfo)) {
        //Serial.println("Failed to obtain time");
        return(0);
    }
    time(&now);
    return now;
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["time"] = String(getTime());
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Read file from SPIFFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent += file.readStringUntil('\n');
        break;
    }
}

```

```

    }
    file.close();
    return fileContent;
}

// Append data to file in SPIFFS
void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}

// Delete File
void deleteFile(fs::FS &fs, const char * path){
    Serial.printf("Deleting file: %s\r\n", path);
    if(fs.remove(path)){
        Serial.println("- file deleted");
    } else {
        Serial.println("- delete failed");
    }
}

// Get file size
int getFileSize(fs::FS &fs, const char * path){
    File file = fs.open(path);

    if(!file){
        Serial.println("Failed to open file for checking size");
        return 0;
    }
    Serial.print("File size: ");
    Serial.println(file.size());

    return file.size();
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
}

```

```

    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    Serial.print("ok");
    initBME();
    initWiFi();
    initSPIFFS();

    // Create a data.txt file
    bool fileexists = SPIFFS.exists(dataPath);
    Serial.print(fileexists);
    if(!fileexists) {
        Serial.println("File doesn't exist");
        Serial.println("Creating file...");
        // Prepare readings to add to the file
        String message = getSensorReadings() + ",";
        // Append data to file to create it
        appendFile(SPIFFS, dataPath, message.c_str());
    }
    else {
        Serial.println("File already exists");
    }
    //file.close();

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

    server.serveStatic("/", SPIFFS, "/");

    // Request for the latest sensor readings
    server.on("/readings", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/data.txt", "text/txt");
    });

    // Request for the latest sensor readings
    server.on("/view-data", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/data.txt", "text/txt");
    });

    // Request for the latest sensor readings
    server.on("/delete-data", HTTP_GET, [] (AsyncWebServerRequest *request){
        deleteFile(SPIFFS, dataPath);
        request->send(200, "text/plain", "data.txt has been deleted.");
    });

    events.onConnect([] (AsyncEventSourceClient *client){
        if(client->lastId()){


```

```

        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);

configTime(0, 0, ntpServer);

// Start server
server.begin();

events.send(getSensorReadings().c_str(),"new_readings" ,millis());
}

void loop() {
if ((millis() - lastTime) > timerDelay) {

    // Send Events to the client with the Sensor Readings
    events.send("ping",NULL,millis());
    events.send(getSensorReadings().c_str(),"new_readings" ,millis());

    String message = getSensorReadings() + ",";
    if ((getFileSize(SPIFFS, dataPath))>= 3400){
        Serial.print("Too many data points, deleting file...");
        // Comment the next two lines if you don't want to delete the data file automatically.
        // It won't log more data into the file
        deleteFile(SPIFFS, dataPath);
        appendFile(SPIFFS, "/data.txt", message.c_str());
    }
    else{
        // Append new readings to the file
        appendFile(SPIFFS, "/data.txt", message.c_str());
    }

    lastTime = millis();
    Serial.print(readFile(SPIFFS, dataPath));
}
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include <NTPClient.h>
#include <WiFiUdp.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create an Event Source on /events
AsyncEventSource events("/events");

// Define NTP Client to get time
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// File name where readings will be saved
const char* dataPath = "/data.txt";

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 1800000;

// Create a sensor object
Adafruit_BME280 bme;// BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
    }
}
```

```

        while (1);
    }

// Function that gets current epoch time
unsigned long getTime() {
    timeClient.update();
    unsigned long now = timeClient.getEpochTime();
    return now;
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["time"] = String(getTime());
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Read file from LittleFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path, "r");
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent += file.readStringUntil('\n');
        break;
    }
    file.close();
    return fileContent;
}

// Append data to file in LittleFS
void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, "a");
    if(!file){

```

```

        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}

// Delete File
void deleteFile(fs::FS &fs, const char * path){
    Serial.printf("Deleting file: %s\r\n", path);
    if(fs.remove(path)){
        Serial.println("- file deleted");
    } else {
        Serial.println("- delete failed");
    }
}

// Get file size
int getFileSize(fs::FS &fs, const char * path){
    File file = fs.open(path, "r");

    if(!file){
        Serial.println("Failed to open file for checking size");
        return 0;
    }
    Serial.print("File size: ");
    Serial.println(file.size());

    return file.size();
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {
    // Serial port for debugging purposes
    Serial.begin(115200);
    Serial.print("ok");
    initBME();
    initWiFi();
    initFS();
}

```

```

// Initialize a NTPClient to get time
timeClient.begin();
// Set offset time in seconds to adjust for your timezone, for example:
// GMT +1 = 3600
// GMT +8 = 28800
// GMT -1 = -3600
// GMT 0 = 0
timeClient.setTimeOffset(0);

// Create a data.txt file
bool fileexists = LittleFS.exists(dataPath);
Serial.print(fileexists);
if(!fileexists) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    // Prepare readings to add to the file
    String message = getSensorReadings() + ",";
    // Append data to file to create it
    appendFile(LittleFS, dataPath, message.c_str());
}
else {
    Serial.println("File already exists");
}

// Web Server Root URL
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/index.html", "text/html");
});

server.serveStatic("/", LittleFS, "/");

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/data.txt", "text/txt");
});

// Request for the latest sensor readings
server.on("/view-data", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/data.txt", "text/txt");
});

// Request for the latest sensor readings
server.on("/delete-data", HTTP_GET, [](AsyncWebServerRequest *request){
    deleteFile(LittleFS, dataPath);
    request->send(200, "text/plain", "data.txt has been deleted.");
});

events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
});

```

```

    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});

server.addHandler(&events);

// Start server
server.begin();

events.send(getSensorReadings().c_str(),"new_readings", millis());

}

void loop() {
if ((millis() - lastTime) > timerDelay) {

    // Send Events to the client with the Sensor Readings Every 3 seconds
    events.send("ping",NULL,millis());
    events.send(getSensorReadings().c_str(),"new_readings" ,millis());
    String message = getSensorReadings() + ",";

    if ((getFileSize(LittleFS, dataPath))>= 3400){
        Serial.print("Too many data points, deleting file...");
        // Uncomment the next two lines if you don't want to delete the data file automatically
        // It won't log more data into the file
        deleteFile(LittleFS, dataPath);
        appendFile(LittleFS, "/data.txt", message.c_str());
    }
    else{
        // Append new readings to the file
        appendFile(LittleFS, "/data.txt", message.c_str());
    }

    lastTime = millis();

    Serial.print(readFile(LittleFS, dataPath));
}
}

```

How the Code Works

This code is similar to that in the previous project, but adds some functionality to save the readings in a file and get time from an NTP server. We'll take a look at the most relevant parts for this project.

Including Libraries

First, you need to include the required libraries. If you're using an ESP32 you load these libraries:

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include "time.h"
#include <WiFiUdp.h>
```

The `time.h` and `WiFiUdp.h` libraries are needed to get the time from an NTP Server.

If you're using the ESP8266 board, you need to load the following libraries instead (notice that you need to include the `NTPClient` library):

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>
#include <NTPClient.h>
#include <WiFiUdp.h>
```

NTP Server

We'll request time from `pool.ntp.org`:

```
// NTP server to request epoch time
const char* ntpServer = "pool.ntp.org";
```

If you're using an ESP8266, you need to define an NTP Client as follows:

```
// Define NTP Client to get time
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org");
```

We recommend taking a look at the following tutorials to better understand how to get date and time using the ESP32 and ESP8266 boards:

- [Get Epoch/Unix Time with the ESP32](#)
- [Get Epoch/Unix Time with the ESP8266](#)

data.txt file

We'll save the readings in a file called *data.txt*. Define the path for that file in the `dataPath` variable:

```
const char* dataPath = "/data.txt";
```

Sampling Rate

In the `timerDelay` variable define how frequently you want to get readings. In this example, we'll get it every 30 minutes (1800000 milliseconds).

```
unsigned long timerDelay = 1800000;
```

getTime()

The `getTime()` function returns the time in epoch time. This is how the function looks for the ESP32 board:

```
unsigned long getTime() {
    time_t now;
    struct tm timeinfo;
    if (!getLocalTime(&timeinfo)) {
        //Serial.println("Failed to obtain time");
        return(0);
    }
    time(&now);
    return now;
}
```

And this is how it looks for the ESP8266:

```
unsigned long getTime() {
    timeClient.update();
```

```
    unsigned long now = timeClient.getEpochTime();
    return now;
}
```

getSensorReadings()

The `getSensorReadings()` function is slightly different from previous project. In this particular example, we also add the time to the `readings` JSON variable as follows:

```
String getSensorReadings(){
    readings["time"] = String(getTime());
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}
```

So, here's an example of how the `readings` JSON variable looks like:

```
{
  "time" : "1610021148",
  "temperature" : "25",
  "humidity" : "50",
  "pressure" : "1015"
}
```

Handle Files

We need to create several functions to handle files: read a file, append data to a file, delete a file and get the file size.

Read a File

The `readFile()` function reads and returns the content of a file.

```
// Read file from SPIFFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }
```

```
String fileContent;
while(file.available()){
    fileContent += file.readStringUntil('\n');
    break;
}
file.close();
return fileContent;
}
```

Append Data to a File

The `appendFile()` function appends new data to a file without overwriting what is already on the file.

```
// Append data to file in SPIFFS
void appendFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Appending to file: %s\r\n", path);

    File file = fs.open(path, FILE_APPEND);
    if(!file){
        Serial.println("- failed to open file for appending");
        return;
    }
    if(file.print(message)){
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}
```

Delete a File

The `deleteFile()` function, as the name suggests, deletes a file.

```
// Delete File
void deleteFile(fs::FS &fs, const char * path){
    Serial.printf("Deleting file: %s\r\n", path);
    if(fs.remove(path)){
        Serial.println("- file deleted");
    } else {
        Serial.println("- delete failed");
    }
}
```

Getting File Size

The `getFileSize()` function returns the size of the file in bytes.

```
// Get file size
int getFileSize(fs::FS &fs, const char * path){
    File file = fs.open(path);

    if(!file){
        Serial.println("Failed to open file for checking size");
        return 0;
    }
    Serial.print("File size: ");
    Serial.println(file.size());

    return file.size();
}
```

Creating a New File

In the `setup()`, after initializing the BME280 sensor, Wi-Fi and SPIFFS, create a new file called `data.txt` (`dataPath`) if it doesn't exist.

```
// Create a data.txt file
bool fileexists = SPIFFS.exists(dataPath);
Serial.print(fileexists);
if(!fileexists) {
    Serial.println("File doesn't exist");
    Serial.println("Creating file...");
    // Prepare readings to add to the file
    String message = getSensorReadings() + ",";
    // Append data to file to create it
    appendFile(SPIFFS, dataPath, message.c_str());
}
else {
    Serial.println("File already exists");
}
```

Handle Requests

When the ESP receives a request on the root URL, send the HTML and all the other necessary files:

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

server.serveStatic("/", SPIFFS, "/");
```

When each client first connects to the ESP web server, it makes a request for the stored sensor readings. The ESP sends back the contents of the *data.txt* file as follows:

```
// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/data.txt", "text/txt");
});
```

The **View Raw Data button** on the web page makes a request on the `/view-data` URL when clicked. When that happens, we respond with the content of the *data.txt* file:

```
// Request for raw data
server.on("/view-data", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/data.txt", "text/txt");
});
```

When you click the **Delete Data** button, it makes a request on the `/delete-data` path. When that happens, we call the `deleteFile()` function to delete the saved file:

```
// Request to delete data
server.on("/delete-data", HTTP_GET, [](AsyncWebServerRequest *request){
    deleteFile(SPIFFS, dataPath);
    request->send(200, "text/plain", "data.txt has been deleted.");
});
```

Finally, configure an event source:

```
events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n", client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);
```

loop()

In the `loop()`, we send an event to the server every 30 minutes with the latest sensor readings.

```
events.send(getSensorReadings().c_str(), "new_readings" ,millis());
```

After sending the readings, we need to save them in the file. As we've mentioned previously, each reading is followed by a comma. So, we need to add that comma to the message we want to save:

```
String message = getSensorReadings() + ",";
```

After that, we check the current size of the file. If the file is bigger than 3400 bytes (it saves approximately 40 data points), we delete the file's content before writing a new data point.

```
if ((getFileSize(SPIFFS, dataPath))>= 3400){
    Serial.print("Too many data points, deleting file...");
    // Comment the next two lines if you don't want to delete the data file automatically.
    // It won't log more data into the file
    deleteFile(SPIFFS, dataPath);
    appendFile(SPIFFS, "/data.txt", message.c_str());
}
```

Otherwise, we simply append data to the file.

```
else{
    // Append new readings to the file
    appendFile(SPIFFS, "/data.txt", message.c_str());
}
```

If you don't want the file to be deleted automatically, you can comment these next lines, as shown in the code:

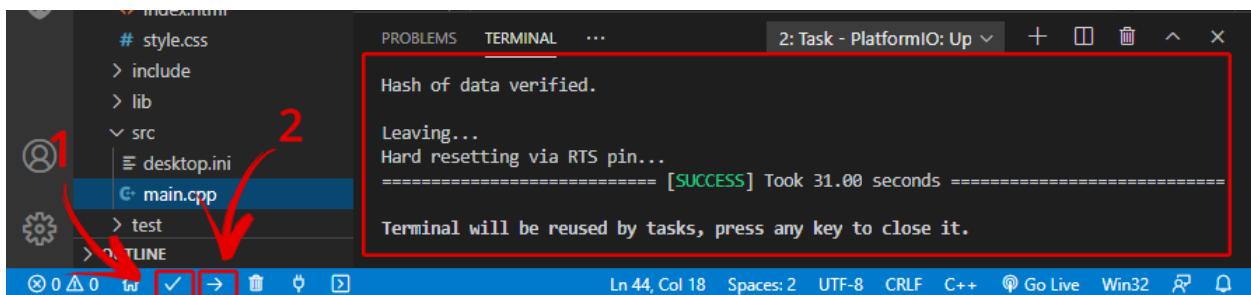
```
// Comment the next two lines if you don't want to delete the data file automatically.
// It won't log more data into the file
deleteFile(SPIFFS, dataPath);
appendFile(SPIFFS, "/data.txt", message.c_str());
```

However, you should notice that it won't record any more data points until you delete the file manually on the web page by clicking the **Delete File** button.

Instead of 40, you can save more data points. However, as file size increases, it will take more and more time to load all points on the web page when you access it for the first time.

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.



Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

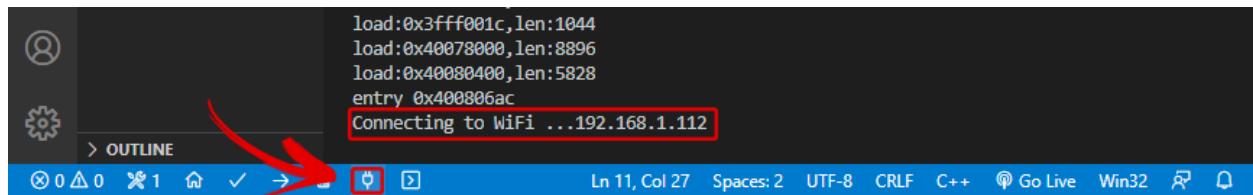
Finally, upload the files (*index.html*, *style.css*, *script.js*, and *favicon.png*) to the filesystem:

1. Click the PIO icon at the left sidebar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Select **Build Filesystem Image**.

- Finally, click **Upload Filesystem Image**.

Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.



Open a browser on your local network and type the ESP32 or ESP8266 IP address. The page displays two charts. A new data point is added every 30 minutes to a total of 40 points.



As in the previous example, you can select each point to see the exact timestamp.

You can close your web browser window and when you open it again, you'll still see the data points.

Download Project Folder

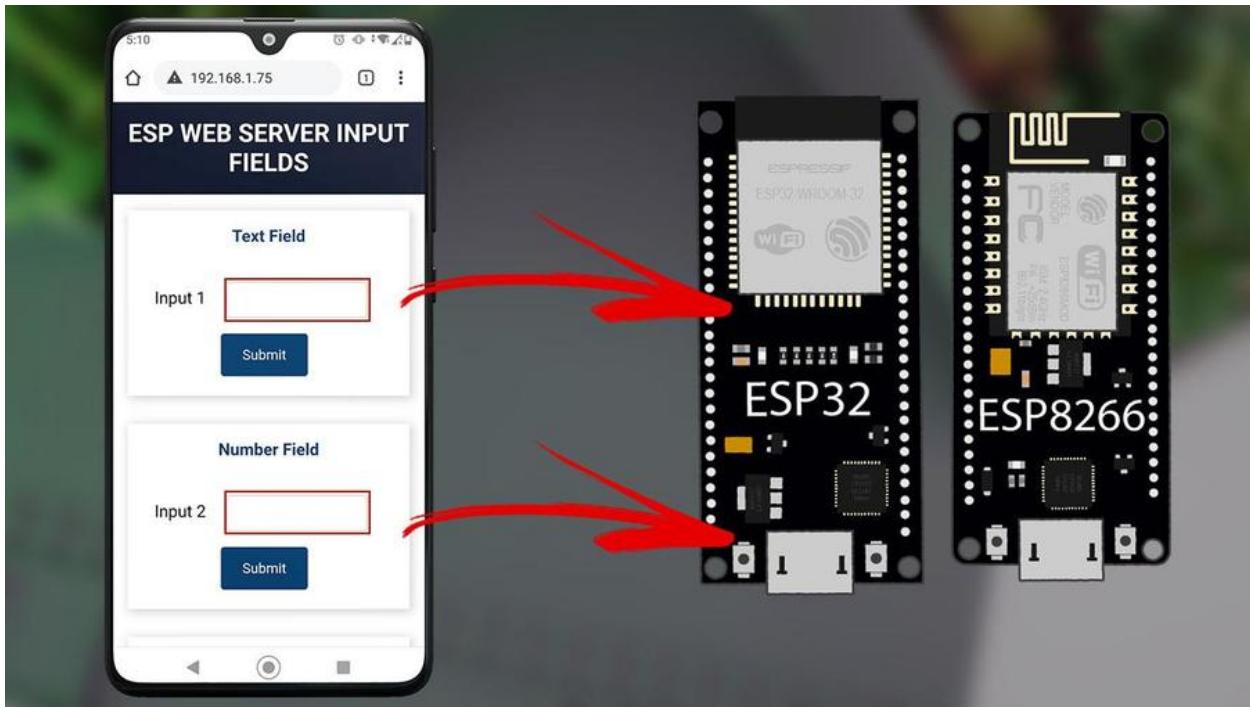
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to save readings to a file and how to display those readings on charts on a web page. As an example, we created a *data.txt* file in the filesystem. If you want to log data frequently for extended periods of time, we recommend using a microSD card (not covered in this eBook).

4.1 - Web Server with Input Fields (HTML Form)



In this Unit, you'll learn how to add input fields to your web pages to pass values to your ESP using HTML forms. Then, you can use those values as variables in your code. With this method, you avoid hard-coding variables because you can create an input field in a web page to update any variable with a new value. This can be especially useful to set threshold values, set and change API keys, etc.

The variables will be saved permanently on files saved in the flash memory (SPIFFS and LittleFS). This way, those values are saved even if the ESP resets or loses power.

Project Overview

Here's the web page we'll build for this project.

ESP WEB SERVER INPUT FIELDS

Text Field

Input 1

Submit

Number Field

Input 2

Submit

Text Value

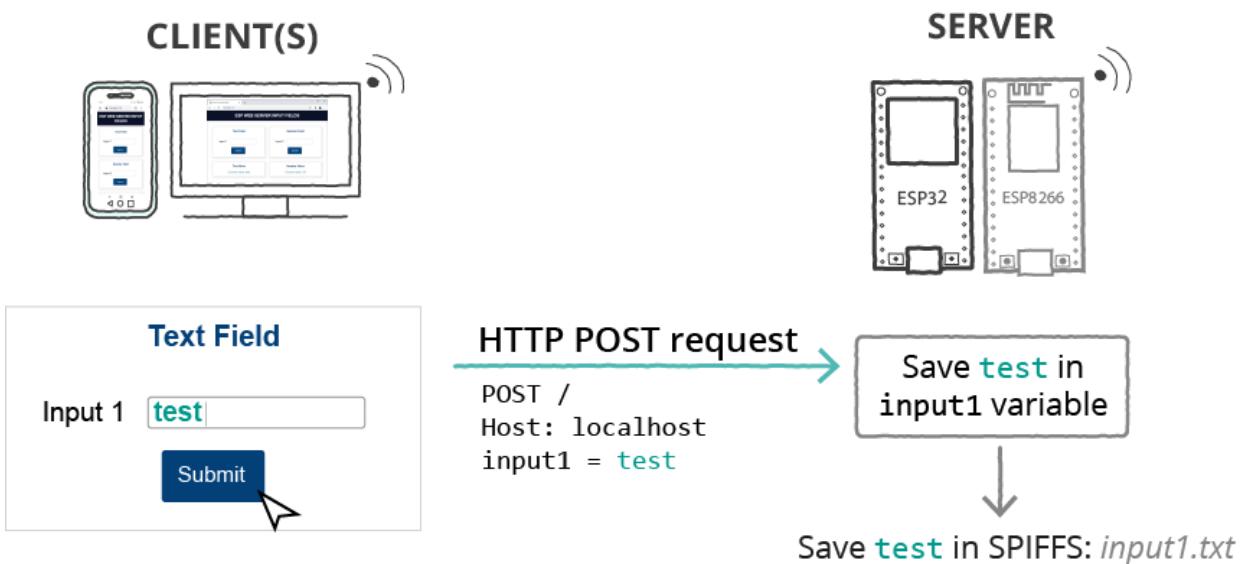
Current value: test

Number Value

Current value: 26

The web page displays two input fields: a text field and a number field. The current value of each field is also displayed.

How it Works?



- When you submit something in the text input field, the client makes an HTTP POST request on the / URL with the value of the `input1` field.
- The server receives that request and saves the value in a variable (in this case, variable `input1`). That variable is saved permanently in a file called `input1.txt` in the ESP filesystem.
- At the same time, the web page refreshes and the new input field value is displayed on the web page.
- The next time the ESP runs, it will get the last `input1` value from the `input1.txt` file saved in SPIFFS.
- This is useful if you want to save threshold values or other configurations.

Building the Web Page

To build the web page for this project, place the following files in your project's *data* folder:

- `index.html`
- `style.css`
- `script.js`
- `favicon.png`

HTML File

Here's the text you should copy to your `index.html` file.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP IOT DASHBOARD</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnnOCqbTlWIj8LyTjo7mOUSTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
  <link rel="stylesheet" type="text/css" href="style.css">
```

```

</head>
<body>
  <div class="topnav">
    <h1>ESP WEB SERVER INPUT FIELDS</h1>
  </div>
  <div class="content">
    <div class="card-grid">
      <div class="card">
        <form action="/" method="POST">
          <p class="card-title">Text Field</p>
          <p>
            <label for="input1">Input 1</label>
            <input type="text" id ="input1" name="input1">
            <input type ="submit" value ="Submit">
          </p>
        </form>
      </div>
      <div class="card">
        <form action="/" method="POST">
          <p class="card-title">Number Field</p>
          <p>
            <label for="input2">Input 2</label>
            <input type="number" id ="input2" name="input2">
            <input type ="submit" value ="Submit">
          </p>
        </form>
      </div>
      <div class="card">
        <p class="card-title">Text Value</p>
        <p class="value">Current value: <span id="textFieldValue"></span></p>
      </div>
      <div class="card">
        <p class="card-title">Number Value</p>
        <p class="value">Current value: <span id="numberFieldValue"></span></p>
      </div>
    </div>
    <script src="script.js"></script>
  </body>
</html>

```

This file creates two cards with input fields: one text field and a number field; and two cards to display the last values submitted in the fields.

HTML Forms

The `<form>` tag is used to create an HTML form to collect user input. The user input can then be sent to the server (ESP32 or ESP8266) for processing. Based on the values collected on the form, your ESP board may perform different tasks.

Alternatively, the user input can be used on the client side (browser) to update the interface in some way using JavaScript, for example.

In our case, we want to send the value submitted in the input field to the server.

Here's the HTML form for the Text Input field.

```
<form action="/" method="POST">
  <p class="card-title">Text Field</p>
  <p>
    <label for="input1">Input 1</label>
    <input type="text" id="input1" name="input1">
    <input type="submit" value="Submit">
  </p>
</form>
```

The HTML form contains different form elements. All the form elements are enclosed inside this `<form>` tag. It contains controls (the number input field):

```
<input type="number" id="input2" name="input2">
```

And labels for those controls (Input 2).

```
<label for="input2">Input 2</label>
```

Additionally, the `<form>` tag must include the `action` attribute that specifies what you want to do when the form is submitted (it redirects to the / root URL, so that we remain on the same page). In our case, we want to send that data to the server (ESP32/ESP8266) when the user clicks the **Submit** button. The `method` attribute specifies the HTTP method (GET or POST) used when submitting the form data. In this case, we'll use HTTP POST method.

```
<form action="/" method="POST">
```

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. In this case, if you submit **test** in the input field, the body of the HTTP POST request would look like this:

```
POST /  
Host: localhost  
input1 = test
```

The `<input type="submit" value="Submit">` creates a submit button with the text **"Submit"**. When you click this button, the data submitted in the form is sent to the server.

A similar process happens for the other input field. But, this time the input field is of type `number` (it only accepts number values).

```
<form action="/" method="POST">  
  <p class="card-title">Number Field</p>  
  <p>  
    <label for="input2">Input 2</label>  
    <input type="number" id ="input2" name="input2">  
    <input type ="submit" value ="Submit">  
  </p>  
</form>
```

Besides the HTML forms, there are two more cards to display the last value inserted in the input fields.

```
<div class="card">  
  <p class="card-title">Text Value</p>  
  <p class="value">Current value: <span id="textFieldValue"></span></p>  
</div>  
<div class="card">  
  <p class="card-title">Number Value</p>  
  <p class="value">Current value: <span id="numberFieldValue"></span></p>  
</div>
```

Those are paragraphs with `` tags that will be used to insert the values. The `id` for the text field is `textFieldValue`, and the `id` for the number field is `numberFieldValue`.

CSS File

Copy the following styles to your *style.css* file.

```
html {
    font-family: Arial, Helvetica, sans-serif;
    display: inline-block;
    text-align: center;
}
h1 {
    font-size: 1.8rem;
    color: white;
}
p {
    font-size: 1.4rem;
}
.topnav {
    overflow: hidden;
    background-color: #0A1128;
}

body {
    margin: 0;
}
.content {
    padding: 5%;
}
.card-grid {
    max-width: 800px;
    margin: 0 auto;
    display: grid;
    grid-gap: 2rem;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
}
.card {
    background-color: white;
    box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}
.card-title {
    font-size: 1.2rem;
    font-weight: bold;
    color: #034078
}
input[type=submit] {
    border: none;
    color: #FEFCFB;
    background-color: #034078;
    padding: 15px 15px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    width: 100px;
```

```
margin-right: 10px;
border-radius: 4px;
transition-duration: 0.4s;
}
input[type=submit]:hover {
background-color: #1282A2;
}
input[type=text], input[type=number], select {
width: 50%;
padding: 12px 20px;
margin: 18px;
display: inline-block;
border: 1px solid #ccc;
border-radius: 4px;
box-sizing: border-box;
}
label {
font-size: 1.2rem;
}
.value{
font-size: 1.2rem;
color: #1282A2;
}
```

The following lines style the **Submit** button. The selector for the submit button is `input[type=submit]`.

```
input[type=submit] {
border: none;
color: #FEFCFB;
background-color: #034078;
padding: 15px 15px;
text-align: center;
text-decoration: none;
display: inline-block;
font-size: 16px;
width: 100px;
margin-right: 10px;
border-radius: 4px;
transition-duration: 0.4s;
}
input[type=submit]:hover {
background-color: #1282A2;
}
```

The input fields for the text and the number are styled below:

```
input[type=text], input[type=number], select {
width: 50%;
padding: 12px 20px;
margin: 18px;
display: inline-block;
```

```
border: 1px solid #ccc;
border-radius: 4px;
box-sizing: border-box;
}
```

JavaScript File

Copy the following code to your *script.js* file.

```
// Get current sensor readings when the page loads
window.addEventListener('load', getValues);

// Function to get current values on the webpage when it loads/refreshes
function getValues(){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var myObj = JSON.parse(this.responseText);
            console.log(myObj);
            document.getElementById("textFieldValue").innerHTML = myObj.textValue;
            document.getElementById("numberFieldValue").innerHTML = myObj.numberValue;
        }
    };
    xhr.open("GET", "/values", true);
    xhr.send();
}
```

Let's take a look at this JavaScript code to see how it works.

We add an event listener that will call the `getValues` function when the page loads:

```
window.addEventListener('load', getValues);
```

The `getValues()` function makes an HTTP request on the `/values` URL:

```
xhr.open("GET", "/values", true);
```

When the server receives this request, it responds with a JSON variable that contains the value of the text and number fields.

Here's an example on how the JSON variable looks:

```
{
  "textValue": "Hello",
  "numberValue": "26"
}
```

When the response is ready, put the values in the right HTML elements.

```
var myObj = JSON.parse(this.responseText);
console.log(myObj);
document.getElementById("textFieldValue").innerHTML = myObj.textValue;
document.getElementById("numberFieldValue").innerHTML = myObj.numberValue;
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed=115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "input1";
const char* PARAM_INPUT_2 = "input2";

// Variables to save values from HTML form
String input1;
String input2;

// File paths to save input values permanently
const char* input1Path = "/input1.txt";
const char* input2Path = "/input2.txt";

JSONVar values;

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Read File from SPIFFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent = file.readStringUntil('\n');
        break;
    }
}
```

```

    }
    return fileContent;
}

// Write file to SPIFFS
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

String getCurrentInputValues(){
    values["textValue"] = input1;
    values["numberValue"] = input2;
    String jsonString = JSON.stringify(values);
    return jsonString;
}

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

    // Load values saved in SPIFFS
    input1 = readFile(SPIFFS, input1Path);
    input2 = readFile(SPIFFS, input2Path);

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

```

```

server.serveStatic("/", SPIFFS, "/");

server.on("/values", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getCurrentInputValues();
    request->send(200, "application/json", json);
    json = String();
});

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    for(int i=0;i<params;i++){
        AsyncWebParameter* p = request->getParam(i);
        if(p->isPost()){
            // HTTP POST input1 value
            if (p->name() == PARAM_INPUT_1) {
                input1 = p->value().c_str();
                Serial.print("Input 1 set to: ");
                Serial.println(input1);
                // Write file to save value
                writeFile(SPIFFS, input1Path, input1.c_str());
            }
            // HTTP POST input2 value
            if (p->name() == PARAM_INPUT_2) {
                input2 = p->value().c_str();
                Serial.print("Input 2 set to: ");
                Serial.println(input2);
                // Write file to save value
                writeFile(SPIFFS, input2Path, input2.c_str());
            }
            //Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value().c_str());
        }
        request->send(SPIFFS, "/index.html", "text/html");
    });
}

server.begin();
}

void loop() {
/*Serial.println(readFile(SPIFFS, input1Path));
Serial.println(readFile(SPIFFS, input2Path));
delay(10000);*/
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "input1";
const char* PARAM_INPUT_2 = "input2";

// Variables to save values from HTML form
String input1;
String input2;

// File paths to save input values permanently
const char* input1Path = "/input1.txt";
const char* input2Path = "/input2.txt";

JSONVar values;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Read File from LittleFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path, "r");
    if(!file || file.isDirectory()){

```

```

        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent = file.readStringUntil('\n');
        break;
    }
    return fileContent;
}

// Write file to LittleFS
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, "w");
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

String getCurrentInputValues(){
    values["textValue"] = input1;
    values["numberValue"] = input2;
    String jsonString = JSON.stringify(values);
    return jsonString;
}

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

    // Load values saved in LittleFS
}

```

```

input1 = readFile(LittleFS, input1Path);
input2 = readFile(LittleFS, input2Path);

// Web Server Root URL
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/index.html", "text/html");
});

server.serveStatic("/", LittleFS, "/");

server.on("/values", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getCurrentInputValues();
    request->send(200, "application/json", json);
    json = String();
});

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    for(int i=0;i<params;i++){
        AsyncWebParameter* p = request->getParam(i);
        if(p->isPost()){
            // HTTP POST input1 value
            if (p->name() == PARAM_INPUT_1) {
                input1 = p->value().c_str();
                Serial.print("Input 1 set to: ");
                Serial.println(input1);
                // Write file to save value
                writeFile(LittleFS, input1Path, input1.c_str());
            }
            // HTTP POST input2 value
            if (p->name() == PARAM_INPUT_2) {
                input2 = p->value().c_str();
                Serial.print("Input 2 set to: ");
                Serial.println(input2);
                // Write file to save value
                writeFile(LittleFS, input2Path, input2.c_str());
            }
            //Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value().c_str());
        }
    }
    request->send(LittleFS, "/index.html", "text/html");
});

server.begin();
}

void loop() {
/*Serial.println(readFile(LittleFS, input1Path));
Serial.println(readFile(LittleFS, input2Path));
delay(10000);*/
}

```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";
```

How The Code Works

Let's take a look at the relevant parts of this code.

The PARAM_INPUT_1 and PARAM_INPUT_2 variables will be used to search for the input field values in the body of the HTTP POST request.

```
// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "input1";
const char* PARAM_INPUT_2 = "input2";
```

As a reminder, the body of the HTTP POST request is like this if you click the submit button for the first input field:

```
POST /
Host: localhost

```

Or like this if you click the second input field:

```
POST /
Host: localhost

```

The following variables will save the values passed in the HTTP POST request:

```
// Variables to save values from HTML form
String input1;
String input2;
```

The values of these variables will be permanently stored in SPIFFS or LittleFS files.

The file that saves `input1` is `input1.txt` and the file that saves `input2` is `input2.txt`.

```
// File paths to save input values permanently
const char* input1Path = "/input1.txt";
const char* input2Path = "/input2.txt";
```

The `values` JSON variable saves `input1` and `input2` values to be sent to the client.

```
JSONVar values;
```

Handle Files

The `readFile()` function reads and returns the content of a file:

```
// Read File from SPIFFS
String readFile(FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent = file.readStringUntil('\n');
        break;
    }
    return fileContent;
}
```

The `writeFile()` function writes into a file:

```
// Write file to SPIFFS
void writeFile(FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
}
```

getCurrentInputValues()

The `getCurrentInputValues()` puts `input1` and `input2` values into the `values` JSON variable and returns a JSON string variable (`jsonString`).

```
String getCurrentInputValues(){
    values["textValue"] = input1;
    values["numberValue"] = input2;
    String jsonString = JSON.stringify(values);
```

```
    return jsonString;
}
```

In the `setup()`, after initializing Wi-Fi and the filesystem, load the current values saved in the `input1.txt` and `input2.txt` files:

```
// Load values saved in SPIFFS
input1 = readFile(SPIFFS, input1Path);
input2 = readFile(SPIFFS, input2Path);
```

This assures that you'll always have the current values, even if the ESP resets or loses power.

Handle Requests

When the web page loads, it makes an HTTP GET request on the `/values` URL. When that happens, the server responds with the current values of `input1` and `input2` in JSON format (call the `getCurrentInputValues()` function).

```
server.on("/values", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getCurrentInputValues();
    request->send(200, "application/json", json);
    json = String();
});
```

On the web page, when you submit the form, it makes an HTTP POST request as we've seen previously. We need to handle what to do when that happens:

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

First, we search for parameters in the HTTP POST request:

```
int params = request->params();
for(int i=0;i<params;i++){
    AsyncWebParameter* p = request->getParam(i);
    if(p->isPost()){
```

If any of the parameters is equal to `PARAM_INPUT_1`, we know that you've submitted something on the `input1` field. If that's the case, we get the value of that parameter, save it in the `input1` variable and in the corresponding file in SPIFFS or LittleFS.

```

// HTTP POST input1 value
if (p->name() == PARAM_INPUT_1) {
    input1 = p->value().c_str();
    Serial.print("Input 1 set to: ");
    Serial.println(input1);
    // Write file to save value
    writeFile(SPIFFS, input1Path, input1.c_str());
}

```

We follow a similar procedure for `PARAM_INPUT_2`, but we save the value in the `input2` variable.

```

if (p->name() == PARAM_INPUT_2) {
    input2 = p->value().c_str();
    Serial.print("Input 2 set to: ");
    Serial.println(input2);
    // Write file to save value
    writeFile(SPIFFS, input2Path, input2.c_str());
}

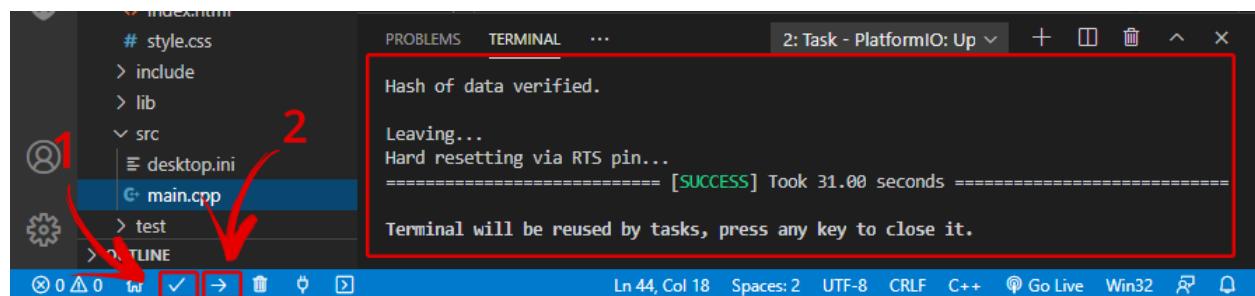
```

Finally, we respond with the content of the HTML page, it will load the page and update the values.

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

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload code to your board.

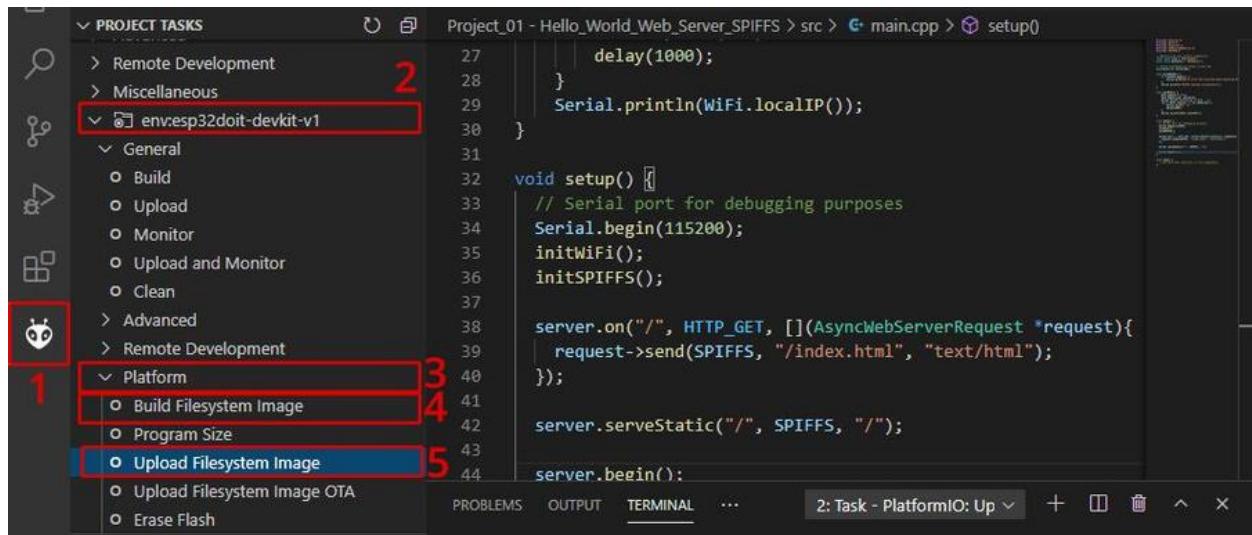


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

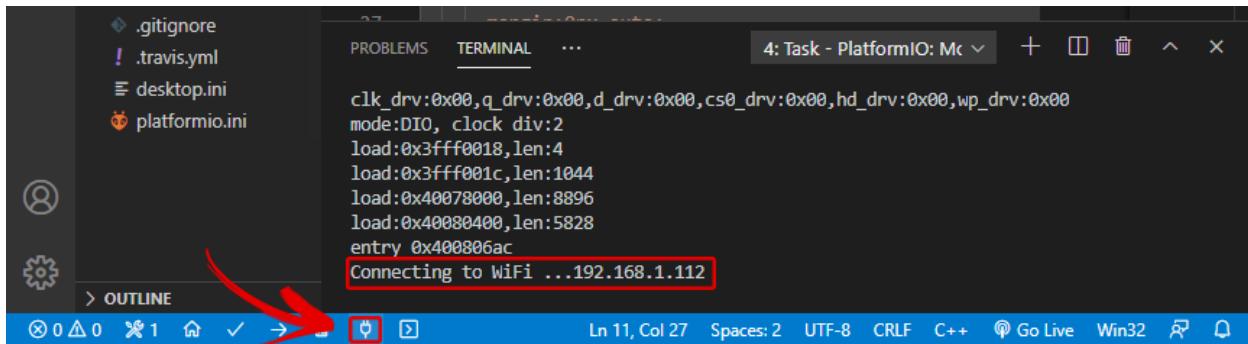
Finally, upload the files (*index.html*, *style.css*, *script.js* and *favicon.png*) to the filesystem:

1. Click on the PIO icon at the left bar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Click on **Build Filesystem Image**.
5. Finally, click on **Upload Filesystem Image**.



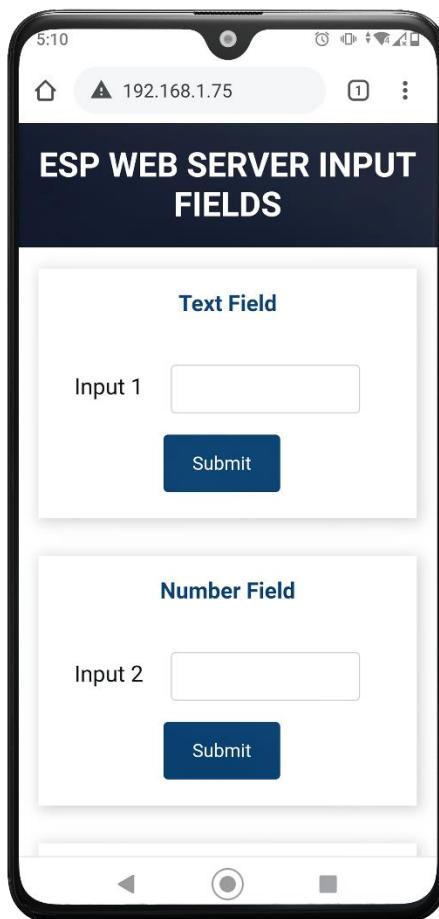
Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.



```
.gitignore  
.travis.yml  
desktop.ini  
platformio.ini  
  
PROBLEMS TERMINAL ...  
4: Task - PlatformIO: Mc  
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00  
mode:DIO, clock div:2  
load:0x3fff0018,len:4  
load:0x3fff001c,len:1044  
load:0x40078000,len:8896  
load:0x40080400,len:5828  
entry 0x400806ac  
Connecting to WiFi ...192.168.1.112  
  
Ln 11, Col 27 Spaces: 2 UFT-8 CRLF C++ Go Live Win32
```

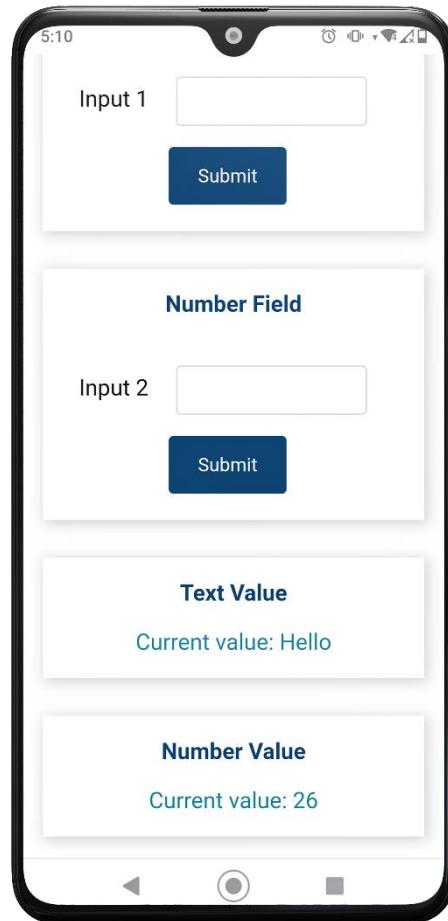
Open a browser on your local network and insert the ESP IP address. You should get access to the web page to insert data in the input fields.



Type something in the input fields. After submitting, you should get something as follows on the Serial Monitor indicating the value was successfully saved in the filesystem:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Input 1 set to: Hello
Writing file: /input1.txt
- file written
Input 2 set to: 26
Writing file: /input2.txt
- file written
```

After submitting the values, these are updated on the corresponding cards as shown below.



Download Project Folder

You can download the complete project folder for this project using the links below.

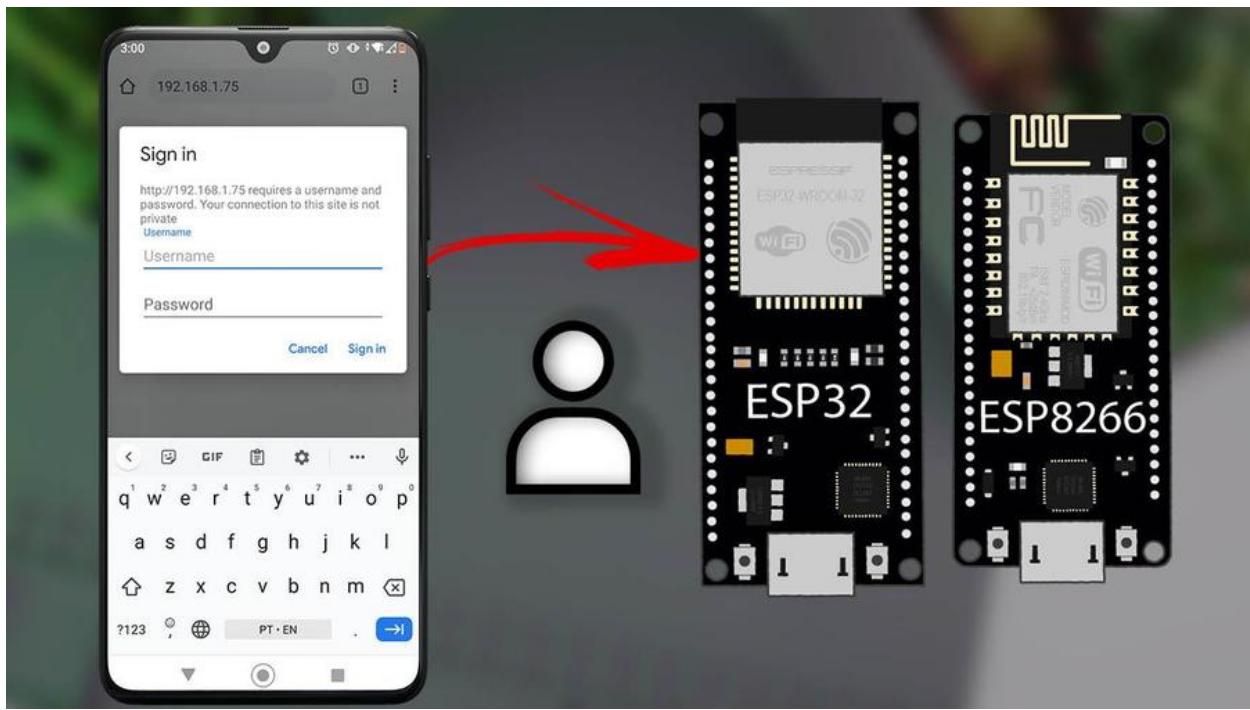
- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit you've learned how to add input fields to your web server. As an example, we've shown you how to add a text input field and a number input field and how to save the values submitted in the filesystem. This can be useful to set threshold values in your code or save settings that can change in the future, but not often (like API keys, for example).

The idea is that you use what you've learned here to complement your other projects.

4.2 – HTTP Authentication: Password Protected Web Server

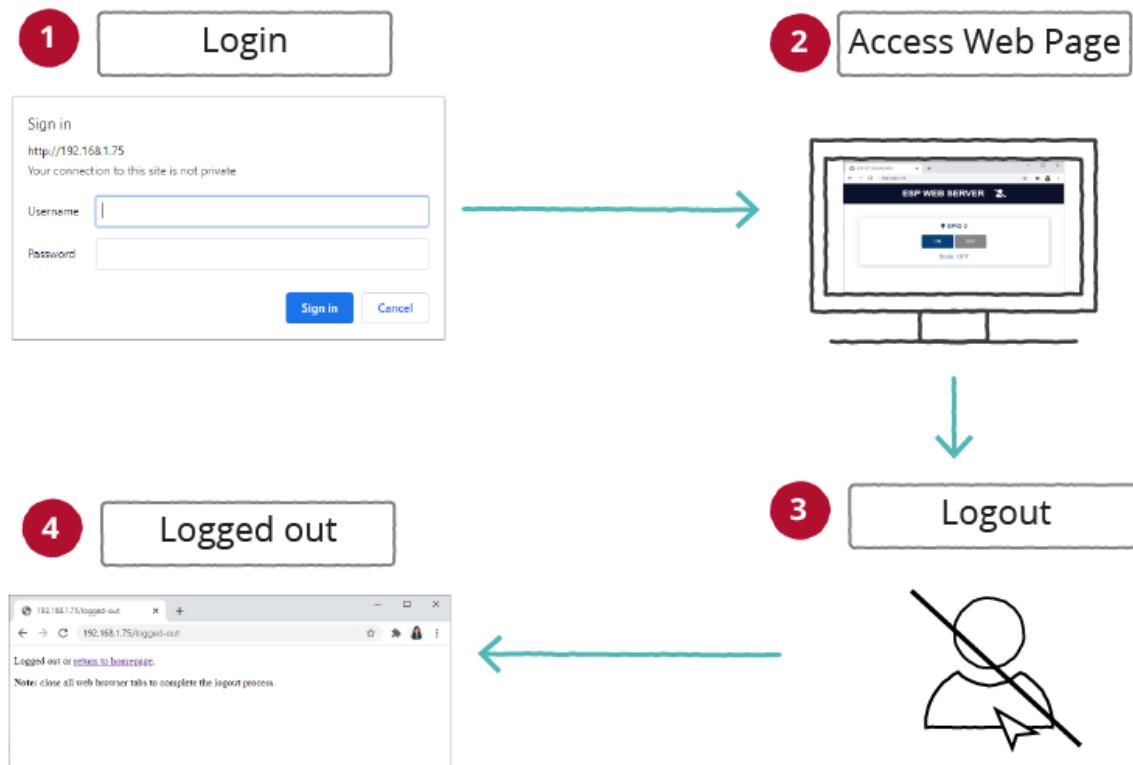


In this Unit, you'll learn how to add HTTP authentication with username and password to your ESP32 and ESP8266 web server projects. You can only access your web server if you type the correct username and password. If you logout, you can only access the web server again if you enter the right credentials. This authentication method can be applied to any web server built using the ESPAsyncWebServer library.

As an example, we'll add username and password to the project built in "Unit 2.3 - WebSocket Web Server: Control Outputs (ON/OFF Buttons)".

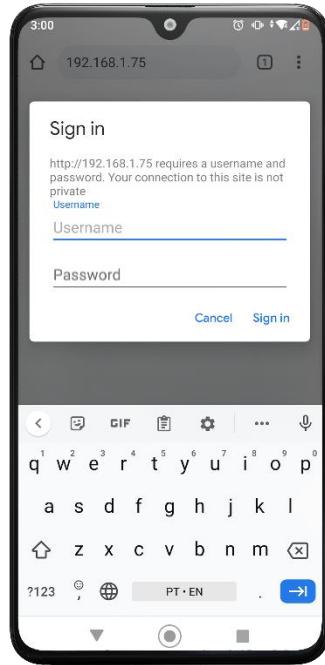
Project Overview

In this Unit, you'll learn how to password protect your web server. Here's how it works:



- When you try to access the web page on the ESP IP address, a window pops up asking for a username and password;
- To get access to the web page, you need to enter the correct username and password (defined in the ESP32/ESP8266 *main.cpp* file);
- There's a logout button on the web server. If you click the logout button, you'll be redirected to a logout page. Then, close all web browser tabs to complete the logout process;
- You can only access the web server again if you login with the proper credentials;

- If you try to access the web server from a different device (on the local network) you also need to login with the right credentials (even if you have a successful login on another device);
- The authentication is not encrypted.



We'll add authentication to the project built in Unit 2.3, but this can be used with any web server project.

Building the Web Page

To build the web page for this project, make sure you place all the following files inside the *data* folder in your project folder:

- *index.html*
- *logged-out.html*
- *style.css*
- *script.js*
- *favicon.png*

HTML File

Notice that you need two HTML files for this project: one to build the main page and another to build the logout page (the page that shows up right after clicking the logout button).

index.html

Here's the text you should copy to your *index.html* file. It is the same of Unit 2.3, but adds the logout button.

```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP IOT DASHBOARD</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css">
  </head>
  <body>
    <div class="topnav">
      <h1>ESP WEB SERVER &nbsp;&nbsp; <i class="fas fa-user-slash icon-pointer" onclick="logoutButton()"></i></span></h1>
    </div>
    <div class="content">
      <div class="card-grid">
        <div class="card">
          <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
          <p>
            <button class="button-on" id="bON">ON</button>
            <button class="button-off" id="bOFF">OFF</button>
          </p>
          <p class="state">State: <span id="state">%STATE%</span></p>
        </div>
      </div>
    </div>
    <script src="script.js"></script>
  </body>
</html>
```

We've added the logout button as an icon in the main top bar. You can place it wherever you want. When it's clicked, the `logoutButton()` JavaScript function is executed.

```
<h1>ESP WEB SERVER &nbsp;&nbsp; <i class="fas fa-user-slash icon-pointer" onclick="logoutButton()"></i></span></h1>
```

logged-out.html

Besides the `index.html` file, you also need to create a file for the logout page. Create a file called `logged-out.html`. Copy the following into that file:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <p>Logged out or <a href="/">return to homepage</a>.</p>
  <p><strong>Note:</strong> close all web browser tabs to complete the logout process.</p>
</body>
</html>
```

This creates a simple web page with some text and a link to return to homepage.

```
<p>Logged out or <a href="/">return to homepage</a>.</p>
```

CSS File

Copy the following to your `style.css` file. These are the same styles used in Unit 2.3.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  text-align: center;
}
h1 {
  font-size: 1.8rem;
  color: white;
}
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}
body {
  margin: 0;
```

```
}

.content {
  padding: 50px;
}

.card-grid {
  max-width: 800px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
}

.card {
  background-color: white;
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}

.card-title {
  font-size: 1.2rem;
  font-weight: bold;
  color: #034078
}

.state {
  font-size: 1.2rem;
  color: #1282A2;
}

button {
  border: none;
  color: #FEFCFB;
  padding: 15px 32px;
  text-align: center;
  text-decoration: none;
  font-size: 16px;
  width: 100px;
  border-radius: 4px;
  transition-duration: 0.4s;
}

.button-on {
  background-color: #034078;
}

.button-on:hover {
  background-color: #1282A2;
}

.button-off {
  background-color: #858585;
}

.button-off:hover {
  background-color: #252524;
}

.icon-pointer {
  cursor: pointer;
}
```

JavaScript File

Copy the following code to your *script.js* file. This is the same JavaScript used in Unit 2.3, but adds the *logoutButton()* function.

```
var gateway = `ws://${window.location.hostname}/ws`;
var websocket;

window.addEventListener('load', onload);

function onload(event) {
    initWebSocket();
    initButton();
}

function initWebSocket() {
    console.log('Trying to open a WebSocket connection...');
    websocket = new WebSocket(gateway);
    websocket.onopen = onOpen;
    websocket.onclose = onClose;
    websocket.onmessage = onMessage;
}

function onOpen(event) {
    console.log('Connection opened');
    websocket.send('hi');
}

function onClose(event) {
    console.log('Connection closed');
    setTimeout(initWebSocket, 2000);
}

function onMessage(event) {
    document.getElementById('state').innerHTML = event.data;
    console.log(event.data);
}

function initButton() {
    document.getElementById('bON').addEventListener('click', toggleON);
    document.getElementById('bOFF').addEventListener('click', toggleOFF);
}

function toggleON(event) {
    websocket.send('bON');
}

function toggleOFF(event) {
    websocket.send('bOFF');
}

function logoutButton() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "/logout", true);
    xhr.send();
    setTimeout(function(){ window.open("/logged-out","_self"); }, 1000);
}
```

The `logoutButton()` function makes an HTTP GET request on the `/logout` URL, so that the server removes the user authentication.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/logout", true);
xhr.send();
```

After one second (1000 milliseconds), you are redirected to the `/logged-out` page to complete the logout process.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the `platformio.ini` file and the `main.cpp` file inside the `src` folder.

platformio.ini file (ESP32)

The `platformio.ini` file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The `platformio.ini` file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it is in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Web Server HTTP Authentication credentials
const char* http_username = "admin";
const char* http_password = "admin";

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
bool ledState = 0;

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
```

```

    }
    Serial.println(WiFi.localIP());
}

// Replaces placeholder with LED state value
String processor(const String& var) {
    if(var == "STATE") {
        if(digitalRead(ledPin)) {
            ledState = 1;
            return "ON";
        }
        else {
            ledState = 0;
            return "OFF";
        }
    }
    return String();
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "bON") == 0) {
            ledState = 1;
            notifyClients("ON");
        }
        if (strcmp((char*)data, "bOFF") == 0) {
            ledState = 0;
            notifyClients("OFF");
        }
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
    }
}

```

```

        break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request) {
        if (!request->authenticate(http_username, http_password))
            return request->requestAuthentication();
        request->send(SPIFFS, "/index.html", "text/html", false, processor);
    });

    server.on("/logged-out", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/logged-out.html", "text/html", false, processor);
    });

    server.on("/logout", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(401);
    });

    server.serveStatic("/", SPIFFS, "/").setAuthentication(http_username, http_password);

    // Start server
    server.begin();
}

void loop() {
    ws.cleanupClients();
    digitalWrite(ledPin, ledState);
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <LittleFS.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID ";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Web Server HTTP Authentication credentials
const char* http_username = "admin";
const char* http_password = "admin";

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
bool ledState = 0;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

// Replaces placeholder with LED state value
String processor(const String& var) {
    if(var == "STATE") {
        if(digitalRead(ledPin)) {
            ledState = 0;
```

```

        return "ON";
    }
    else{
        ledState = 1;
        return "OFF";
    }
}
return String();
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "bON") == 0) {
            ledState = 0;
            notifyClients("ON");
        }
        if (strcmp((char*)data, "bOFF") == 0) {
            ledState = 1;
            notifyClients("OFF");
        }
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

void setup() {
    // Serial port for debugging purposes
}

```

```

Serial.begin(115200);
pinMode(ledPin, OUTPUT);

initFS();
initWiFi();
initWebSocket();

server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request) {
    if (!request->authenticate(http_username, http_password))
        return request->requestAuthentication();
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});
server.on("/logged-out", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(LittleFS, "/logged-out.html", "text/html", false, processor);
});
server.on("/logout", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(401);
});

server.serveStatic("/", LittleFS, "/").setAuthentication(http_username, http_password);

// Start server
server.begin();
}
void loop() {
    // put your main code here, to run repeatedly:
    ws.cleanupClients();
    digitalWrite(ledPin, ledState);
}

```

Modify the code to include your network credentials, and it will work straight away.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";

```

How The Code Works

Let's look at the code and see the relevant parts to add username and password to your web page.

Setting Your Username and Password

In the following variables, set the username and password for your web server. By default, the username is **admin**, and the password is also **admin**. We definitely recommend that you change them.

```
// Web Server HTTP Authentication credentials
const char* http_username = "admin";
const char* http_password = "admin";
```

Handle Requests with Authentication

Every time you make a request to the ESP32 or ESP8266 to access the web server, it will check if you've already authenticated with the correct username and password.

Basically, to add authentication to your web server, you just need to add the following lines after each request:

```
if(!request->authenticate(http_username, http_password))
    return request->requestAuthentication();
```

These two lines continuously pop up the authentication window until you insert the right credentials defined in the variables `http_username` and `http_password`. You need to do this for all requests. This way, you ensure that you'll only get responses if you are logged in.

For example, when you try to access the root URL (ESP IP address), you add the previous two lines before sending the page. If you enter the wrong credentials, the browser will keep asking for them.

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
    if(!request->authenticate(http_username, http_password))
        return request->requestAuthentication();
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});
```

For the static files, you handle the authentication by using the `setAuthentication()` method as shown below:

```
server.serveStatic("/", SPIFFS, "/").setAuthentication(http_username, http_password);
```

Handle Logout Button

When you click the logout button, the ESP receives a request on the /logout URL.

When that happens send the response code 401.

```
server.on("/logout", HTTP_GET, [](AsyncWebRequest *request){  
    request->send(401);  
});
```

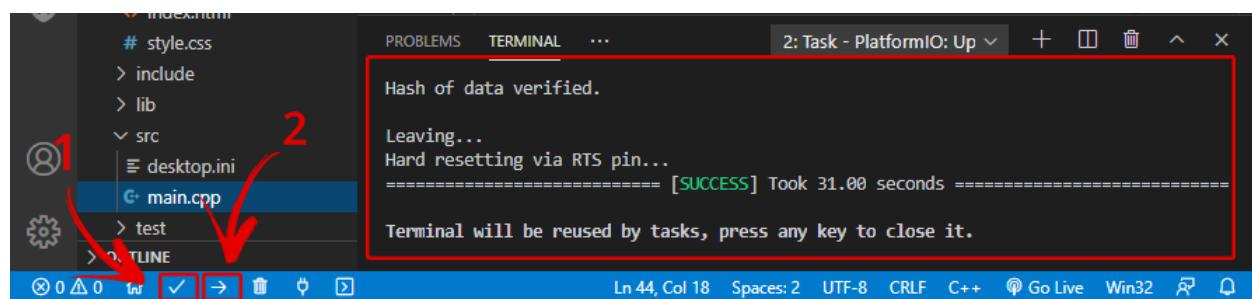
The response code 401 is an unauthorized error HTTP response status code indicating that the request sent by the client could not be authenticated. So, it will have the same effect as a logout – it will ask for the username and password and won't let you access the web server until you login again.

When you click the web server logout button, after one second, the ESP receives another request on the /logged-out URL. When that happens, send the HTML text to build the logout page (*logged-out.html* file).

```
server.on("/logged-out", HTTP_GET, [](AsyncWebRequest *request){  
    request->send(SPIFFS, "/logged-out.html", "text/html", false, processor);  
});
```

Uploading Code

After modifying the code with your network credentials, and adding your username and password to the web page, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.

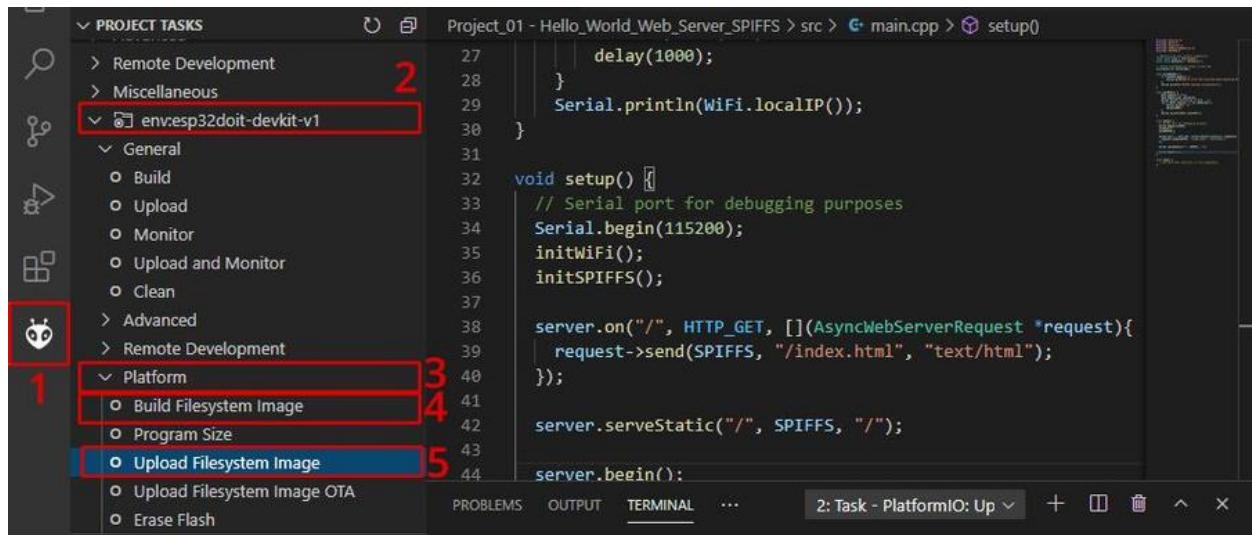


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, upload the files (*index.html*, *logged-out.html*, *style.css*, *script.js*, and *favicon.png*) to the filesystem:

1. Click on the PIO icon at the left bar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Click on **Build Filesystem Image**.
5. Finally, click on **Upload Filesystem Image**.



Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. If you're using the same board as the previous examples, it will probably have the same IP.

```

PROBLEMS TERMINAL ...
4: Task - PlatformIO: Mc + [-] [x] ^ x

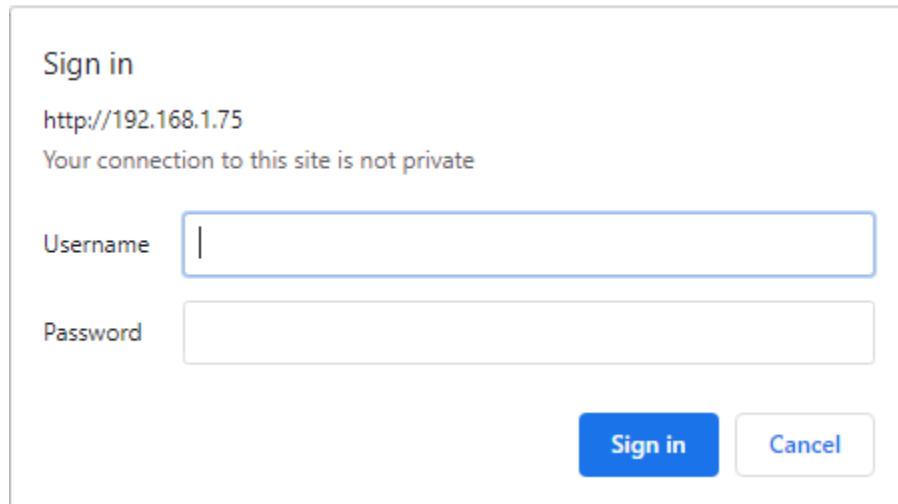
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5828
entry 0x400806ac
Connecting to WiFi ...192.168.1.112

```

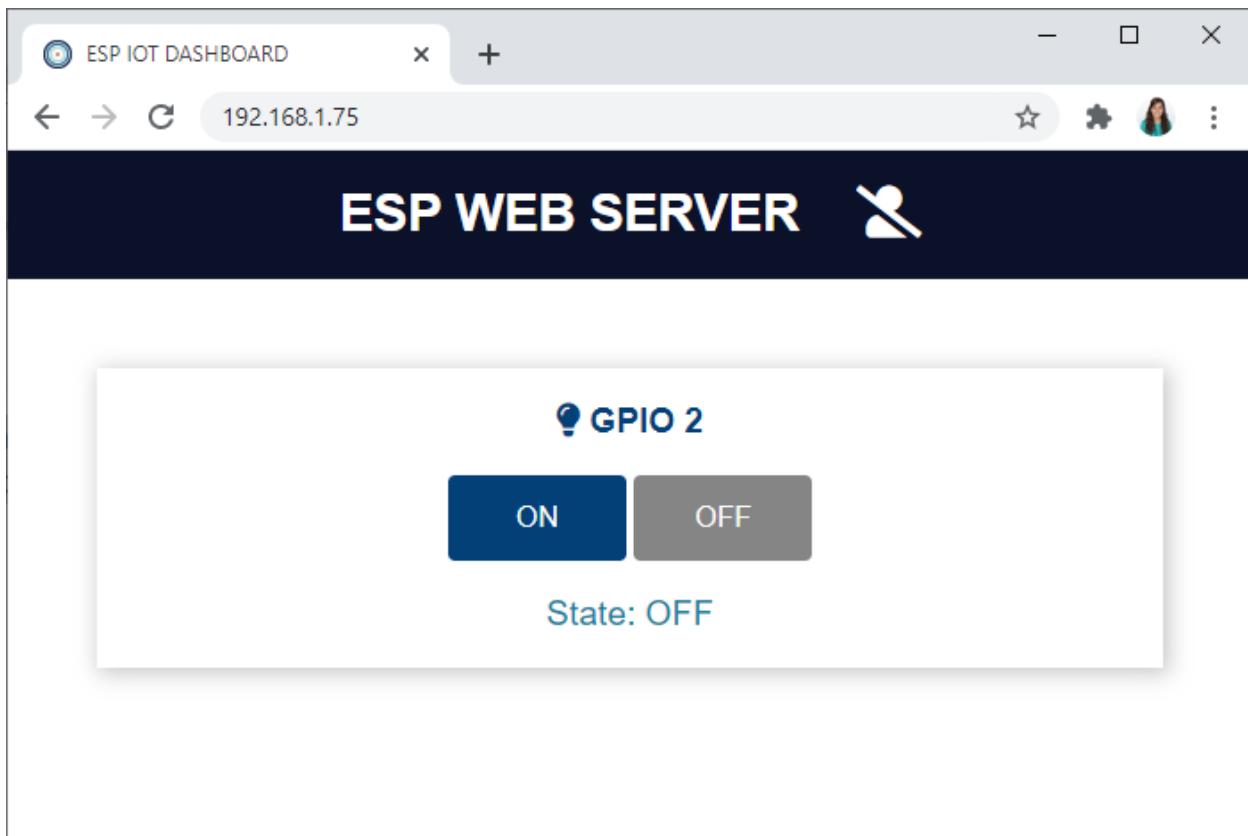
The screenshot shows the Microsoft Code Editor interface. On the left is the file explorer with files like .gitignore, .travis.yml, desktop.ini, and platformio.ini. The right side has a terminal window titled '4: Task - PlatformIO: Mc' showing serial port communication. A red arrow points to the terminal tab. The status bar at the bottom shows line 11, column 27, spaces: 2, and other settings.

Open a browser on your local network and insert the ESP IP address.

The following page should load asking for the username and password. Enter the username and password, and you should get access to the web server. If you didn't modify the code, the username is **admin**, and the password is **admin**.

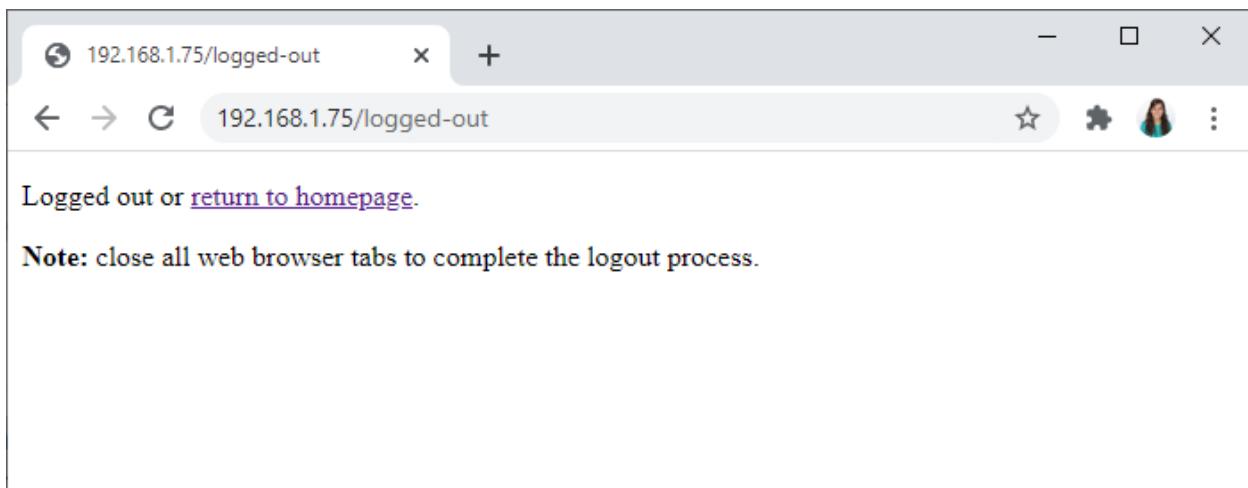


After typing the correct username and password, you should get access to the web server.



You can play with the web server and see that it controls the ESP32 or ESP8266 on-board LED.

In the web page, there's a logout button. If you click it, you'll be redirected to a logout page, as shown below.



If you click the “**return to homepage**” link, you’ll be redirected to the main page. If you’re using Google Chrome, you’ll need to enter the username and password to access the web server again.

Important: if you’re using other web browsers like Firefox or Safari, you need to close all web browser tabs to completely logout. Otherwise, if you go back to the main web page, you’ll still have access.

So, we advise that you close all web browser tabs after clicking the logout button.

You also need to enter the username and password if you try to access using a different device on the local network, even though you have access on another device.

Download Project Folder

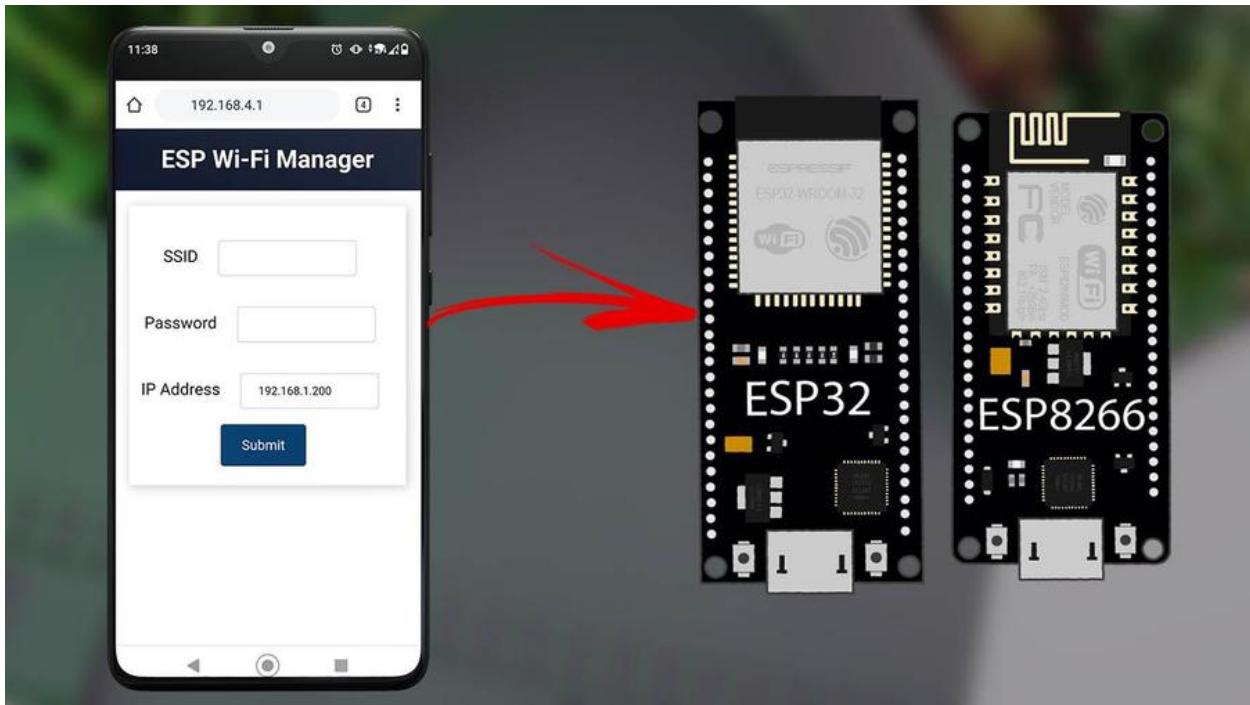
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you’ve learned how to add authentication to your ESP32 and ESP8266 web servers (password-protected web server). You can apply what you learned in this tutorial to any web server built with the ESPAsyncWebServer library.

4.3 – Wi-Fi Manager for Web Server



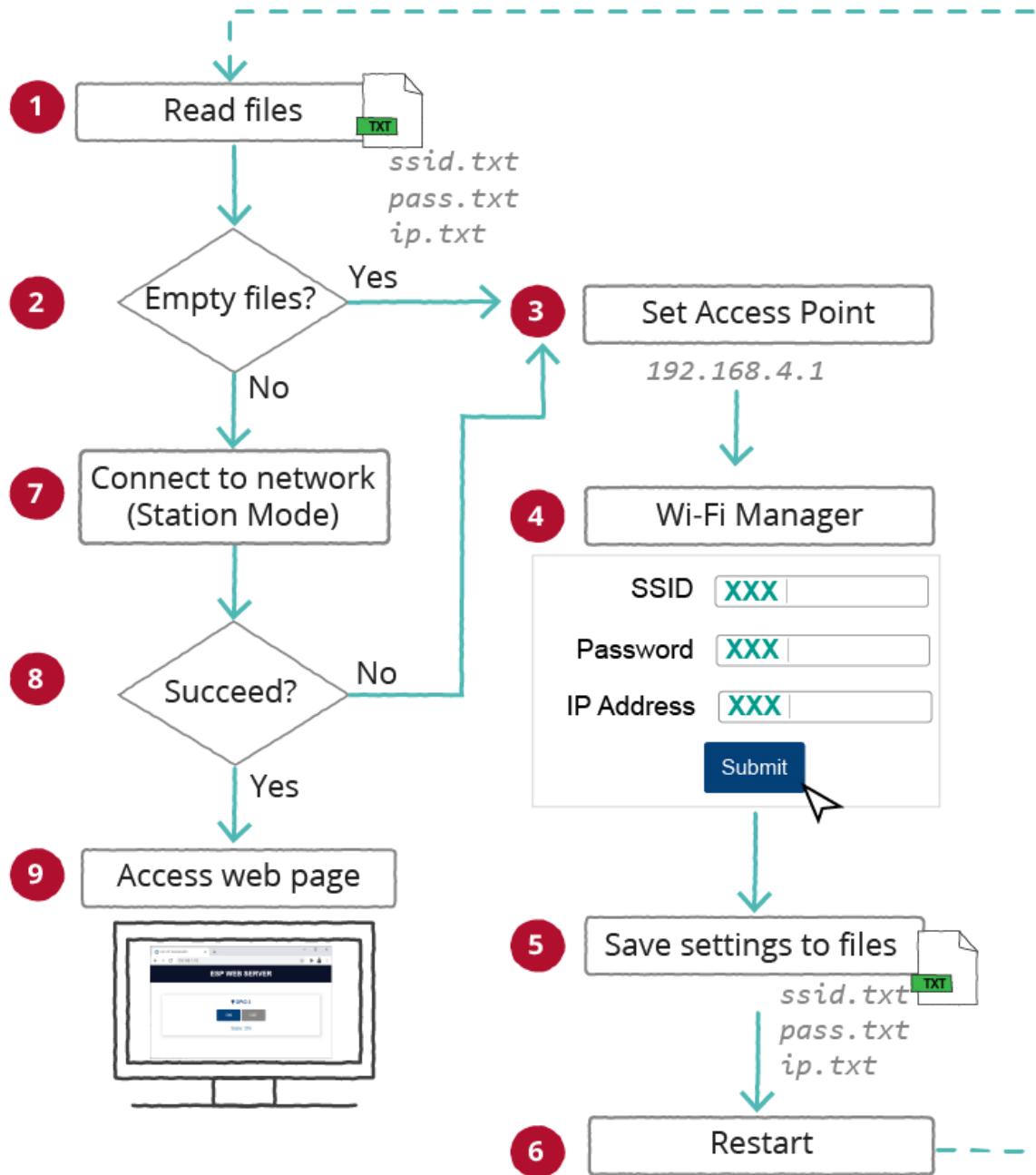
In this Unit, you'll create and set up a Wi-Fi Manager for your web server projects. The Wi-Fi Manager allows you to connect the ESP32 or ESP8266 boards to different Access Points (networks) without having to hard-code network credentials (SSID and password) and upload new code to your board. Your ESP will automatically join the last saved network or set up an Access Point that you can use to configure the network credentials.

To better understand how this project works, we recommend taking a look at the following units first:

- [4.1 - Web Server with Input Fields \(HTML Form\)](#)
- [Setting the ESP32 and ESP8266 as an Access Point](#)
- [ESP32/ESP8266 Static IP Address](#)

How It Works?

Here's how the Wi-Fi Manager works:



- When the ESP first starts, it tries to read the `ssid.txt`, `pass.txt` and `ip.txt` files (1);
- If the files are empty (2) (the first time you run the board, the files are empty), your board is set as an access point (3);

- Using any Wi-Fi enabled device with a browser, you can connect to the newly created Access Point (default name **ESP-WIFI-MANAGER**);
- After establishing a connection with the **ESP-WIFI-MANAGER**, you can go to the default IP address **192.168.4.1** to open a web page that allows you to configure your SSID and password (**4**);
- The SSID, password, and IP address inserted in the form are saved in the corresponding files: *ssid.txt*, *pass.txt*, and *ip.txt* (**5**);
- After that, the ESP board restarts (**6**);
- This time, after restarting, the files are not empty, so the ESP will try to connect to the network in station mode using the settings you've inserted in the form (**7**);
- If it establishes a connection, the process is completed successfully, and you can access the main web page that controls and monitors your board (**9**). Otherwise, it will set the Access Point (**3**), and you can access the default IP address (**192.168.4.1**) to add another SSID/password combination.

To show you how to set the Wi-Fi Manager, we'll use the project from Unit 2.1 as an example. You can apply the Wi-Fi Manager to any web server project.

Building the Web Page

For this project, make sure you place all these files inside the *data* folder under your project folder:

- *index.html*
- *wifimanager.html*
- *style.css*
- *favicon.png*

HTML File

For this project, you need two HTML files: one to build the main page that controls the outputs (*index.html*) and another to build the Wi-Fi Manager page (*wifimanager.html*).

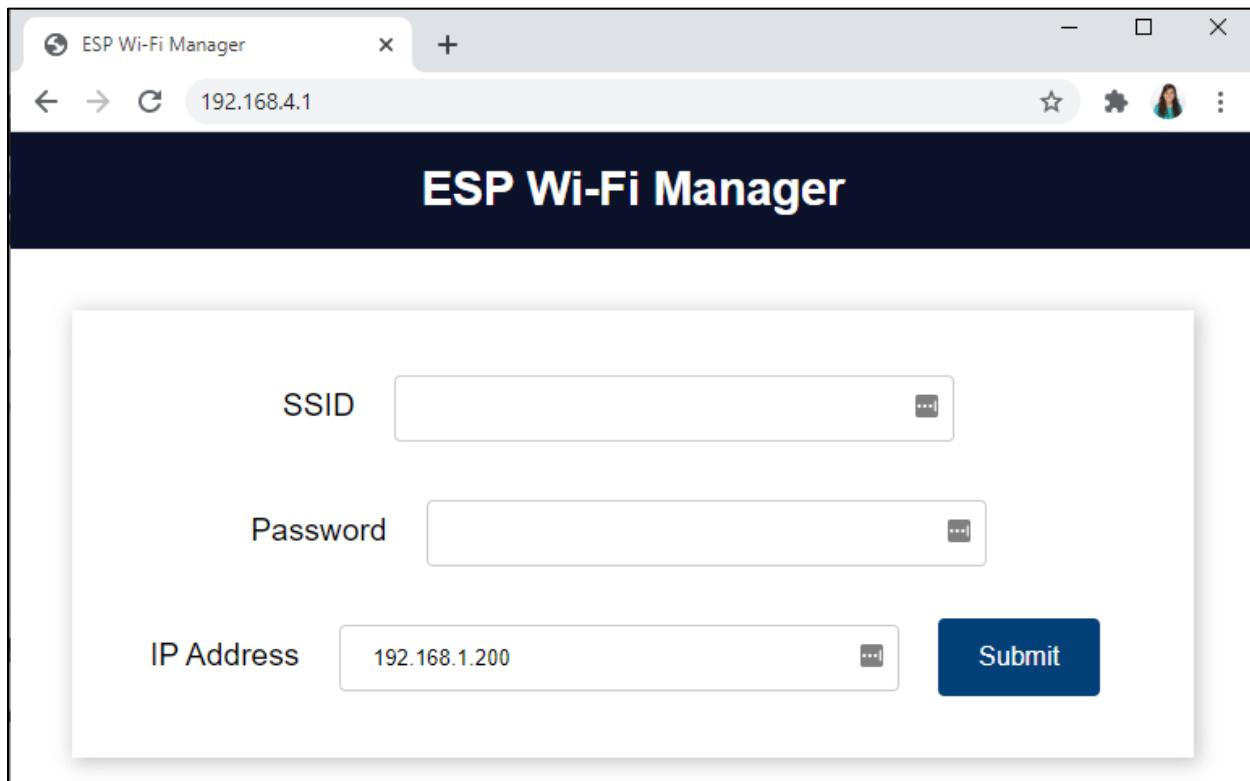
index.html

Here's the text you should copy to your *index.html* file. It is the same as in Unit 2.1.

```
<!DOCTYPE html>
<html>
  <head>
    <title>ESP WEB SERVER</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" href="style.css">
    <link rel="icon" type="image/png" href="favicon.png">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fNmOCqbTlWIj8LyTjo7mOUSTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
  </head>
  <body>
    <div class="topnav">
      <h1>ESP WEB SERVER</h1>
    </div>
    <div class="content">
      <div class="card-grid">
        <div class="card">
          <p class="card-title"><i class="fas fa-lightbulb"></i> GPIO 2</p>
          <p>
            <a href="on"><button class="button-on">ON</button></a>
            <a href="off"><button class="button-off">OFF</button></a>
          </p>
          <p class="state">State: %STATE%</p>
        </div>
      </div>
    </div>
  </body>
</html>
```

wifimanager.html

The Wi-Fi Manager web page looks like this:



Copy the following to the *wifimanager.html* file. This creates a web page with a form with three input fields and a **Submit** button.

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP Wi-Fi Manager</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" href="data:,">
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.2/css/all.css" integrity="sha384-fnmOCqbTlWIlj8LyTjo7m0USTjsKC4p0pQbqyi7RrhN7udi9RwhKkMHpvLbHG9Sr" crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
    <div class="topnav">
        <h1>ESP Wi-Fi Manager</h1>
    </div>
    <div class="content">
        <div class="card-grid">
            <div class="card">
```

```
<form action="/" method="POST">
  <p>
    <label for="ssid">SSID</label>
    <input type="text" id ="ssid" name="ssid"><br>
    <label for="pass">Password</label>
    <input type="text" id ="pass" name="pass"><br>
    <label for="ip">IP Address</label>
    <input type="text" id ="ip" name="ip" value="192.168.1.200">
    <input type = "submit" value = "Submit">
  </p>
</form>
</div>
</div>
</div>
</body>
</html>
```

This is the input field for the SSID:

```
<label for="ssid">SSID</label>
<input type="text" id ="ssid" name="ssid"><br>
```

The input field for the password:

```
<label for="pass">Password</label>
<input type="text" id ="pass" name="pass"><br>
```

And finally, there is an input field for the IP address that you want to attribute to the ESP in station mode. As default, we set it to 192.168.1.200 (you can set another default IP address or you can delete the `value` parameter—it won't have a default value).

```
<label for="ip">IP Address</label>
<input type="text" id ="ip" name="ip" value="192.168.1.200">
```

CSS File

Copy the following styles to your `style.css` file. The styles are the same used throughout the eBook.

```
html {
  font-family: Arial, Helvetica, sans-serif;
  display: inline-block;
  text-align: center;
}
```

```
h1 {
  font-size: 1.8rem;
  color: white;
}

p {
  font-size: 1.4rem;
}

.topnav {
  overflow: hidden;
  background-color: #0A1128;
}

body {
  margin: 0;
}

.content {
  padding: 5%;
}

.card-grid {
  max-width: 800px;
  margin: 0 auto;
  display: grid;
  grid-gap: 2rem;
  grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
}

.card {
  background-color: white;
  box-shadow: 2px 2px 12px 1px rgba(140,140,140,.5);
}

.card-title {
  font-size: 1.2rem;
  font-weight: bold;
  color: #034078
}

input[type=submit] {
  border: none;
  color: #FEFCFB;
  background-color: #034078;
  padding: 15px 15px;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  width: 100px;
  margin-right: 10px;
  border-radius: 4px;
```

```
    transition-duration: 0.4s;
}

input[type=submit]:hover {
    background-color: #1282A2;
}

input[type=text], input[type=number], select {
    width: 50%;
    padding: 12px 20px;
    margin: 18px;
    display: inline-block;
    border: 1px solid #ccc;
    border-radius: 4px;
    box-sizing: border-box;
}

label {
    font-size: 1.2rem;
}
.value{
    font-size: 1.2rem;
    color: #1282A2;
}
.state {
    font-size: 1.2rem;
    color: #1282A2;
}
button {
    border: none;
    color: #FEFCFB;
    padding: 15px 32px;
    text-align: center;
    font-size: 16px;
    width: 100px;
    border-radius: 4px;
    transition-duration: 0.4s;
}
.button-on {
    background-color: #034078;
}
.button-on:hover {
    background-color: #1282A2;
}
.button-off {
    background-color: #858585;
}
.button-off:hover {
    background-color: #252524;
}
```

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed=115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

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

```

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

// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "ssid";
const char* PARAM_INPUT_2 = "pass";
const char* PARAM_INPUT_3 = "ip";

// Variables to save values from HTML form
String ssid;
String pass;
String ip;

// File paths to save input values permanently
const char* ssidPath = "/ssid.txt";
const char* passPath = "/pass.txt";
const char* ipPath = "/ip.txt";

IPAddress localIP;
// IPAddress localIP(192, 168, 1, 200); // hardcoded

// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 0, 0);

// Timer variables
unsigned long previousMillis = 0;
const long interval = 10000; // interval to wait for Wi-Fi connection (milliseconds)

// Set LED GPIO
const int ledPin = 2;
// Stores LED state

String ledState;

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Read File from SPIFFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path);
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }
}

```

```

String fileContent;
while(file.available()){
    fileContent = file.readStringUntil('\n');
    break;
}
return fileContent;
}

// Write file to SPIFFS
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, FILE_WRITE);
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
}

// Initialize WiFi
bool initWiFi() {
    if(ssid=="") || ip==""){
        Serial.println("Undefined SSID or IP address.");
        return false;
    }

    WiFi.mode(WIFI_STA);
    localIP.fromString(ip.c_str());

    if (!WiFi.config(localIP, gateway, subnet)){
        Serial.println("STA Failed to configure");
        return false;
    }
    WiFi.begin(ssid.c_str(), pass.c_str());
    Serial.println("Connecting to WiFi...");

    unsigned long currentMillis = millis();
    previousMillis = currentMillis;

    while(WiFi.status() != WL_CONNECTED) {
        currentMillis = millis();
        if (currentMillis - previousMillis >= interval) {
            Serial.println("Failed to connect.");
            return false;
        }
    }
}

Serial.println(WiFi.localIP());

```

```

        return true;
    }

    // Replaces placeholder with LED state value
String processor(const String& var) {
    if(var == "STATE") {
        if(digitalRead(ledPin)) {
            ledState = "ON";
        }
        else {
            ledState = "OFF";
        }
        return ledState;
    }
    return String();
}

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

    initSPIFFS();

    // Set GPIO 2 as an OUTPUT
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);

    // Load values saved in SPIFFS
    ssid = readFile(SPIFFS, ssidPath);
    pass = readFile(SPIFFS, passPath);
    ip = readFile(SPIFFS, ipPath);
    Serial.println(ssid);
    Serial.println(pass);
    Serial.println(ip);

    if(initWiFi()) {
        // Route for root / web page
        server.on("/", HTTP_GET, [] (AsyncWebRequest *request) {
            request->send(SPIFFS, "/index.html", "text/html", false, processor);
        });
        server.serveStatic("/", SPIFFS, "/");
    }

    // Route to set GPIO state to HIGH
    server.on("/on", HTTP_GET, [] (AsyncWebRequest *request) {
        digitalWrite(ledPin, HIGH);
        request->send(SPIFFS, "/index.html", "text/html", false, processor);
    });

    // Route to set GPIO state to LOW
    server.on("/off", HTTP_GET, [] (AsyncWebRequest *request) {
        digitalWrite(ledPin, LOW);
        request->send(SPIFFS, "/index.html", "text/html", false, processor);
    });
    server.begin();
}

```

```

}

else {
    // Connect to Wi-Fi network with SSID and password
    Serial.println("Setting AP (Access Point)");
    // Remove the password parameter, if you want the AP (Access Point) to be open
    WiFi.softAP("ESP-WIFI-MANAGER", NULL);

    IPAddress IP = WiFi.softAPIP();
    Serial.print("AP IP address: ");
    Serial.println(IP);

    // Web Server Root URL
    server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
        request->send(SPIFFS, "/wifimanager.html", "text/html");
    });

    server.serveStatic("/", SPIFFS, "/");
}

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    for(int i=0;i<params;i++){
        AsyncWebParameter* p = request->getParam(i);
        if(p->isPost()){

            // HTTP POST ssid value
            if (p->name() == PARAM_INPUT_1) {
                ssid = p->value().c_str();
                Serial.print("SSID set to: ");
                Serial.println(ssid);
                // Write file to save value
                writeFile(SPIFFS, ssidPath, ssid.c_str());
            }

            // HTTP POST pass value
            if (p->name() == PARAM_INPUT_2) {
                pass = p->value().c_str();
                Serial.print("Password set to: ");
                Serial.println(pass);
                // Write file to save value
                writeFile(SPIFFS, passPath, pass.c_str());
            }

            // HTTP POST ip value
            if (p->name() == PARAM_INPUT_3) {
                ip = p->value().c_str();
                Serial.print("IP Address set to: ");
                Serial.println(ip);
                // Write file to save value
                writeFile(SPIFFS, ipPath, ip.c_str());
            }

            //Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value().c_str());
        }
    }

    request->send(200, "text/plain", "Done. ESP will restart,
        connect to your router and go to IP address: " + ip);
    delay(3000);
    ESP.restart();
}

```

```
    });
    server.begin();
}
}

void loop() {
}
```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"

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

// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "ssid";
const char* PARAM_INPUT_2 = "pass";
const char* PARAM_INPUT_3 = "ip";

// Variables to save values from HTML form
String ssid;
String pass;
String ip;

// File paths to save input values permanently
const char* ssidPath = "/ssid.txt";
const char* passPath = "/pass.txt";
const char* ipPath = "/ip.txt";

IPAddress localIP;
// IPAddress localIP(192, 168, 1, 200); // hardcoded

// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 0, 0);

// Timer variables
unsigned long previousMillis = 0;
```

```

const long interval = 10000; // interval to wait for Wi-
Fi connection (milliseconds)

// Set LED GPIO
const int ledPin = 2;
// Stores LED state
String ledState;

boolean restart = false;

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Read File from LittleFS
String readFile(fs::FS &fs, const char * path){
    Serial.printf("Reading file: %s\r\n", path);

    File file = fs.open(path, "r");
    if(!file || file.isDirectory()){
        Serial.println("- failed to open file for reading");
        return String();
    }

    String fileContent;
    while(file.available()){
        fileContent = file.readStringUntil('\n');
        break;
    }
    file.close();
    return fileContent;
}

// Write file to LittleFS
void writeFile(fs::FS &fs, const char * path, const char * message){
    Serial.printf("Writing file: %s\r\n", path);

    File file = fs.open(path, "w");
    if(!file){
        Serial.println("- failed to open file for writing");
        return;
    }
    if(file.print(message)){
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}

```

```

// Initialize WiFi
bool initWiFi() {
    if(ssid=="" || ip==""){
        Serial.println("Undefined SSID or IP address.");
        return false;
    }

    WiFi.mode(WIFI_STA);
    localIP.fromString(ip.c_str());

    if (!WiFi.config(localIP, gateway, subnet)){
        Serial.println("STA Failed to configure");
        return false;
    }
    WiFi.begin(ssid.c_str(), pass.c_str());

    Serial.println("Connecting to WiFi...");
    delay(20000);
    if(WiFi.status() != WL_CONNECTED) {
        Serial.println("Failed to connect.");
        return false;
    }
    Serial.println(WiFi.localIP());
    return true;
}

// Replaces placeholder with LED state value
String processor(const String& var) {
    if(var == "STATE") {
        if(!digitalRead(ledPin)) {
            ledState = "ON";
        }
        else {
            ledState = "OFF";
        }
        return ledState;
    }
    return String();
}

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

    initFS();

    // Set GPIO 2 as an OUTPUT
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, HIGH);

    // Load values saved in LittleFS
    ssid = readFile(LittleFS, ssidPath);
    pass = readFile(LittleFS, passPath);
    ip = readFile(LittleFS, ipPath);
}

```

```

Serial.println(ssid);
Serial.println(pass);
Serial.println(ip);

if(initWiFi()) {
    // Route for root / web page
    server.on("/", HTTP_GET, [](AsyncWebServerRequest *request) {
        request->send(LittleFS, "/index.html", "text/html", false, processor);
    });
    server.serveStatic("/", LittleFS, "/");
}

// Route to set GPIO state to HIGH
server.on("/on", HTTP_GET, [](AsyncWebServerRequest *request) {
    digitalWrite(ledPin, LOW);
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});

// Route to set GPIO state to LOW
server.on("/off", HTTP_GET, [](AsyncWebServerRequest *request) {
    digitalWrite(ledPin, HIGH);
    request->send(LittleFS, "/index.html", "text/html", false, processor);
});
server.begin();
}
else {
    // Connect to Wi-Fi network with SSID and password
    Serial.println("Setting AP (Access Point)");
    // Remove the password parameter, if you want the AP (Access Point) to be open
    WiFi.softAP("ESP-WIFI-MANAGER", NULL);

    IPAddress IP = WiFi.softAPIP();
    Serial.print("AP IP address: ");
    Serial.println(IP);

    // Web Server Root URL
    server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
        request->send(LittleFS, "/wifimanager.html", "text/html");
    });

    server.serveStatic("/", LittleFS, "/");
}

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    for(int i=0;i<params;i++){
        AsyncWebParameter* p = request->getParam(i);
        if(p->isPost()){
            // HTTP POST ssid value
            if (p->name() == PARAM_INPUT_1) {
                ssid = p->value().c_str();
                Serial.print("SSID set to: ");
                Serial.println(ssid);
                // Write file to save value
                writeFile(LittleFS, ssidPath, ssid.c_str());
            }
        }
    }
}

```

```

// HTTP POST pass value
if (p->name() == PARAM_INPUT_2) {
    pass = p->value().c_str();
    Serial.print("Password set to: ");
    Serial.println(pass);
    // Write file to save value
    writeFile(LittleFS, passPath, pass.c_str());
}
// HTTP POST ip value
if (p->name() == PARAM_INPUT_3) {
    ip = p->value().c_str();
    Serial.print("IP Address set to: ");
    Serial.println(ip);
    // Write file to save value
    writeFile(LittleFS, ipPath, ip.c_str());
}
//Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value().c_str());
}
restart = true;
request->send(200, "text/plain", "Done. ESP will restart,
connect to your router and go to IP address: " + ip);
});
server.begin();
}
}

void loop() {
    if(restart){
        delay(5000);
        ESP.restart();
    }
}

```

How The Code Works

Let's take a look at the code and see how the Wi-Fi Manager works.

The following variables are used to search for the SSID, password, and IP address on the HTTP POST request made when the form is submitted.

```

// Search for parameter in HTTP POST request
const char* PARAM_INPUT_1 = "ssid";
const char* PARAM_INPUT_2 = "pass";
const char* PARAM_INPUT_3 = "ip";

```

The `ssid`, `pass` and `ip` variables save the values of the SSID, password, and IP address submitted in the form.

```
//Variables to save values from HTML form
String ssid;
String pass;
String ip;
```

The SSID, password, and IP address, when submitted, are saved in files in the ESP filesystem. The following variables refer to the path of those files.

```
// File paths to save input values permanently
const char* ssidPath = "/ssid.txt";
const char* passPath = "/pass.txt";
const char* ipPath = "/ip.txt";
```

The station IP address is submitted in the Wi-Fi Manager form. However, you need to set the gateway and subnet in your code:

```
IPAddress localIP;
//IPAddress localIP(192, 168, 1, 200); // hardcoded

// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 0, 0);
```

initWiFi()

The `initWiFi()` function returns a boolean value (either `true` or `false`) indicating if the ESP board connected successfully to a network.

First, it checks if the `ssid` and `ip` variables are empty. If they are, it won't be able to connect to a network, so it returns `false`.

```
if(ssid=="" || ip==""){
    Serial.println("Undefined SSID or IP address.");
    return false;
}
```

If that's not the case, we'll try to connect to the network using the SSID and password saved in the `ssid` and `pass` variables and set the IP address.

```
WiFi.mode(WIFI_STA);
localIP.fromString(ip.c_str());

if (!WiFi.config(localIP, gateway, subnet)){
    Serial.println("STA Failed to configure");
```

```
    return false;
}
WiFi.begin(ssid.c_str(), pass.c_str());
Serial.println("Connecting to WiFi...");
```

If after 10 seconds (`interval` variable), it cannot connect to Wi-Fi, it will return `false`.

```
unsigned long currentMillis = millis();
previousMillis = currentMillis;

while(WiFi.status() != WL_CONNECTED) {
    currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        Serial.println("Failed to connect.");
        return false;
    }
}
```

If none of the previous conditions are met, it means that the ESP successfully connected to the network in station mode (returns `true`).

```
return true;
```

setup()

In the `setup()`, start reading the files to get the previously saved SSID, password and IP address.

```
ssid = readFile(SPIFFS, ssidPath);
pass = readFile(SPIFFS, passPath);
ip = readFile(SPIFFS, ipPath);
```

If the ESP connects successfully in station mode (`initWiFi()` function returns `true`), we can set the commands to handle the web server requests:

```
if(initWiFi()) {
    // Route for root / web page
    server.on("/", HTTP_GET, [](){AsyncWebRequest *request) {
        request->send(SPIFFS, "/index.html", "text/html", false, processor);
    });
    server.serveStatic("/", SPIFFS, "/");

    // Route to set GPIO state to HIGH
    server.on("/on", HTTP_GET, [](){AsyncWebRequest *request) {
        digitalWrite(ledPin, HIGH);
        request->send(SPIFFS, "/index.html", "text/html", false, processor);
    });
}
```

```

});  
  

// Route to set GPIO state to LOW  

server.on("/off", HTTP_GET, [](AsyncWebRequest *request) {  

    digitalWrite(ledPin, LOW);  

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

});  

server.begin();  

}

```

If that's not the case, the `initWiFi()` function returns `false`. The ESP will set an access point:

```

// Connect to Wi-Fi network with SSID and password  

Serial.println("Setting AP (Access Point)");  

// NULL sets an open Access Point  

WiFi.softAP("ESP-WIFI-MANAGER", NULL);  
  

IPAddress IP = WiFi.softAPIP();  

Serial.print("AP IP address: ");  

Serial.println(IP);

```

To set an access point, we use the `softAP()` method and pass as arguments the name for the access point and the password. We want the access point to be open, so we set the password to `NULL`. You can add a password if you wish. To learn more about setting up an Access Point, go to this section:

- [Setting the ESP32 and ESP8266 as an Access Point](#)

When you access the Access Point, it shows the web page to enter the network credentials in the form. So, the ESP must send the `wifimanager.html` file when it receives a request on the root / URL.

```

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

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

});

```

We must also handle what happens when the form is submitted via an HTTP POST request.

The following lines save the submitted values in the `ssid`, `pass` and `ip` variables and save those variables in the corresponding files.

```

server.on("/", HTTP_POST, [](AsyncWebServerRequest *request) {
    int params = request->params();
    for(int i=0;i<params;i++){
        AsyncWebParameter* p = request->getParam(i);
        if(p->isPost()){
            // HTTP POST ssid value
            if (p->name() == PARAM_INPUT_1) {
                ssid = p->value().c_str();
                Serial.print("SSID set to: ");
                Serial.println(ssid);
                // Write file to save value
                writeFile(SPIFFS, ssidPath, ssid.c_str());
            }
            // HTTP POST pass value
            if (p->name() == PARAM_INPUT_2) {
                pass = p->value().c_str();
                Serial.print("Password set to: ");
                Serial.println(pass);
                // Write file to save value
                writeFile(SPIFFS, passPath, pass.c_str());
            }
            // HTTP POST ip value
            if (p->name() == PARAM_INPUT_3) {
                ip = p->value().c_str();
                Serial.print("IP Address set to: ");
                Serial.println(ip);
                // Write file to save value
                writeFile(SPIFFS, ipPath, ip.c_str());
            }
            //Serial.printf("POST[%s]: %s\n", p->name().c_str(), p->value().c_str());
        }
    }
}
}

```

For a more in-depth explanation about getting the values submitted in an HTTP POST request, we recommend getting back to unit [4.1 - Web Server with Input Fields \(HTML Form\)](#).

After submitting the form, send a response with some text so that we know that the ESP received the form details:

```

request->send (200, "text/plain", "Done. ESP will restart, connect to your
router and go to IP address: " + ip);

```

After three seconds, restart the ESP board with `ESP.restart()`:

```

delay(3000);
ESP.restart();

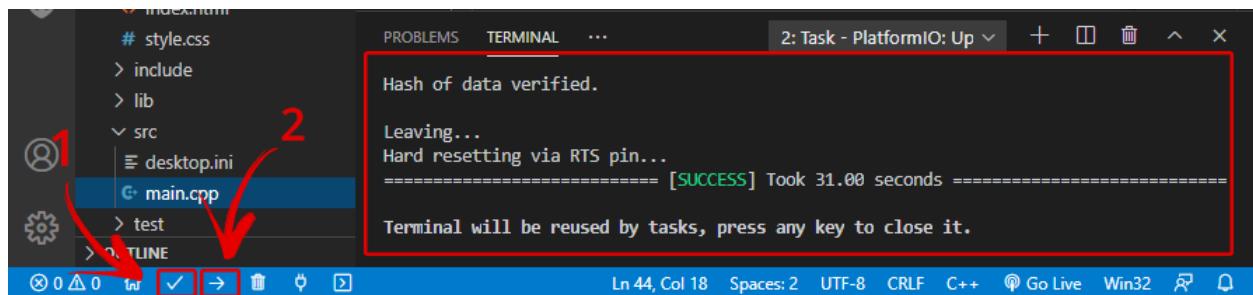
```

That's how the code works.

You can apply this idea to any of the other web server projects presented in the eBook or any of your projects.

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.

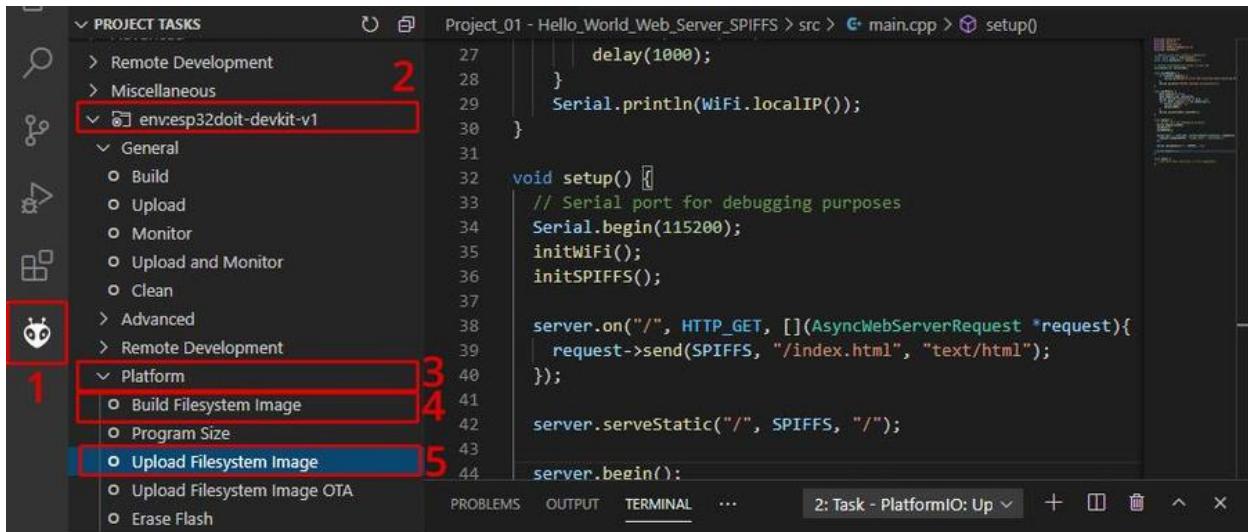


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, upload the files (*index.html*, *wifimanager.html*, *style.css*, and *favicon.png*) to the filesystem:

1. Click on the PIO icon at the left bar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Click on **Build Filesystem Image**.
5. Finally, click on **Upload Filesystem Image**.



Demonstration

After successfully uploading all files, you can open the Serial Monitor. If it is running the code for the first time, it will try to read the *ssid.txt*, *pass.txt*, and *ip.txt* files and it won't succeed because those files weren't created yet. So, it will start an Access Point.

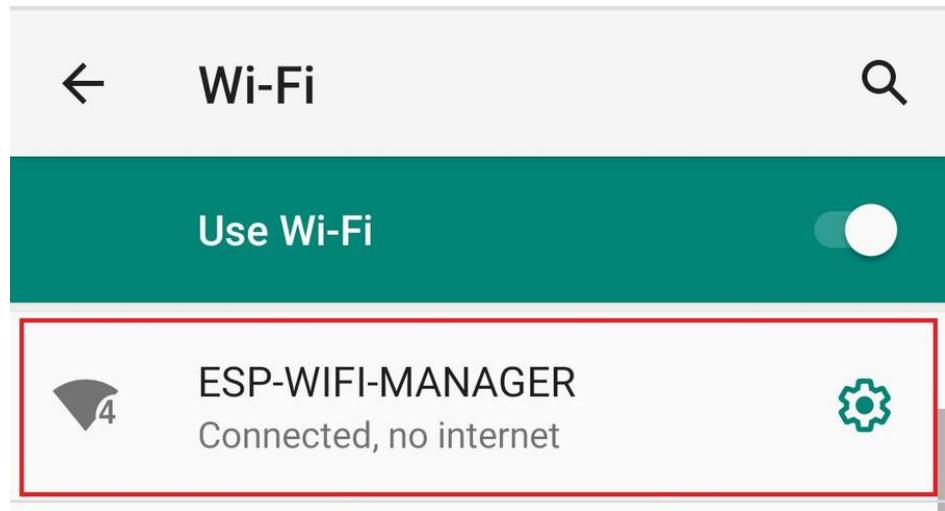
```

Reading file: /ssid.txt
- failed to open file for reading
Reading file: /pass.txt
- failed to open file for reading
Reading file: /ip.txt
- failed to open file for reading

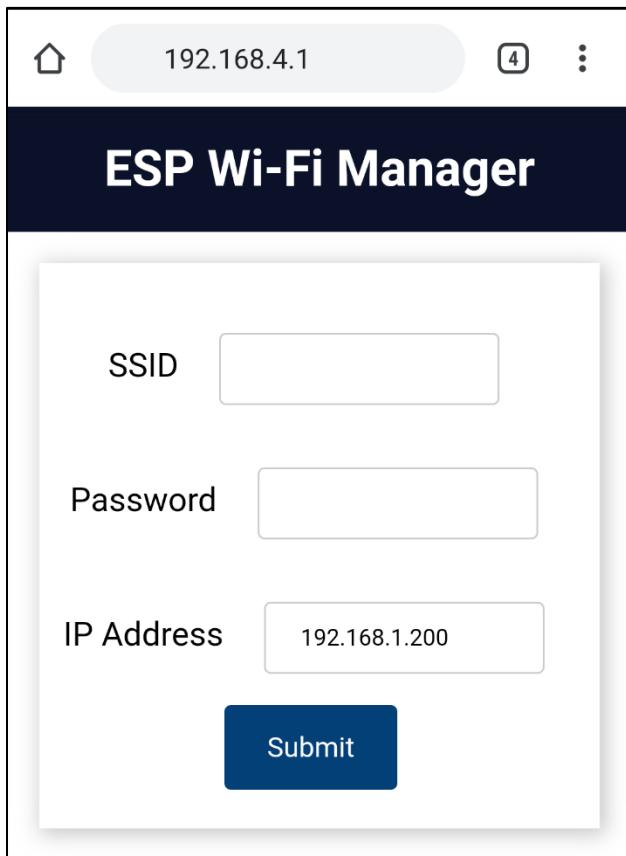
Undefined SSID or IP address.
Setting AP (Access Point)
AP IP address: 192.168.4.1

```

On your computer or smartphone, go to your network settings and connect to the **ESP-WIFI-MANAGER** access point.

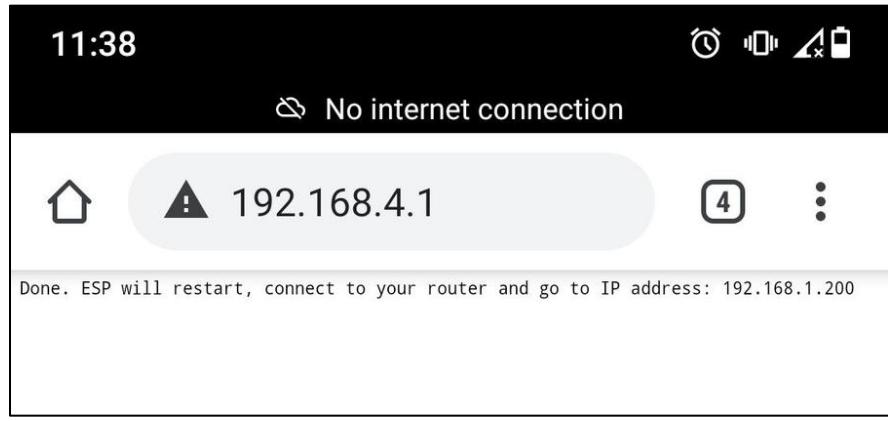


Then, open your browser and go to `192.168.4.1`. The Wi-Fi Manager web page should open.



Enter your network credentials: SSID and Password and an available IP address on your local network.

After that, you'll be redirected to the following page:



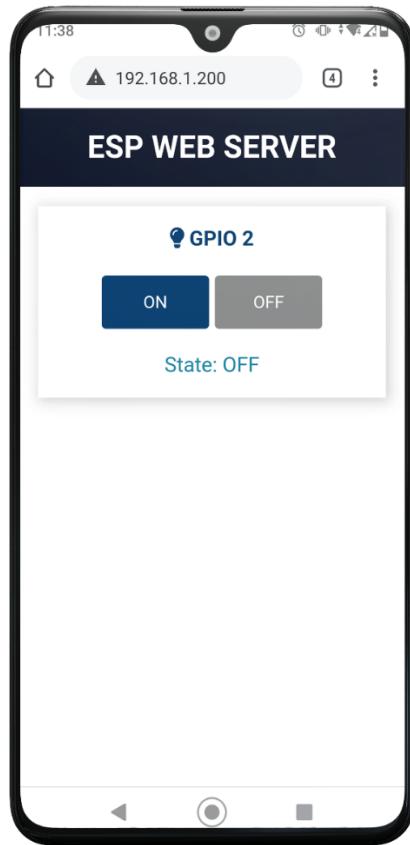
At the same time, the ESP should print the following in the Serial Monitor indicating that the parameters you've inserted were successfully saved in the corresponding files:

```
SSID set to: [REDACTED]
Writing file: /ssid.txt
- file written
Password set to: [REDACTED]
Writing file: /pass.txt
- file written
IP Address set to: 192.168.1.200
Writing file: /ip.txt
- file written
```

After a few seconds, the ESP will restart. And if you've inserted the right SSID and password, it will start in station mode:

```
SPIFFS mounted successfully
Reading file: /ssid.txt
Reading file: /pass.txt
Reading file: /ip.txt
[REDACTED]
[REDACTED]
192.168.1.200
Connecting to WiFi...
192.168.1.200
```

This time, open a browser on your local network and insert the ESP IP address. You should get access to the web page to control the outputs:



Download Project Folder

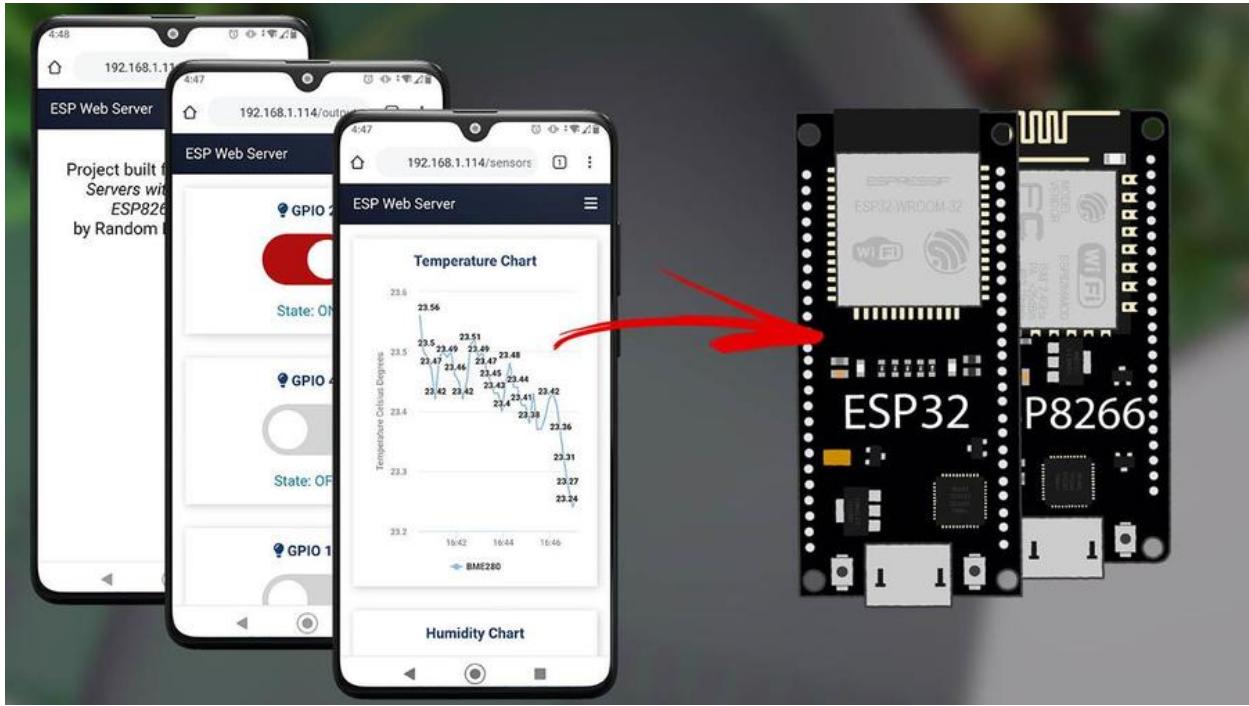
You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to set up a Wi-Fi Manager for your web server projects. With the Wi-Fi Manager, you can easily connect your ESP web servers to different networks without hard-code network credentials. You can apply the Wi-Fi Manager to any web server project.

4.4 – Multiple Web Pages (with Navigation Bar)



In this Unit, you'll learn how to serve multiple web pages and add a navigation bar to navigate between pages. For example, you may want to have sensor readings, output controls, and user input on three different web pages. To do this, you need to have a separate HTML file for each web page. It is also useful to have a navigation bar with hyperlinks to each page so that you can easily navigate between pages.

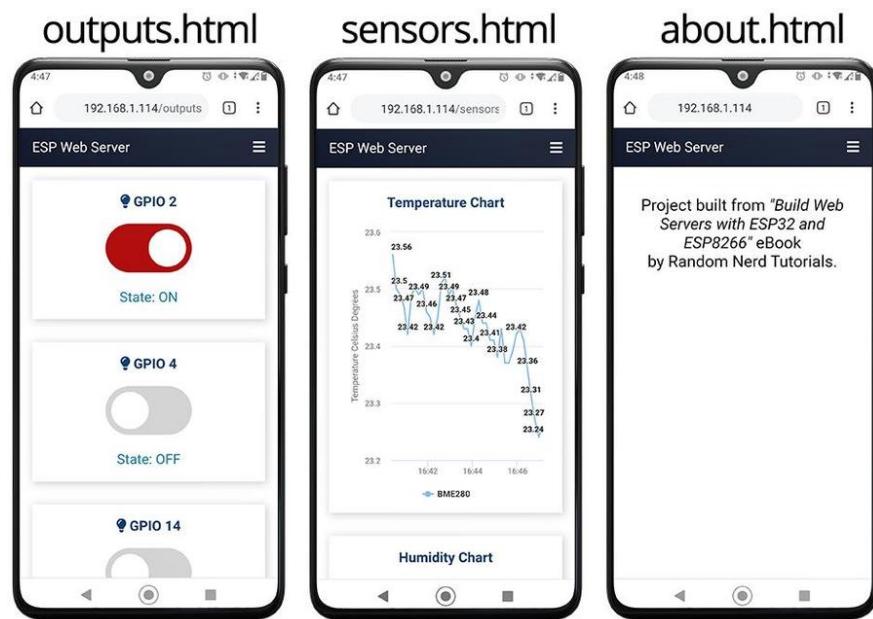
Project Overview

As an example, we'll build the following web pages:

- 1) A welcome/about web page that shows up when you access the root (/) URL;
- 2) A web page that displays sensor readings from the BME280 sensor (project from Unit 3.3);

- 3) A web page that displays toggle switches to control the ESP outputs (project from Unit 2.5) .

You can add any other web pages that might make sense for your project. Each page shows a navigation bar, so that you can easily navigate between pages. The navigation bar is mobile-responsive.

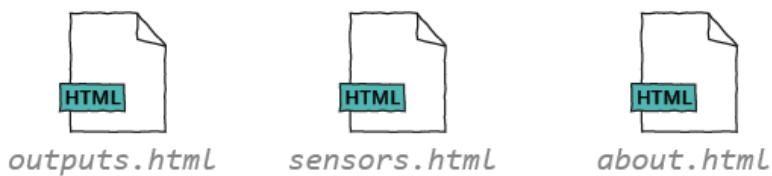


Serve Multiple Web Pages

There are several aspects you need to take into account when serving multiple web pages with ESP.

HTML Files

To serve multiple web pages, you need a different HTML file for each page. For this project, we'll serve the following pages:

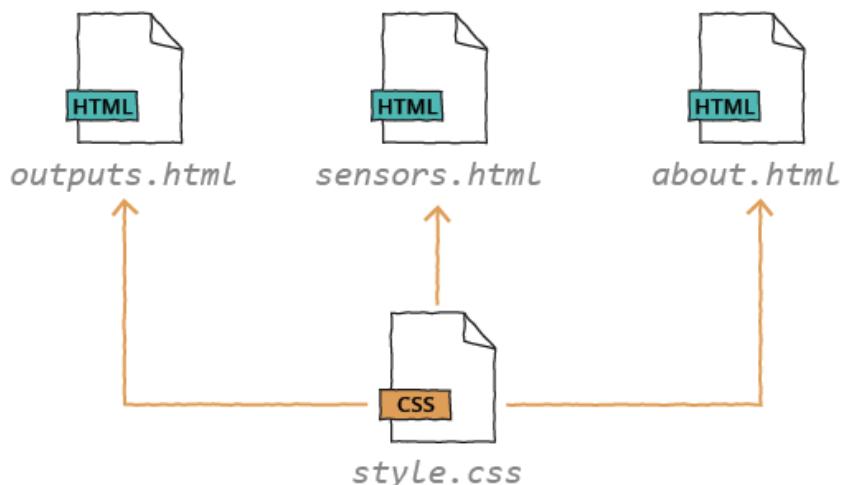


- *outputs.html*: serves a web page to control outputs—exactly like the page from Unit 2.5;
- *sensors.html*: serves a web page that shows sensor readings—like the project from Unit 3.3;
- *about.html*: the web page that shows up when you access the root (/) URL. This web page simply shows a paragraph about the project. The idea is to customize this web page so that it works as a “welcome” personalized page;

Each page should have the top navigation bar so that you can easily navigate between pages.

CSS

You can have a different CSS file for each of your web pages, or use the same stylesheet for all the pages. Usually, we want the same style in all web pages, so we just need one CSS file: *style.css*.

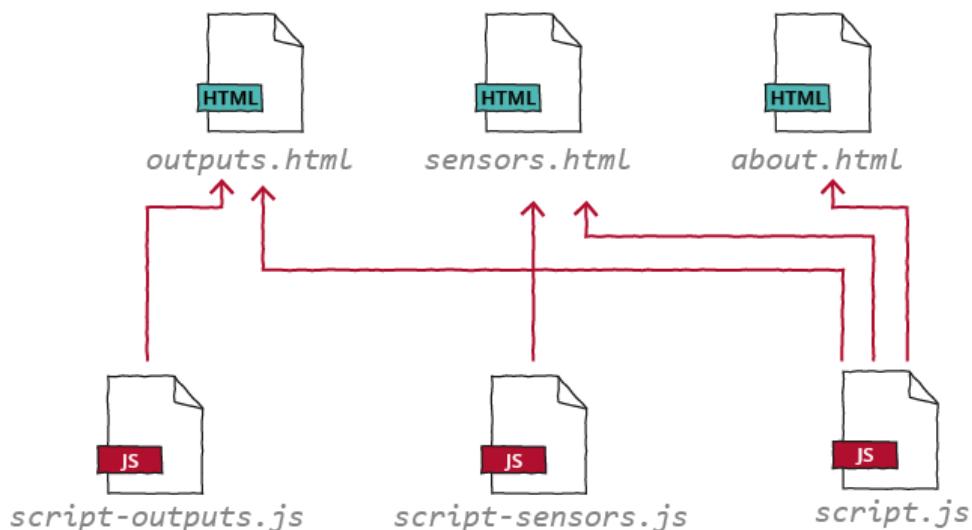


JavaScript Files

Each web page has different functionalities, so it is better to have one JavaScript file for each web page. We'll use:

- *script-outputs.js*: for the outputs page;
- *script-sensors.js*: for the sensors page;

If multiple pages share JavaScript functions, it can be more practical if all pages share the same JavaScript file. In our particular example, all pages share a function to handle the navigation bar. That function will be shared in a file called *script.js*.

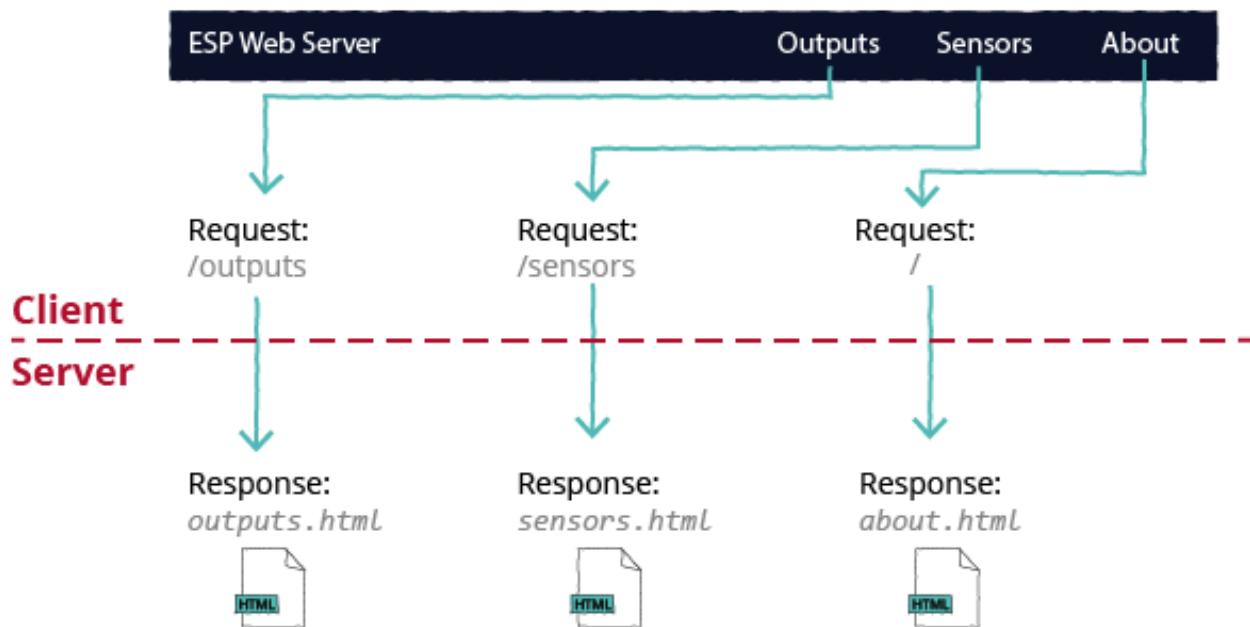


In summary, here's a list of the files needed for each page:

- **Sensor readings page:** *sensors.html, style.css, script-sensors.js, script.js*;
- **Outputs page:** *outputs.html, style.css, script-outputs.js, script.js*;
- **About page:** *about.html, style.css, script.js*;

Navigation Bar

Here's a brief explanation of how the navigation bar works.



- The navigation bar consists of a list of hyperlinks;
- Each hyperlink makes a request on a different URL;
- The ESP receives those requests and responds with the corresponding HTML files.

Assembling the Circuit

For this project, you need to wire a BME280 sensor to your board and four LEDs.

Parts Required

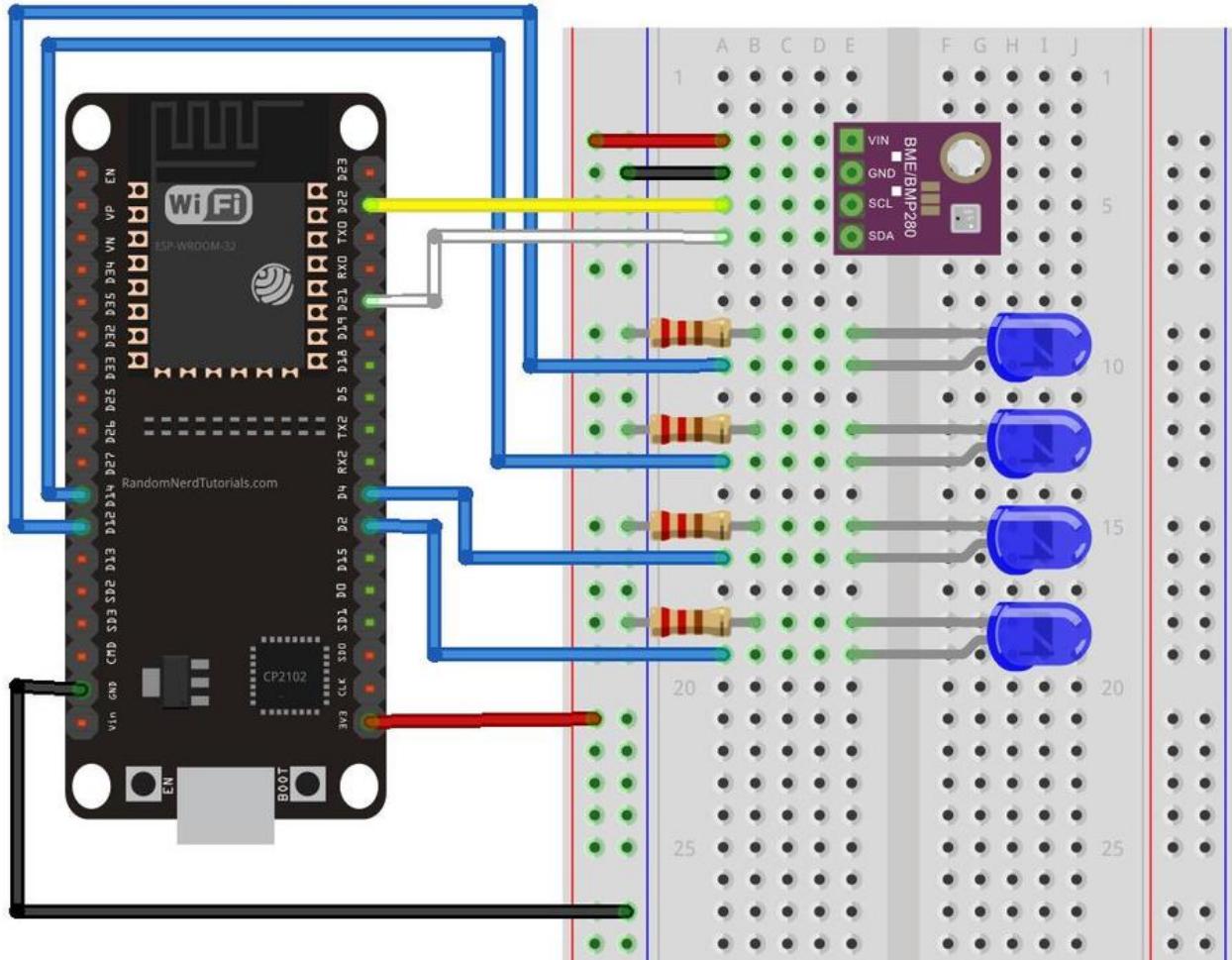
You need the following parts:

- [BME280 sensor module](#)
- 4x [LEDs](#)
- 4x [220Ω resistors](#)

- [ESP32](#) or [ESP8266](#)
- [Breadboard](#)
- [Jumper wires](#)

ESP32 - Schematic Diagram

Follow the next schematic diagram for the ESP32.

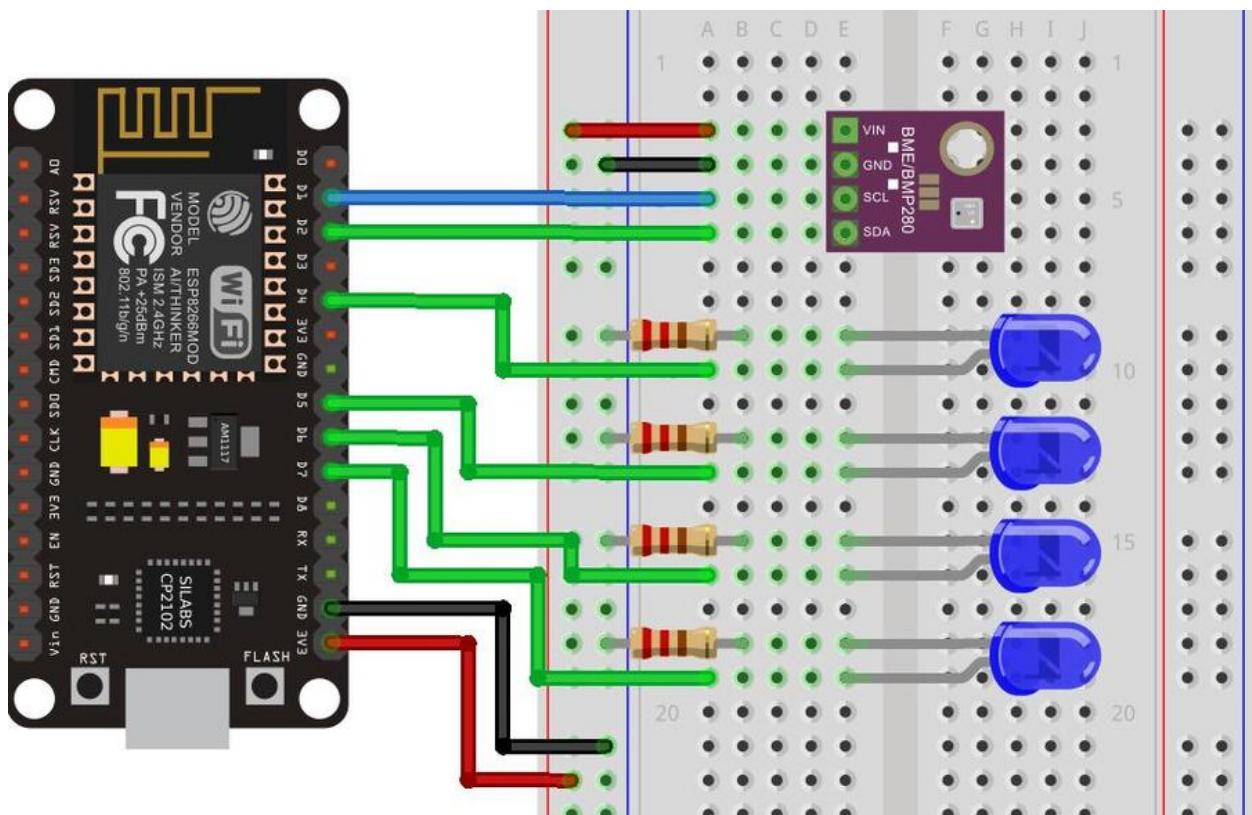


Here's a list with the GPIO connections:

- BME280 SCL → GPIO 22
- BME280 SDA → GPIO 21
- LEDs → GPIOs 2, 4, 12, 14

ESP8266 - Schematic Diagram

Follow the next schematic diagram if you're using an ESP8266.



Here's a list with the GPIO connections:

- BME280 SCL → GPIO 5 (D1)
- BME280 SDA → GPIO 4 (D2)
- LEDs → GPIOs 2 (D4), 12 (D6), 13* (D7), 14 (D5)

* In previous projects, we control GPIO 4 instead of GPIO 13. However, in this particular example, GPIO 4 is connected to the BME280 sensor. So, we're controlling GPIO 13 instead.

Building the Web Page

To build the web page for this project, make sure you place all these next files inside the *data* folder under your project folder:

- *about.html*
- *outputs.html*
- *sensors.html*
- *style.css*
- *script.js*
- *script-outputs.js*
- *script-sensors.js*
- *favicon.png*

HTML Files

The HTML file for the *outputs* page and the *sensor readings* are the same as in Units 2.5 and 3.3, respectively. The only differences are the HTML tags to build the navigation bar. The *about* web page contains a paragraph tag with some text.

You can download all the HTML files at the end of this Unit.

These are the HTML tags used to build the top navigation bar:

```
<div class="topnav" id="myTopnav">
  <a href="#" class="disabled">ESP Web Server</a>
  <div class="topnav-right">
    <a href="outputs">Outputs</a>
    <a href="sensors">Sensors</a>
    <a href="/">About</a>
  </div>
  <a href="javascript:void(0);" class="icon" onclick="handleNavBar()">
    <i class="fa fa-bars"></i>
  </a>
</div>
```

These lines should replace the old *topnav* `<div>` section of previous projects.

The following line defines the text that shows up on the left (it's static text and not clickable):

```
<a href="#" class="disabled">ESP Web Server</a>
```

Then, add the menus to the navigation bar and the URL path they should redirect to:

```
<div class="topnav-right">
  <a href="outputs">Outputs</a>
  <a href="sensors">Sensors</a>
  <a href="/">About</a>
</div>
```

In our case, we have three menus: *Outputs*, *Sensors* and *About*. You can add more menus if you want to have more pages.

The following lines are needed to handle the navigation bar menu on smaller screens (like your smartphone screen).

```
<a href="javascript:void(0);" class="icon" onclick="handleNavBar()">
  <i class="fa fa-bars"></i>
</a>
```

All these previous lines should be included in all your HTML files to have the navigation bar on all pages.

CSS File

The CSS files contain the same styles of previous projects and styles to customize the top navigation bar. These are the styles used for the navigation bar:

```
/* Style for navigation bar */
/* Add a background color to the top navigation */
.topnav {
  overflow: hidden;
  background-color: #0A1128;
}

/* Style the links inside the navigation bar */
.topnav a {
  float: left;
  display: block;
```

```

    color: white;
    text-align: center;
    padding: 16px 18px;
    text-decoration: none;
    font-size: 1.2rem;
}

/* Change the color of links on hover */
.topnav a:hover {
    background-color: #186775;
    color: white;
}

.disabled {
    pointer-events: none;
}

/* Hide the link that should open and close the topnav on small screens */
.topnav .icon {
    display: none;
}

.topnav-right{
    float:right;
}

/* When the screen is less than 600 pixels wide, hide all links, except for the first one.*/
@media screen and (max-width: 600px) {
    .topnav-right a {display: none;}
    .topnav a.icon {
        float: right;
        display: block;
    }
}

/* The "responsive" class is added to the topnav with JavaScript when the user clicks on the icon.
This class makes the topnav look good on small screens (display the links vertically instead of horizontally) */
@media screen and (max-width: 600px) {
    .topnav.responsive {position: relative;}
    .topnav.responsive .icon {
        position: absolute;
        right: 0;
        top: 0;
    }
    .topnav.responsive a, .topnav.responsive .topnav-right {
        float: none;
        display: block;
        text-align: left;
    }
}

```

The CSS file is commented so that you understand how those lines work. For a more in-depth tutorial about formatting a navigation bar using CSS, we recommend taking a look at: [How TO - Responsive Top Navigation.](#)

JavaScript Files

The *script-outputs.js* and *script-sensors.js* are the same JavaScript files used in Units 2.5 and 3.3, respectively.

As for the *script.js* file, it should be as follows:

```
// Handle Top Navigation Bar
function handleNavBar() {
    var x = document.getElementById("myTopnav");
    if (x.className === "topnav") {
        x.className += " responsive";
    }
    else {
        x.className = "topnav";
    }
}
```

This allows us to have a responsive top navigation bar on all screens.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* file for the ESP32 should be as shown below. Note that you need to include the Arduino JSON library to handle JSON strings; and the Adafruit BME280 and Adafruit Unified sensor libraries to interface with the BME280 sensor.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
```

```
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit BME280 Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    adafruit/Adafruit BME280 Library @ ^2.1.0
    adafruit/Adafruit Unified Sensor @ ^1.1.4
board_build.filesystem = littlefs
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");
```

```

// Set number of outputs
#define NUM_OUTPUTS 4

// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Create an Event Source on /events
AsyncEventSource events("/events");

// Json Variable to Hold Sensor Readings
JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 10000;

// Create a sensor object
Adafruit_BME280 bme;// BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpions"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpions"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {

```

```

        notifyClients(getOutputStates());
    }
    else{
        int gpio = atoi((char*)data);
        digitalWrite(gpio, !digitalRead(gpio));
        notifyClients(getOutputStates());
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client-
>remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin()) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

void setup() {

```

```

// Serial port for debugging purposes
Serial.begin(115200);

// Set GPIOs as outputs
for (int i =0; i<NUM_OUTPUTS; i++){
    pinMode(outputGPIOs[i], OUTPUT);
}

initBME();
initWiFi();
initSPIFFS();
initWebSocket();

server.serveStatic("/", SPIFFS, "/");

// Web Server Root URL
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/about.html", "text/html");
});

// Web Server /sensors URL
server.on("/sensors", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/sensors.html", "text/html");
});

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

// Route for /outputs
server.on("/outputs", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/outputs.html", "text/html", false);
});

events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n",
                     client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);

// Start server
server.begin();
}

void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 10 seconds
}

```

```
    events.send("ping",NULL,millis());
    events.send(getSensorReadings().c_str(),"new_readings" ,millis());
    lastTime = millis();
}

ws.cleanupClients();
}
```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";
```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the Unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <Adafruit_BME280.h>
#include <Adafruit_Sensor.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set number of outputs
#define NUM_OUTPUTS 4

// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 13, 12, 14};

// Create an Event Source on /events
AsyncEventSource events("/events");
// Json Variable to Hold Sensor Readings
```

```

JSONVar readings;

// Timer variables
unsigned long lastTime = 0;
unsigned long timerDelay = 10000;

// Create a sensor object
Adafruit_BME280 bme;// BME280 connect to ESP32 I2C (GPIO 21 = SDA, GPIO 22 = SCL)

// Init BME280
void initBME(){
    if (!bme.begin(0x76)) {
        Serial.println("Could not find a valid BME280 sensor, check wiring!");
        while (1);
    }
}

// Get Sensor Readings and return JSON object
String getSensorReadings(){
    readings["temperature"] = String(bme.readTemperature());
    readings["humidity"] = String(bme.readHumidity());
    readings["pressure"] = String(bme.readPressure()/100.0F);
    String jsonString = JSON.stringify(readings);
    return jsonString;
}

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {
            notifyClients(getOutputStates());
        }
        else{
            int gpio = atoi((char*)data);
            digitalWrite(gpio, !digitalRead(gpio));
            notifyClients(getOutputStates());
        }
    }
}

```

```

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client->id(), client-
>remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}
// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

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

    // Set GPIOs as outputs
    for (int i = 0; i < NUM_OUTPUTS; i++) {
        pinMode(outputGPIOs[i], OUTPUT);
    }

    initBME();
    initWiFi();
    initFS();
}

```

```

initWebSocket();

server.serveStatic("/", LittleFS, "/");

// Web Server Root URL
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/about.html", "text/html");
});

// Route for /sensors web page
server.on("/sensors", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/sensors.html", "text/html");
});

// Request for the latest sensor readings
server.on("/readings", HTTP_GET, [](AsyncWebServerRequest *request){
    String json = getSensorReadings();
    request->send(200, "application/json", json);
    json = String();
});

// Route for /outputs web page
server.on("/outputs", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(LittleFS, "/outputs.html", "text/html", false);
});

events.onConnect([](AsyncEventSourceClient *client){
    if(client->lastId()){
        Serial.printf("Client reconnected! Last message ID that it got is: %u\n",
                      client->lastId());
    }
    // send event with message "hello!", id current millis
    // and set reconnect delay to 1 second
    client->send("hello!", NULL, millis(), 10000);
});
server.addHandler(&events);

// Start server
server.begin();
}

void loop() {
    if ((millis() - lastTime) > timerDelay) {
        // Send Events to the client with the Sensor Readings Every 10 seconds
        events.send("ping", NULL, millis());
        events.send(getSensorReadings().c_str(), "new_readings", millis());
        lastTime = millis();
    }

    ws.cleanupClients();
}

```

Modify the code to include your network credentials, and it will work straight away.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = " REPLACE_WITH_YOUR_PASSWORD";
```

How The Code Works

Let's take a look at the code and see how it works to serve multiple web pages.

You need to include all the commands to handle all web pages. For example, in this case, you need to merge code from Unit 2.5 with the code from Unit 3.3.

Then, you need to add the lines of code that will serve different web pages depending on the request received.

In this example, the root URL should display the *About (about.html)* page:

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

The `/sensors` URL should show the *Sensors (sensors.html)* page:

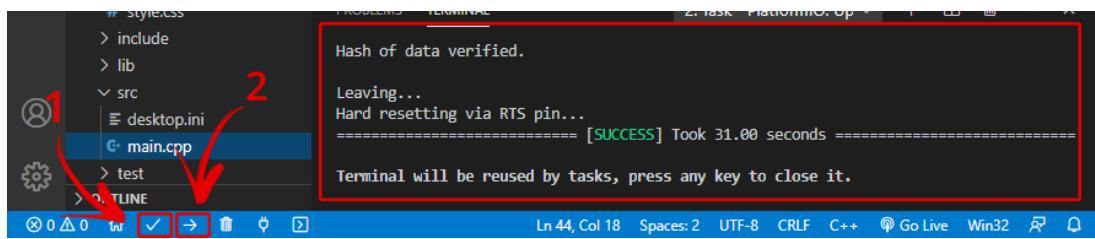
```
server.on("/sensors", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/sensors.html", "text/html");
});
```

The same for the *Outputs* page:

```
// Route for root / web page
server.on("/outputs", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(SPIFFS, "/outputs.html", "text/html", false);
});
```

Uploading Code

After modifying the code with your network credentials, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload the code to your board.

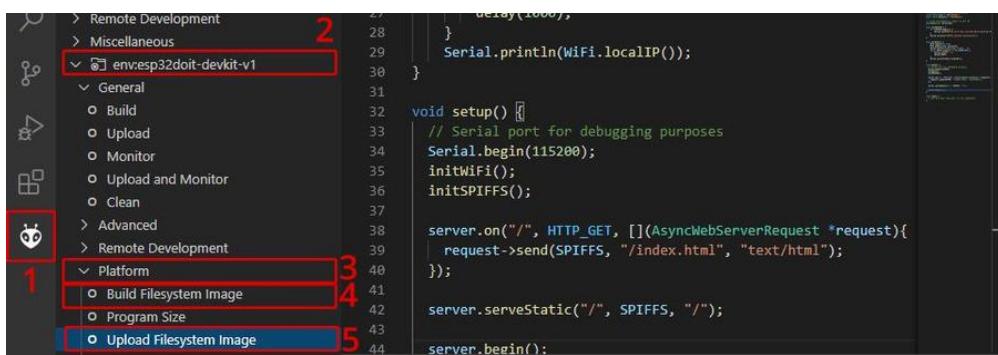


Uploading Filesystem Image

Important: to upload the filesystem image successfully you must close all serial connections (Serial Monitor) with your board.

Finally, upload the files (*outputs.html*, *sensors.html*, *about.html*, *style.css*, *script-outputs.js*, *script-sensors.js*, *script.js*, and *favicon.png*) to the filesystem:

1. Click on the PIO icon at the left bar. The project tasks should open.
2. Select **env:esp12e** or **env:esp32doit-devkit-v1** (it may be slightly different depending on the board you're using).
3. Expand the **Platform** menu.
4. Click on **Build Filesystem Image**.
5. Finally, click on **Upload Filesystem Image**.



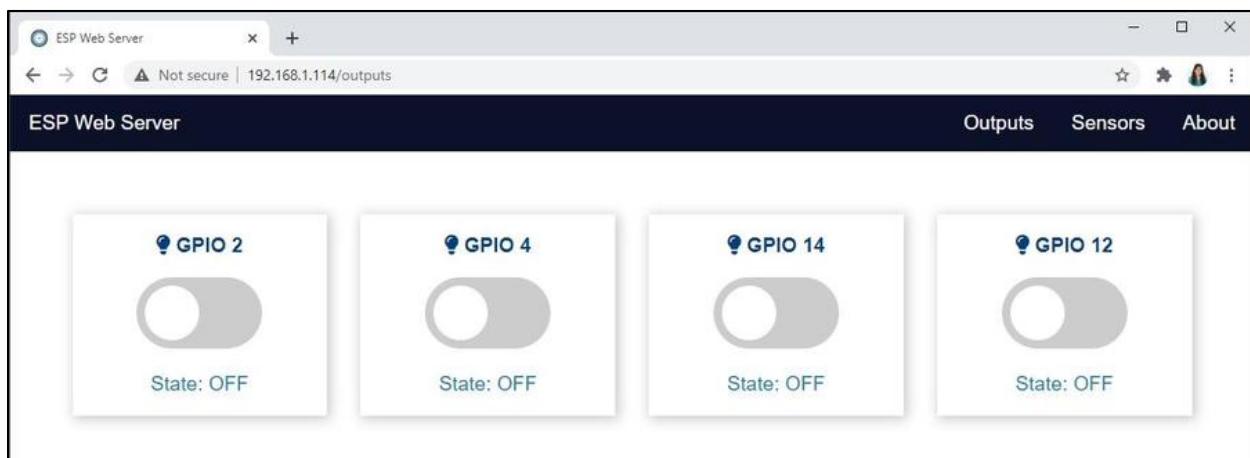
Demonstration

After successfully uploading all files, you can open the Serial Monitor to get the board IP address. Open a browser on your local network and insert the ESP IP address. You should get access to the web pages. Notice that each page shows the navigation top bar to navigate between pages easily.

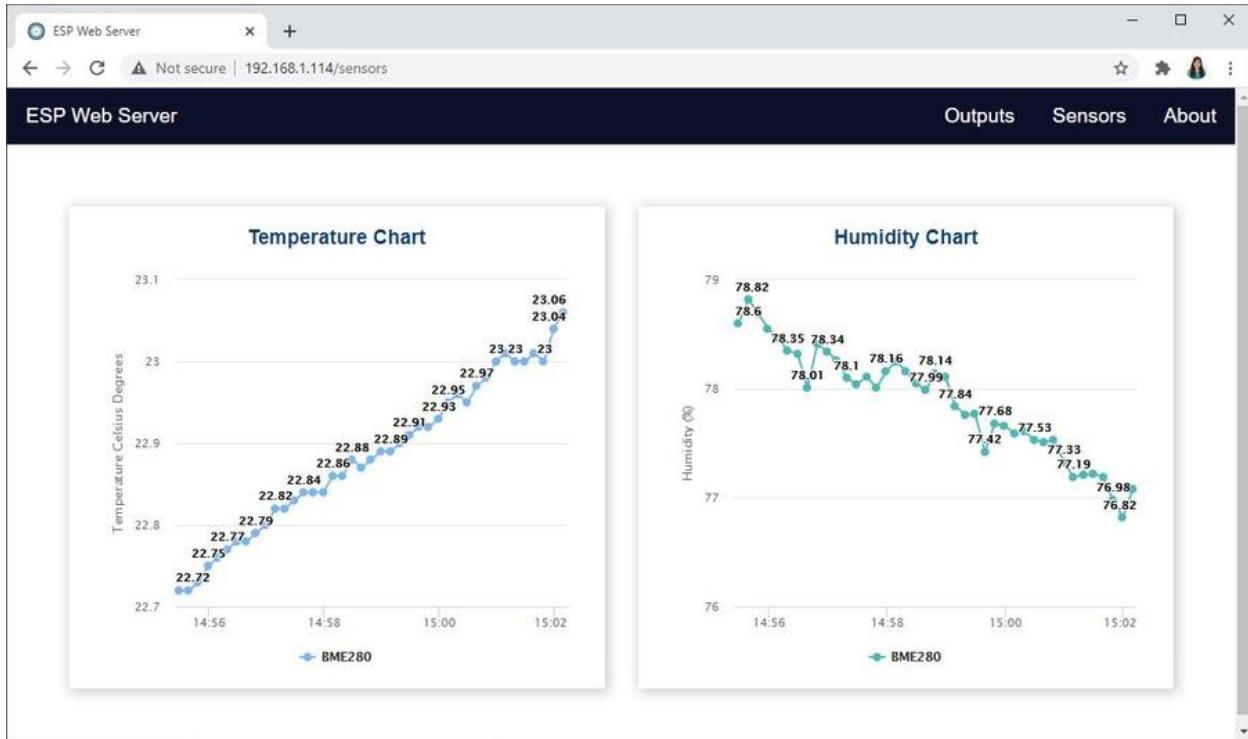
Here's the *About* page.



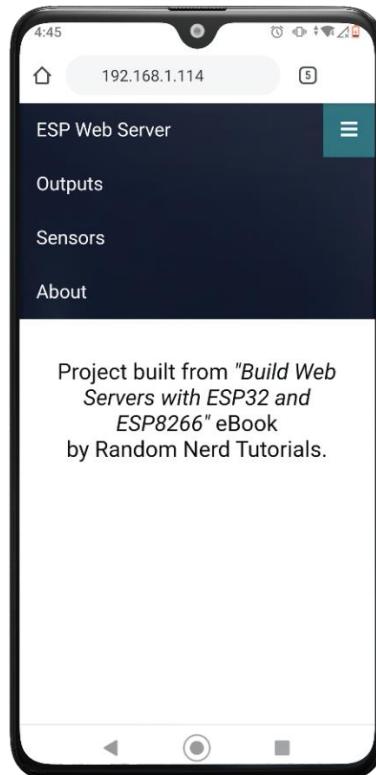
Click on the navigation bar to go to the other pages. This is the *Outputs* page:



And this is the *Sensors* web page.



If you're using a mobile device, it will open the navigation menu like this:



Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this Unit, you've learned how to serve multiple web pages and add a navigation top bar to navigate between pages. If you want multiple web pages, you need to have an HTML file for each page. You can use the same CSS file for all web pages or an individual CSS file for each page. As for the JavaScript file, depending on the page functionalities, you may have one for each page or one for multiple pages.

The navigation top bar is simply a list of hyperlinks that request different web pages.

4.5 – Over-the-air (OTA) Updates for Web Server



In this Unit, you'll learn how to do over-the-air (OTA) updates to your ESP32 or ESP8266 boards using the [Async ElegantOTA library](#). This library creates a web server that allows you to update new firmware (a new sketch) to your board without the need to make a serial connection between the ESP and your computer. Additionally, you can also upload new files to the filesystem (SPIFFS or LittleFS). The library is very easy to use, and it's compatible with the ESPAsyncWebServer library that we use to build the web server projects throughout this eBook.

By the end of this Unit, you'll be able to easily add OTA capabilities to your web server projects with the ESP32 or ESP8266 to upload new firmware and files to the filesystem wirelessly.

Overview

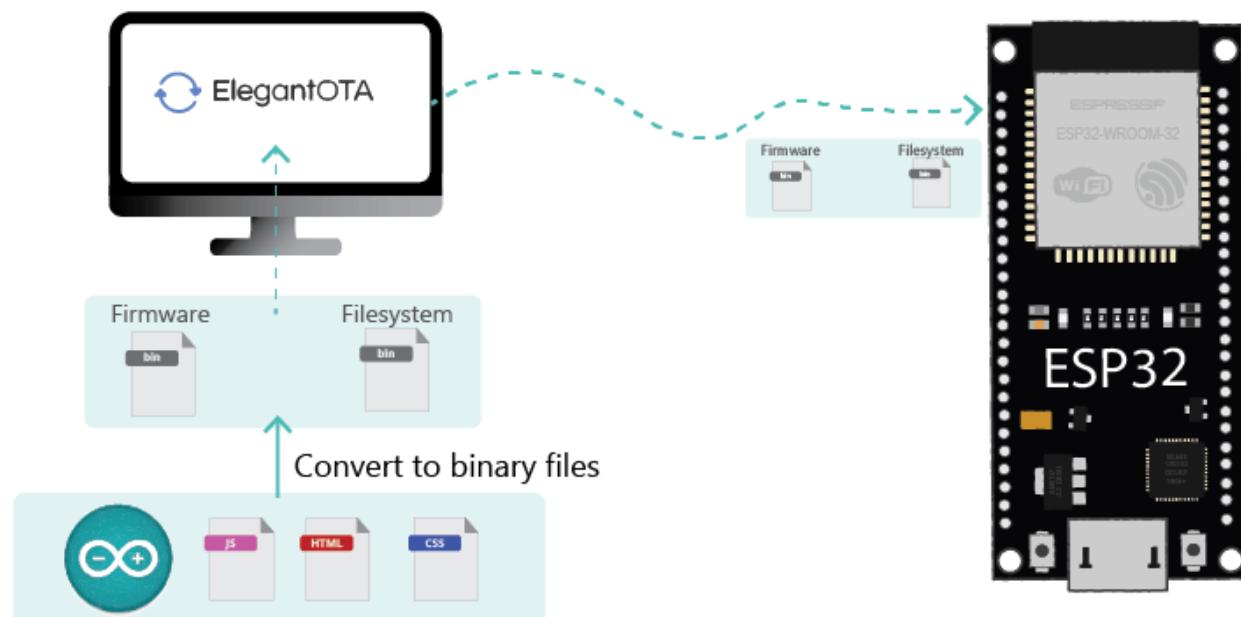
This tutorial covers:

- Add the ElegantOTA feature to your web server;
- Upload new firmware via OTA to ESP32 or ESP8266 boards;
- Upload files to SPIFFS/LittleFS via OTA to ESP32 or ESP8266 boards.

OTA (Over-the-Air) Programming

OTA (Over-the-Air) update is the process of loading new firmware to the ESP boards using a Wi-Fi connection rather than serial communication. This is extremely useful when physical access to the boards is difficult.

There are different ways to perform OTA updates. In this tutorial, we'll cover how to do that using the [Async ElegantOTA library](#). In our opinion, this is one of the best and easiest ways to perform OTA updates.



The Async ElegantOTA library lets you create a web server that you can access on your local network to upload new firmware or files to the filesystem (SPIFFS/LittleFS). The files you upload should be in *.bin* format. We'll show you later in the tutorial how to get your files in *.bin* format.

The only disadvantage of OTA programming is that you need to add the code for OTA to every sketch you upload so that you're able to use OTA in the future. In the case of the Async ElegantOTA library, it consists of just three lines of code.

Async ElegantOTA Library

As mentioned previously, there are a bunch of alternatives for OTA programming. The web server projects built throughout this eBook use the ESPAsyncWebServer library. So, we wanted a solution that was compatible with that library. The [Async ElegantOTA](#) library is just perfect for what we want:



- It is compatible with the ESPAsyncWebServer library;
- You just need to add three lines of code to add OTA capabilities to your “regular” Async Web Server;
- It allows you to update not only new firmware to the board, but also files to the filesystem (SPIFFS or LittleFS);
- It provides a beautiful and modern web server interface;
- It's reliable and it works extremely well.

OTA Updates – Quick Summary

To add OTA capabilities to your projects using the Async ElegantOTA library, follow these steps:

- Include these libraries in the `platformio.ini` file of your project: **AsyncElegantOTA**, **AsyncTCP** and **ESPAsyncWebServer** libraries;
- Include Async ElegantOTA library at the beginning of your code: `#include <AsyncElegantOTA.h>;`
- Add this line `AsyncElegantOTA.begin(&server);` before `server.begin();`
- Open your browser and go to `http://<IPAddress>/update`, where `<IPAddress>` is your ESP IP address;
- Upload a `.bin` file to update firmware or filesystem.

How does OTA Web Updater Work?

Although the points below may look complicated, don't worry – it's all easy to do!

- First, we need to upload a sketch to our ESP board through a standard serial port connection. This code contains instructions to create an OTA Updating web server.
- This web server will then let us upload a new sketch and/or files through a web browser rather than through a serial connection.
- Most importantly, that new sketch (and every later sketch that we upload through the web browser) must include a few lines of code calling OTA routines to let us continue making updates over-the-air in future
- Note that if you later upload a sketch without these OTA lines, you won't be able to access an OTA webserver from this board unless you go through this whole process again.

Install Async ElegantOTA Library

To use the Async ElegantOTA library, include it in your `platformio.ini` file. You also need to include the ESPAsyncWebServer library. Add these libraries as follows:

```
lib_deps = ESP Async WebServer  
ayushsharma82/AsyncElegantOTA @ ^2.2.5
```

Async ElegantOTA Basic Example

Let's start with the basic example provided by the library. This example creates a simple web server on the ESP. The root URL displays some text and the `/update` URL displays the interface to update firmware and filesystem.

Follow the section for the board you're using.

ESP32

Follow this section if you're using an ESP32 board.

The `platformio.ini` file should be like this:

```
[env:esp32doit-devkit-v1]  
platform = espressif32  
board = esp32doit-devkit-v1  
framework = arduino  
monitor_speed = 115200  
lib_deps = ESP Async WebServer  
ayushsharma82/AsyncElegantOTA @ ^2.2.5
```

Copy the following code to the `main.cpp` file of your project.

```
#include <Arduino.h>  
#include <WiFi.h>  
#include <AsyncTCP.h>  
#include <ESPAsyncWebServer.h>  
#include <AsyncElegantOTA.h>  
  
const char* ssid = "REPLACE_WITH_YOUR_SSID";  
const char* password = "REPLACE_WITH_YOUR_PASSWORD";  
  
AsyncWebServer server(80);
```

```

void setup(void) {
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.println("");

    // Wait for connection
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.print("Connected to ");
    Serial.println(ssid);
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());

    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request) {
        request->send(200, "text/plain", "Hi! I am ESP32.");
    });

    AsyncElegantOTA.begin(&server);      // Start ElegantOTA
    server.begin();
    Serial.println("HTTP server started");
}

void loop(void) {
    AsyncElegantOTA.loop();
}

```

Insert your network credentials, and the code should work straight away:

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

ESP8266

Follow this section if you're using an ESP8266.

The *platformio.ini* file should be like this:

```

[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
ayushsharma82/AsyncElegantOTA @ ^2.2.5

```

Copy the following code to the *main.cpp* file of your project.

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <AsyncElegantOTA.h>
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

AsyncWebServer server(80);
void setup(void) {
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.println("");

    // Wait for connection
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.print("Connected to ");
    Serial.println(ssid);
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());

    server.on("/", HTTP_GET, [](){AsyncWebRequest *request) {
        request->send(200, "text/plain", "Hi! I am ESP8266.");
    });
    AsyncElegantOTA.begin(&server); // Start ElegantOTA
    server.begin();
    Serial.println("HTTP server started");
}
void loop(void) {
    AsyncElegantOTA.loop();
}

```

Insert your network credentials, and the code should work straight away:

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

How the Code Works

First, include the necessary libraries. These are the libraries for the ESP32:

```

#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <AsyncElegantOTA.h>

```

And these are the libraries for the ESP8266:

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <AsyncElegantOTA.h>
```

Insert your network credentials in the following variables so that the ESP can connect to your local network.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

Create an `AsyncWebServer` object on port 80:

```
AsyncWebServer server(80);
```

In the `setup()`, initialize the Serial Monitor:

```
Serial.begin(115200);
```

Initialize Wi-Fi and print the ESP IP address:

```
WiFi.mode(WIFI_STA);
WiFi.begin(ssid, password);
Serial.println("");

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

Serial.println("");
Serial.print("Connected to ");
Serial.println(ssid);
Serial.print("IP address: ");
Serial.println(WiFi.localIP());
```

Then, handle the client requests. The following lines, send some text `Hi! I am ESP32/ESP8266.` when you access the root (/) URL:

```
server.on("/", HTTP_GET, [](AsyncWebRequest *request) {
    request->send(200, "text/plain", "Hi! I am ESP32.");
});
```

If your web server needs to handle more requests, you can add them.

Then, add the following line to start ElegantOTA:

```
AsyncElegantOTA.begin(&server); // Start ElegantOTA
```

Finally, initialize the server:

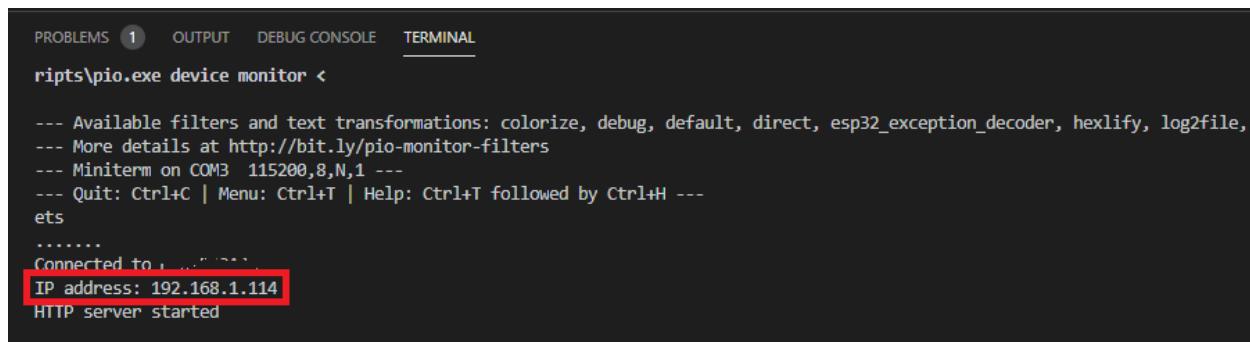
```
server.begin();
```

Lastly, add the following in the `loop()`:

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

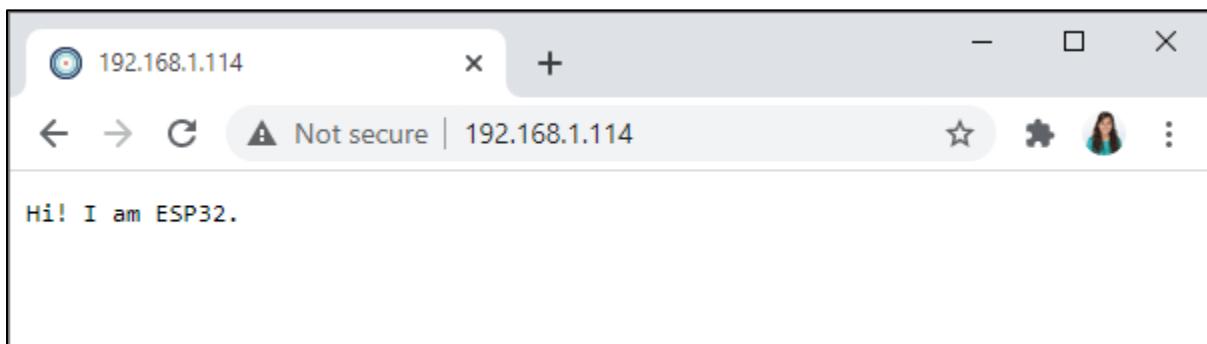
Access the Web Server

Upload the code to your board. After uploading, open the Serial Monitor at a baud rate of 115200 and get your board IP address.

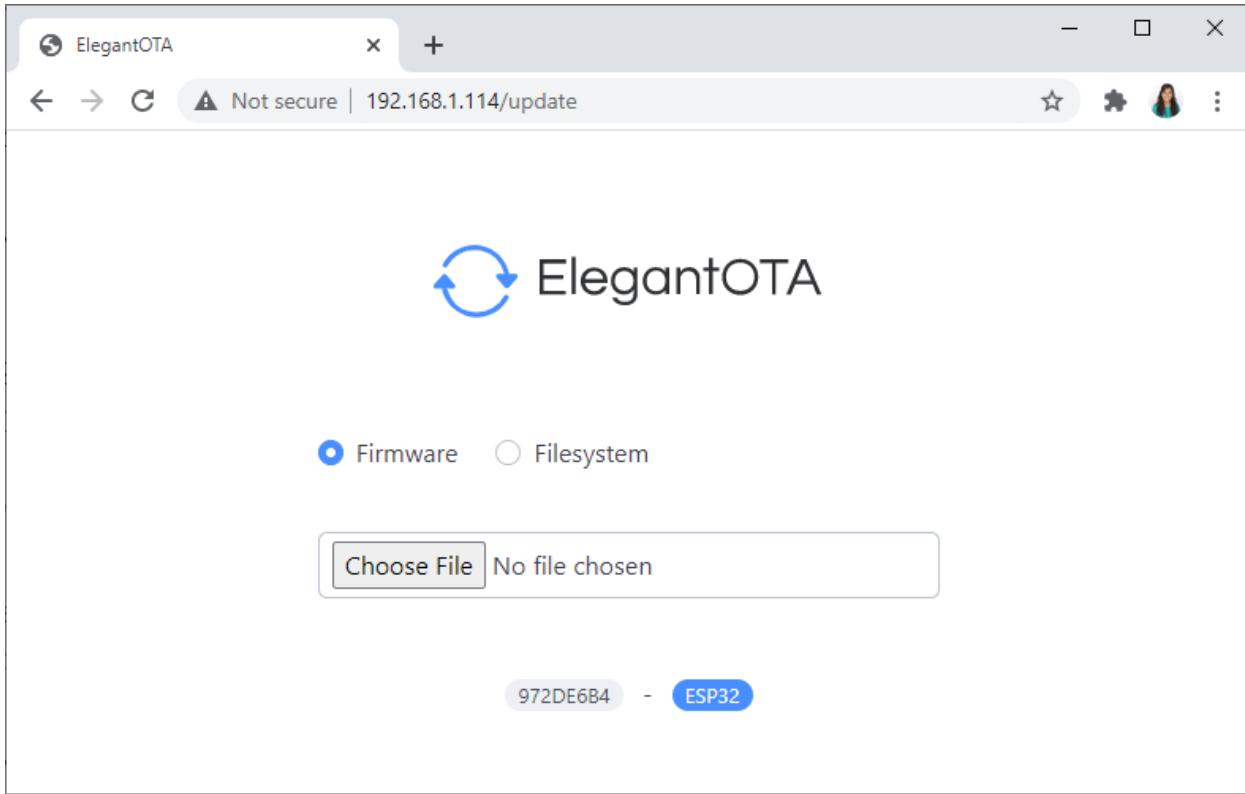


```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
rpi5\pio.exe device monitor <
--- Available filters and text transformations: colorize, debug, default, direct, esp32_exception_decoder, hexlify, log2file,
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on COM3 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ets
.....
Connected to rpi5 (192.168.1.114)
IP address: 192.168.1.114
HTTP server started
```

In your local network, open your browser and type the IP address. You should get access to the root (/) web page with some text displayed.



Now, imagine that you want to modify your web server code. To do that via OTA, go to the ESP IP address followed by `/update`. The following web page loads.



Follow the next sections to learn how to upload new firmware using Async ElegantOTA.

Upload New Firmware OTA

The firmware file that you need to upload via OTA needs to be in `.bin` format. VS Code automatically generates the `.bin` file for your project when you compile the code. The file is called `firmware.bin` and it is saved in your project folder on the following path (or similar depending on the board you're using):

`.pio/build/esp32doit-devkit-v1/firmware.bin`

That's the `.bin` file you should upload using the Async ElegantOTA web page if you want to upload new firmware.

Upload a New Web Server Sketch – Example

Let's see a practical example. Imagine that after uploading the previous sketch, you want to upload a new one. To make things simpler, we'll upload a code from Unit 1.1 (it doesn't need files for the filesystem—in the next section, we'll see how to upload files like HTML, CSS, and JavaScript files)

1. Copy the following code to your *main.cpp* file. Don't forget to insert your network credentials.

ESP32

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

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
  <style>
    html {
      font-family: Arial;
      text-align: center;
    }
    body {
      max-width: 400px;
      margin: 0px auto;
    }
  </style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
)rawliteral";
```

```

</body>
</html>
)rawliteral';

void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

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

    server.on("/", HTTP_GET, [] (AsyncWebRequest *request){
        request->send(200, "text/html", index_html);
    });

    AsyncElegantOTA.begin(&server);      // Start ElegantOTA

    // Start server
    server.begin();
}

void loop() {
    AsyncElegantOTA.loop();
}

```

ESP8266

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include <AsyncElegantOTA.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>

```

```

<title>ESP Web Server</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" href="data:,>
<style>
  html {
    font-family: Arial;
    text-align: center;
  }
  body {
    max-width: 400px;
    margin: 0px auto;
  }
</style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
)rawliteral";

void initWiFi() {
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi ..");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println(WiFi.localIP());
}

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

  server.on("/", HTTP_GET, [](){AsyncWebRequest *request){
    request->send(200, "text/html", index_html);
  });

  AsyncElegantOTA.begin(&server);      // Start ElegantOTA

  // Start server
  server.begin();
}

void loop() {
  AsyncElegantOTA.loop();
}

```

This is the same code used in the project from Unit 1.1, but it contains the needed lines of code to handle ElegantOTA:

```
#include <AsyncElegantOTA.h>
```

```
AsyncElegantOTA.begin(&server); // Start ElegantOTA
```

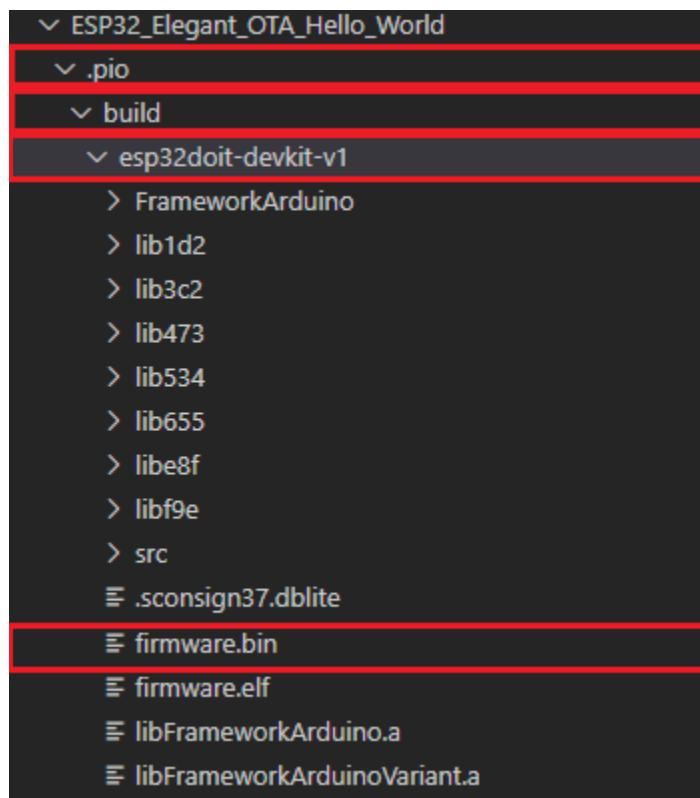
```
AsyncElegantOTA.loop();
```

2. Compile your code - click on the **Build** icon (don't forget to insert your network credentials).



3. Now, in the Explorer tab of VS Code, you can check that you have a *firmware.bin* file under the project folder on the following path (or similar):

```
.pio/build/esp32doit-devkit-v1/firmware.bin
```

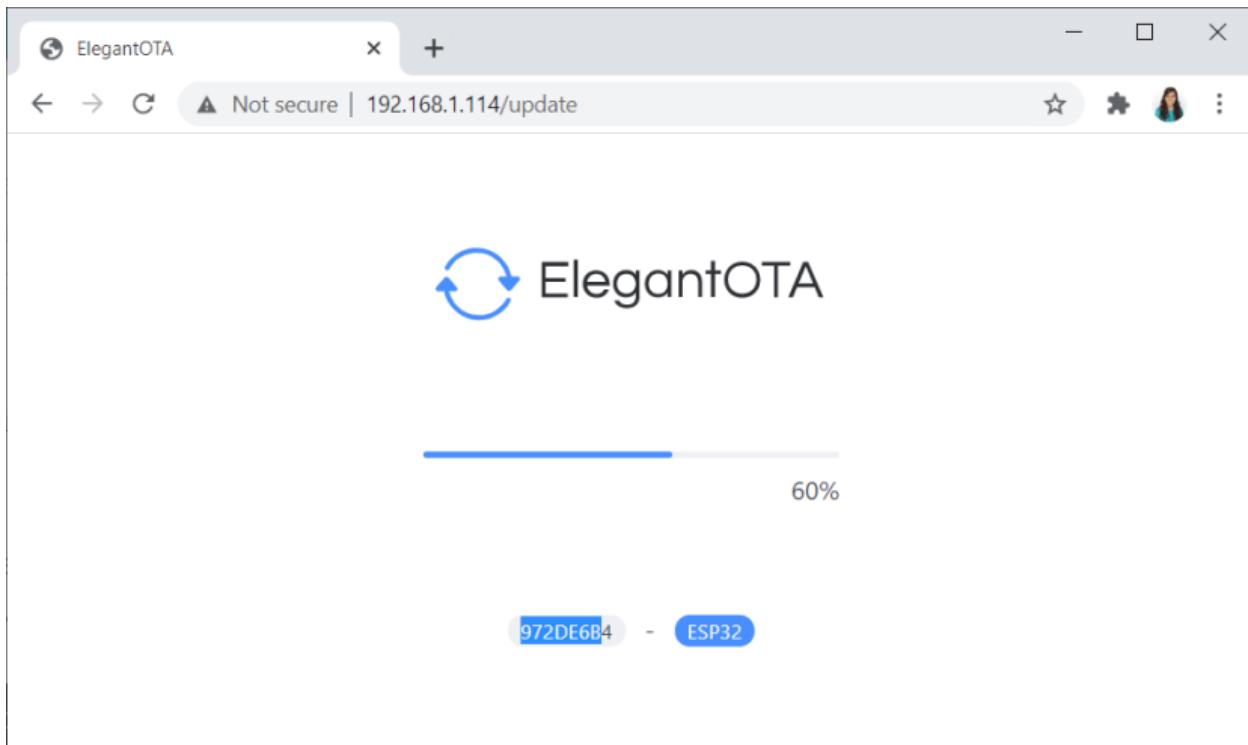


4. Now, you just need to upload that file using the ElegantOTA page. Go to your ESP IP address followed by /update. Make sure you have the **firmware** option selected.

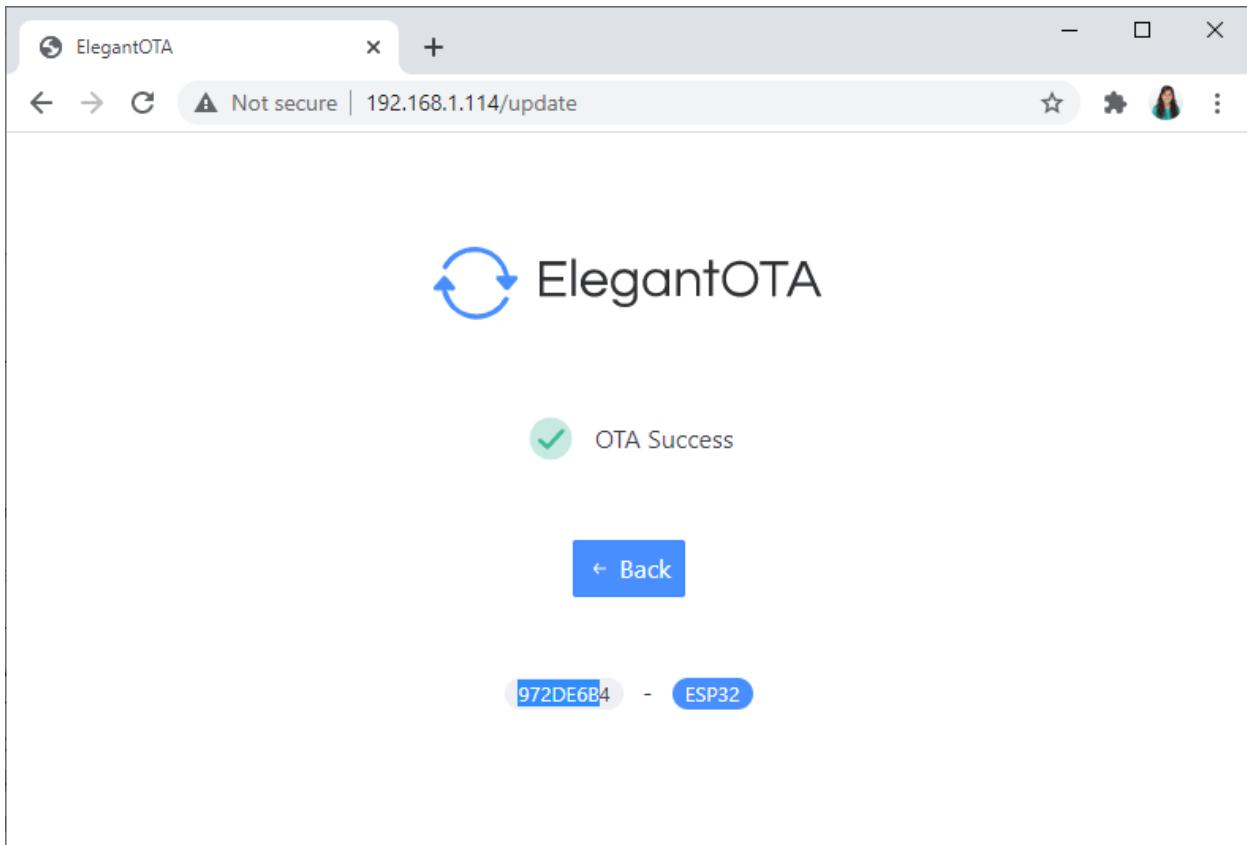
5. Click on **Choose File**, navigate through the folders on your computer, and select your project's file.

Name	Date modified	Type	Size
FrameworkArduino	2/1/2021 12:42 PM	File folder	
lib1d2	2/1/2021 12:42 PM	File folder	
lib3c2	2/1/2021 12:42 PM	File folder	
lib473	2/1/2021 12:42 PM	File folder	
lib534	2/1/2021 12:42 PM	File folder	
lib655	2/1/2021 12:42 PM	File folder	
libe8f	2/1/2021 12:42 PM	File folder	
libf9e	2/1/2021 12:42 PM	File folder	
src	2/1/2021 12:42 PM	File folder	
.sconsign37.dblite	2/1/2021 12:42 PM	DBLITE File	1,046 KB
firmware.bin	2/1/2021 12:42 PM	BIN File	964 KB
firmware.elf	2/1/2021 12:42 PM	ELF File	12,405 KB
libFrameworkArduino.a	2/1/2021 12:42 PM	A File	24,145 KB
libFrameworkArduinoVariant.a	2/1/2021 12:42 PM	A File	1 KB
partitions.bin	2/1/2021 12:42 PM	BIN File	3 KB

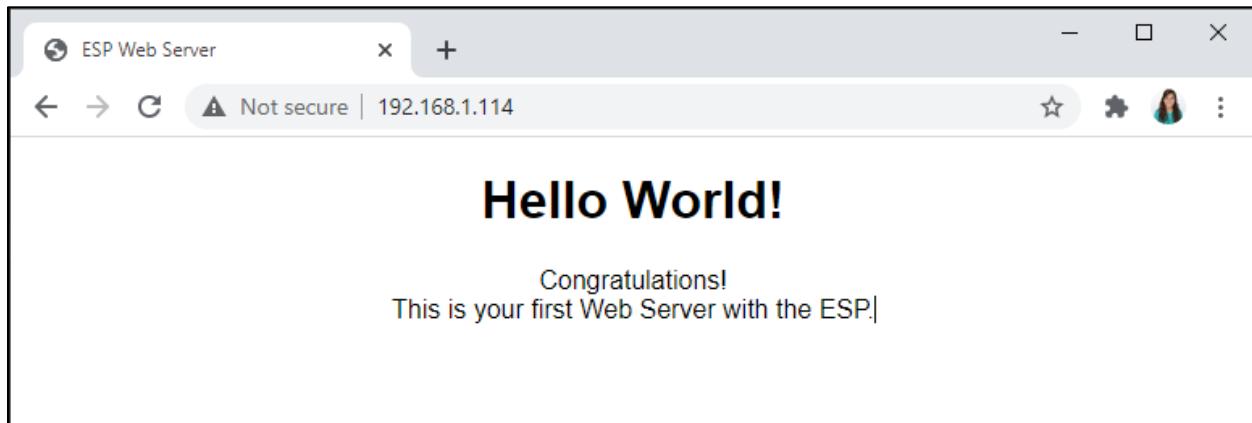
6. Wait until the progress bar reaches 100%.



7. When it's finished, click on the **Back** button.



8. Then, you can go to the root (/) URL to access the new web server. This is the page that you should see when you access the ESP IP address on the root (/) URL.



Because we've also added OTA capabilities to this new web server, we can upload a new sketch in the future if needed. You just need to go to the ESP IP address followed by /update.

Congratulations, you've uploaded new code to your ESP via Wi-Fi using ElegantOTA.

Continue reading if you want to learn how to upload files to the ESP filesystem (SPIFFS or LittleFS) using Async ElegantOTA.

Upload Files to Filesystem OTA

In this section, you'll learn how to upload files to the ESP filesystem (SPIFFS or LittleFS) using Async ElegantOTA.

Web Server with Files from SPIFFS

Imagine the scenario that you need to upload files to the ESP filesystem, for example: configuration files; HTML, CSS, and JavaScript files to update the web server page; or any other file that you may want to save in the filesystem via OTA.

To show you how to do this, we'll use the web server from Unit 2.5.

ESP32

Follow this section if you're using an ESP32.

Your *platformio.ini* file should look like this:

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    arduino-libraries/Arduino_JSON @ 0.1.0
    ayushsharma82/AsyncElegantOTA @ ^2.2.5
```

Copy the following code to your *main.cpp* file:

```
#include <Arduino.h>
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "SPIFFS.h"
#include <Arduino_JSON.h>
#include <AsyncElegantOTA.h>
```

```

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

// Set number of outputs
#define NUM_OUTPUTS 4

// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize SPIFFS
void initSPIFFS() {
    if (!SPIFFS.begin(true)) {
        Serial.println("An error has occurred while mounting SPIFFS");
    }
    Serial.println("SPIFFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {

```

```

        notifyClients(getOutputStates());
    }
    else{
        int gpio = atoi((char*)data);
        digitalWrite(gpio, !digitalRead(gpio));
        notifyClients(getOutputStates());
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,
            void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

    // Set GPIOs as outputs
    for (int i =0; i<NUM_OUTPUTS; i++){
        pinMode(outputGPIOs[i], OUTPUT);
    }
    initSPIFFS();
    initWiFi();
    initWebSocket();

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(SPIFFS, "/index.html", "text/html", false);
    });

    server.serveStatic("/", SPIFFS, "/");
}

AsyncElegantOTA.begin(&server);      // Start ElegantOTA

```

```

// Start server
server.begin();
}

void loop() {
    ws.cleanupClients();
    AsyncElegantOTA.loop();
}

```

Don't forget to insert your network credentials.

```

const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

```

ESP8266

Follow this section if you're using an ESP8266.

Your *platformio.ini* file should be like this:

```

[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
    ayushsharma82/AsyncElegantOTA @ ^2.2.5
    arduino-libraries/Arduino_JSON @ 0.1.0
board_build.filesystem = littlefs

```

Copy the following code to your *main.cpp* file:

```

#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>
#include "LittleFS.h"
#include <Arduino_JSON.h>
#include <AsyncElegantOTA.h>

// Replace with your network credentials
const char* ssid = "REPLACE_WITH_YOUR_SSID ";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";

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

// Create a WebSocket object
AsyncWebSocket ws("/ws");

```

```

// Set number of outputs
#define NUM_OUTPUTS 4
// Assign each GPIO to an output
int outputGPIOs[NUM_OUTPUTS] = {2, 4, 12, 14};

// Initialize LittleFS
void initFS() {
    if (!LittleFS.begin()) {
        Serial.println("An error has occurred while mounting LittleFS");
    }
    Serial.println("LittleFS mounted successfully");
}

// Initialize WiFi
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print('.');
        delay(1000);
    }
    Serial.println(WiFi.localIP());
}

String getOutputStates(){
    JSONVar myArray;
    for (int i =0; i<NUM_OUTPUTS; i++){
        myArray["gpios"][i]["output"] = String(outputGPIOs[i]);
        myArray["gpios"][i]["state"] = String(digitalRead(outputGPIOs[i]));
    }
    String jsonString = JSON.stringify(myArray);
    return jsonString;
}

void notifyClients(String state) {
    ws.textAll(state);
}

void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
    AwsFrameInfo *info = (AwsFrameInfo*)arg;
    if (info->final && info->index == 0 && info->len == len && info->opcode == WS_TEXT) {
        data[len] = 0;
        if (strcmp((char*)data, "states") == 0) {
            notifyClients(getOutputStates());
        }
        else{
            int gpio = atoi((char*)data);
            digitalWrite(gpio, !digitalRead(gpio));
            notifyClients(getOutputStates());
        }
    }
}

void onEvent(AsyncWebSocket *server, AsyncWebSocketClient *client, AwsEventType type,

```

```

        void *arg, uint8_t *data, size_t len) {
    switch (type) {
        case WS_EVT_CONNECT:
            Serial.printf("WebSocket client #%u connected from %s\n", client-
>id(), client->remoteIP().toString().c_str());
            break;
        case WS_EVT_DISCONNECT:
            Serial.printf("WebSocket client #%u disconnected\n", client->id());
            break;
        case WS_EVT_DATA:
            handleWebSocketMessage(arg, data, len);
            break;
        case WS_EVT_PONG:
        case WS_EVT_ERROR:
            break;
    }
}

void initWebSocket() {
    ws.onEvent(onEvent);
    server.addHandler(&ws);
}

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

    // Set GPIOs as outputs
    for (int i =0; i<NUM_OUTPUTS; i++){
        pinMode(outputGPIOs[i], OUTPUT);
    }
    initFS();
    initWiFi();
    initWebSocket();

    // Route for root / web page
    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(LittleFS, "/index.html", "text/html", false);
    });

    server.serveStatic("/", LittleFS, "/");
    AsyncElegantOTA.begin(&server);      // Start ElegantOTA

    // Start server
    server.begin();
}

void loop() {
    ws.cleanupClients();
    AsyncElegantOTA.loop();
}

```

Don't forget to insert your network credentials.

```
const char* ssid = "REPLACE_WITH_YOUR_SSID";
const char* password = "REPLACE_WITH_YOUR_PASSWORD";
```

Update Firmware

After inserting your network credentials, save and compile the code.



Go to the ESP IP address followed by /update and upload the new firmware as shown previously.

Next, we'll see how to upload the files to the filesystem.

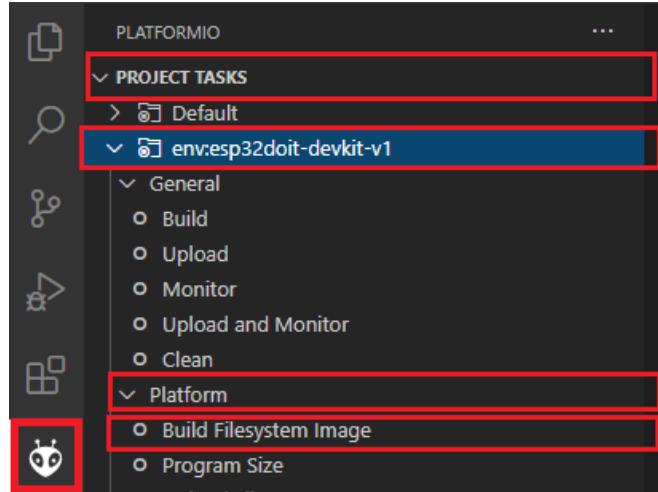
Update Filesystem

Under the project folder, create a folder called *data* and paste the following HTML, CSS, and JavaScript files (click on the links to download the files):

- [HTML file: index.html](#)
- [CSS file: style.css](#)
- [JavaScript file: script.js](#)

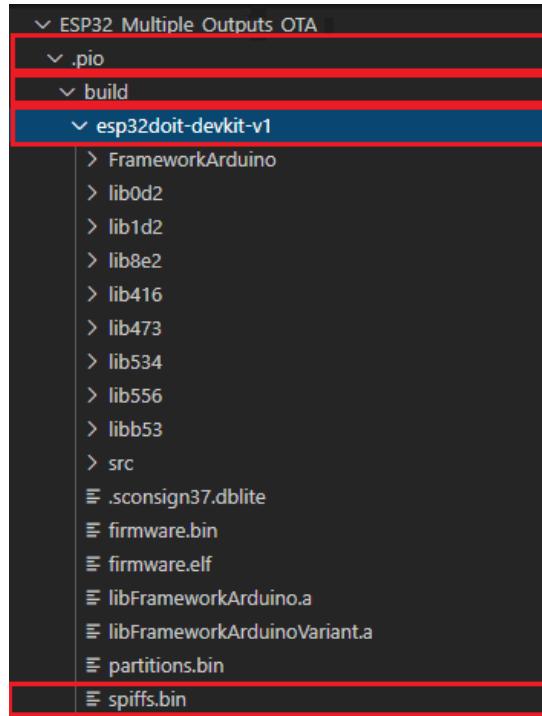
These are the same files used in Unit 2.5.

In VS Code, click on to the PIO icon and go to **Project Tasks > env:esp32doit-devkit-v1** (or similar) >**Platform > Build Filesystem Image**. This creates a *.bin* file from the files saved in the *data* folder.



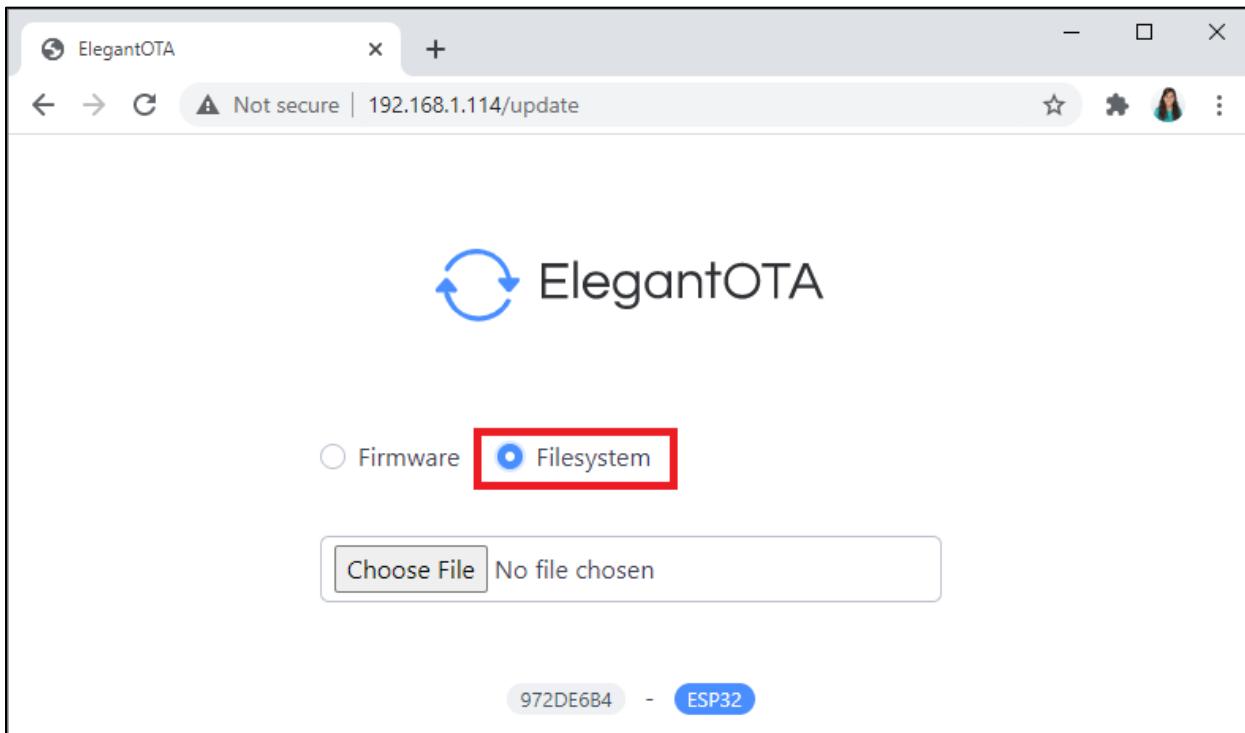
After building the filesystem image, you should have a *spiffs.bin* or *littlefs.bin* file in the following path (or similar):

.pio/build/esp32doit-devkit-v1/spiffs.bin

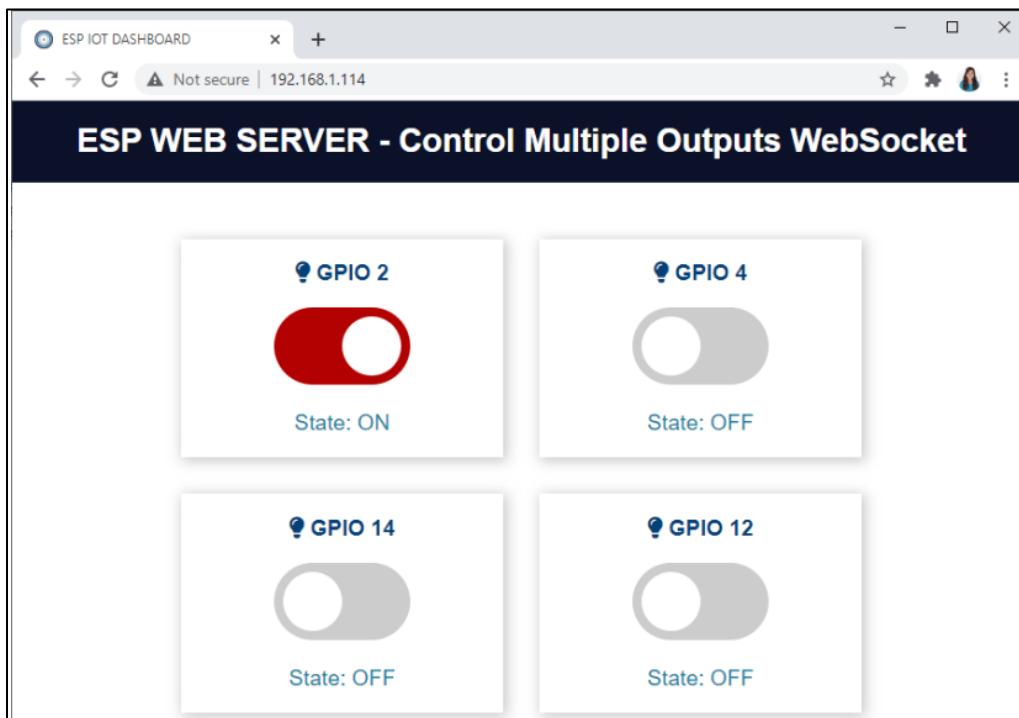


That's that file that you should upload to update the filesystem.

Go to your ESP IP address followed by /update. Make sure you have the **Filesystem** option selected and select the *spiffs.bin* or *littlefs.bin* file.



After successfully uploading, click the **Back** button. Go to the root (/) URL again.



You should get access to the previous web page that controls the ESP outputs using the Web Socket protocol (you are already familiar with this page from Unit 2.5).

To see the web server working, you can connect 4 LEDs to your ESP boards on GPIOs: 2, 4, 12, and 14. You should be able to control those outputs from the web page.

If you need to update something on your project, you just need to go to your ESP IP address followed by `/update`.

Congratulations! You've successfully uploaded files to the ESP filesystem using ElegantOTA.

Download Project Folder

You can download the folders for this Unit using the links below:

- [ESP32 ElegantOTA Demo](#)
- [ESP8266 ElegantOTA Demo](#)
- [ESP32 ElegantOTA Example 1](#)
- [ESP8266 ElegantOTA Example 1](#)
- [ESP32 ElegantOTA Example 2](#)
- [ESP8266 ElegantOTA Example 2](#)

Wrapping Up

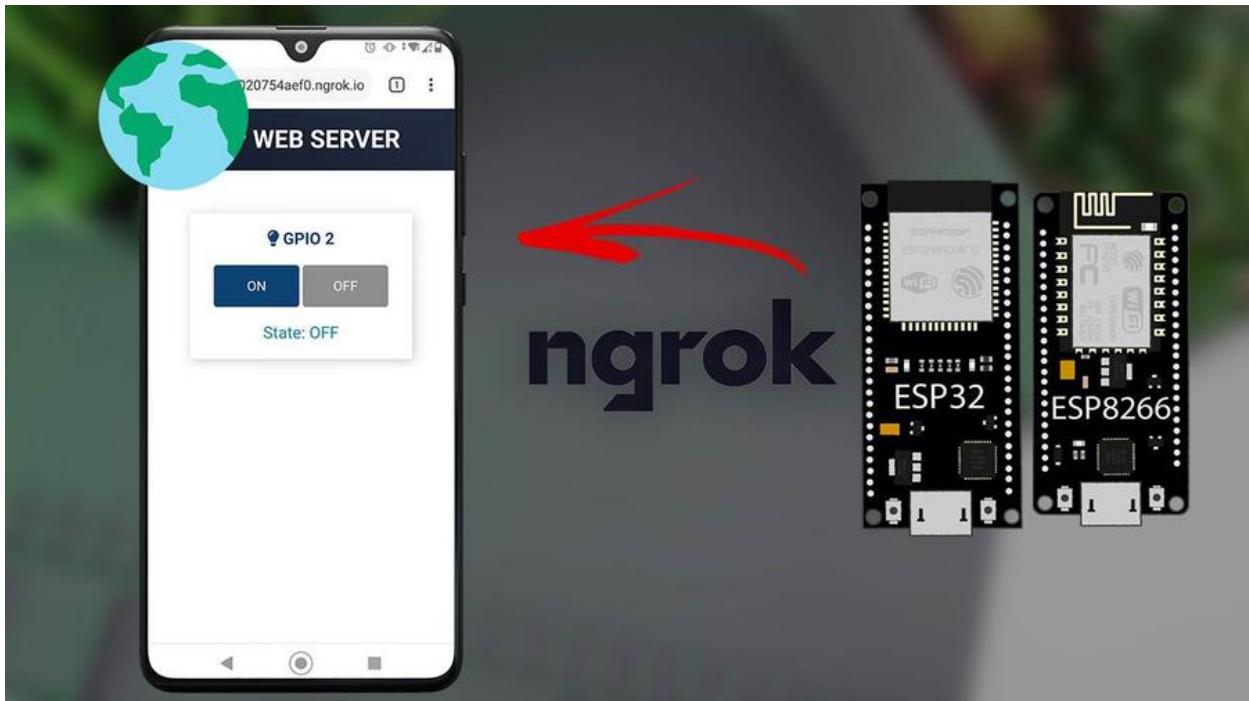
In this Unit, you've learned how to add OTA capabilities to your Async Web Servers using the Async ElegantOTA library. This library is super simple to use and allows you to upload new firmware or files to SPIFFS/LittleFS effortlessly using a web page.

In our opinion, the Async ElegantOTA library is one of the best options to handle OTA web updates.

EXTRA

Useful Resources

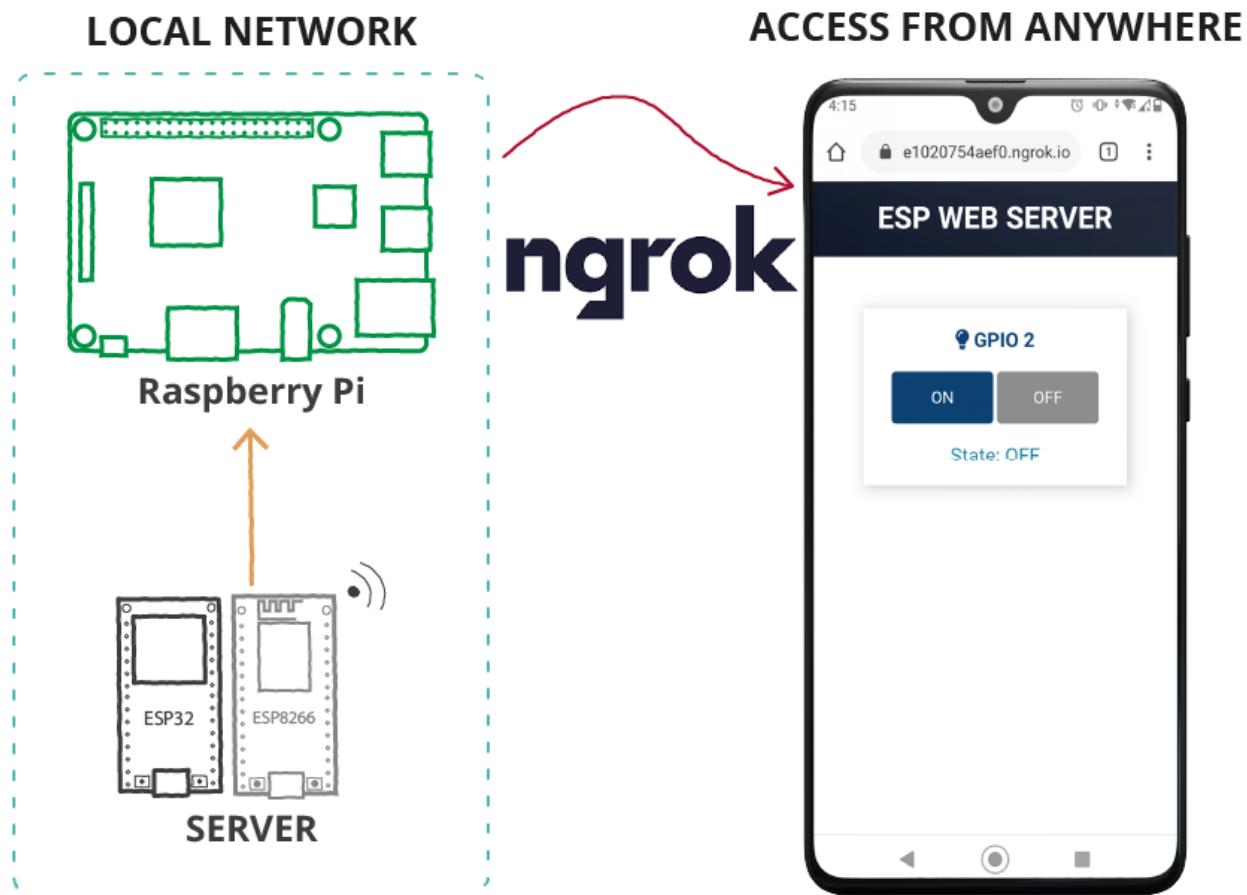
Access Your ESP32 and ESP8266 Web Servers from Anywhere



In this Unit, you're going to make your ESP32 and ESP8266 web servers accessible from anywhere in the world. In previous projects, the ESP web servers were only accessible when you are connected to your local network. With this project, you'll be able to monitor and control your boards from anywhere that you have an internet connection.

Overview

The next diagram shows a high-level overview of how everything works.



Note: this method requires having a computer running *ngrok* software. For a permanent solution, it is a good idea to run this software on the low-cost Raspberry Pi computer. Ngrok can also run on your Windows PC, Mac OS X or Linux.

Prerequisites (if using a Raspberry Pi)

Before continuing:

- You should be familiar with the Raspberry Pi board – [read Getting Started with Raspberry Pi](#);

- You should have the Raspberry Pi OS or Raspberry Pi OS (32-bit) Lite (previous called Raspbian) installed in your Raspberry Pi – [read Installing Raspbian Lite, Enabling and Connecting with SSH](#);
- You also need the following hardware: [Raspberry Pi board](#) – read [Best Raspberry Pi Starter Kits, MicroSD Card – 16GB Class10](#), and [Power Supply \(5V 2.5A\)](#);

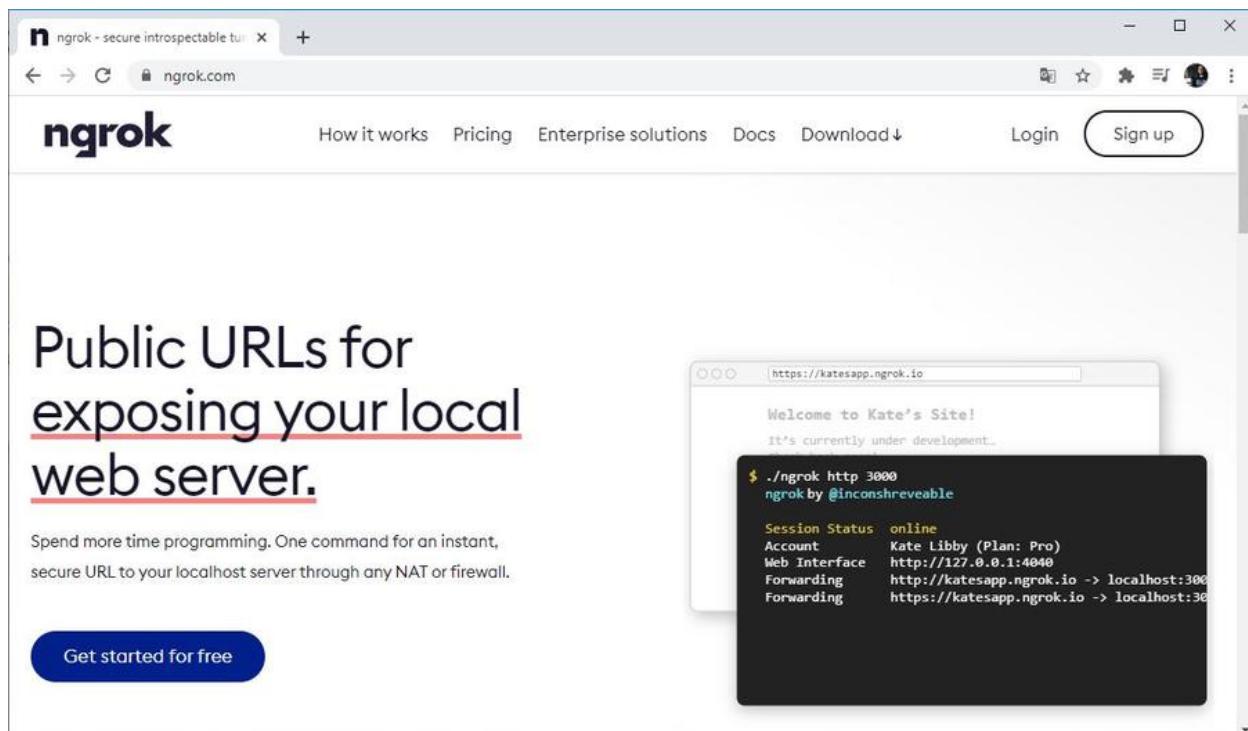
Getting the ESP IP Address

For this project, you can grab any ESP32 or ESP8266 web server code presented in this eBook. After uploading the code, get the board IP address. Save the IP address (in our case, the IP address is 192.168.1.112). You'll need it to run ngrok later.

Introducing ngrok

To make your web servers accessible from anywhere, you will use a free service called **ngrok** to create a tunnel to your local web server.

Go to <https://ngrok.com> to create an account.



The screenshot shows the ngrok website (<https://ngrok.com>) in a browser. The main heading is "Public URLs for exposing your local web server." Below it, a sub-headline says "Spend more time programming. One command for an instant, secure URL to your localhost server through any NAT or firewall." A blue button at the bottom left says "Get started for free". On the right side of the page, there is a terminal window showing the output of the command `./ngrok http 3000`. The terminal output includes:

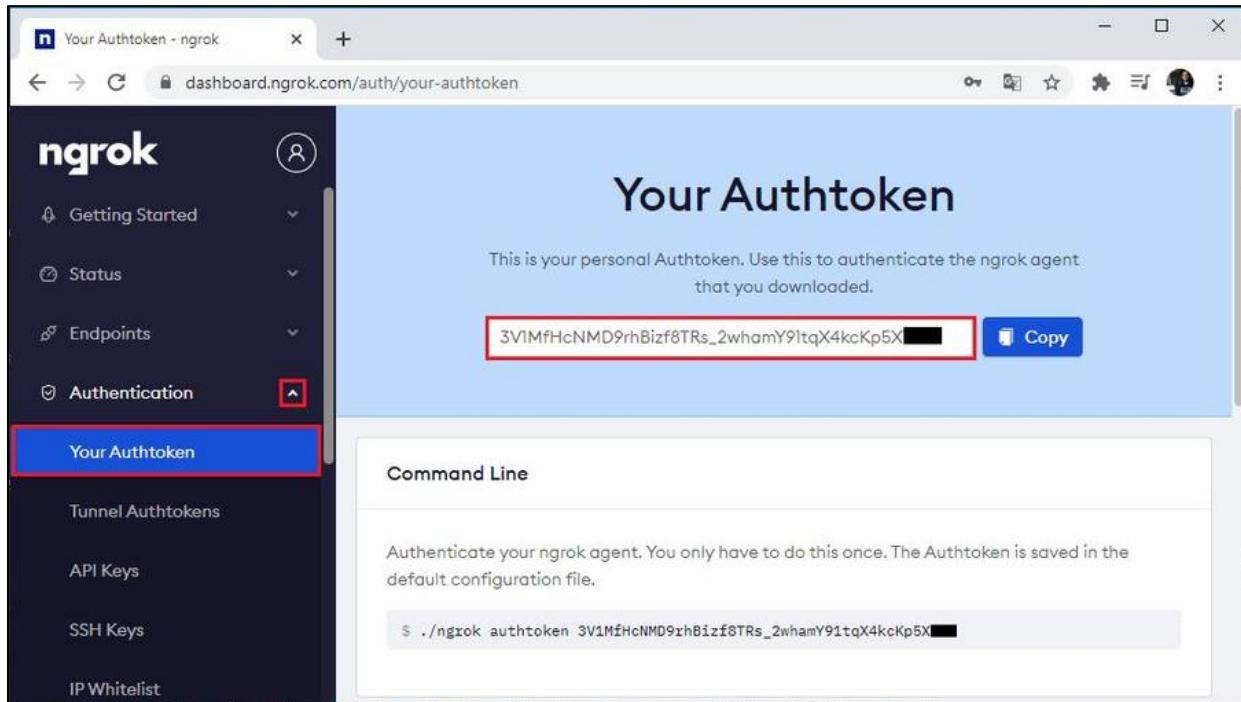
```
$ ./ngrok http 3000
ngrok by @inconshreveable

Session Status: online
Account: Kate Libby (Plan: Pro)
Web Interface: http://127.0.0.1:4040
Forwarding: http://katesapp.ngrok.io -> localhost:3000
Forwarding: https://katesapp.ngrok.io -> localhost:3000
```

Click the blue “**Sign up**” button and enter your details in the fields.

After creating your account, login and go to the “**Authentication**” tab to find your Authtoken.

Copy your unique **Authtoken** to a safe place (you’ll need it later in this Unit).



My Authtoken is:

3V1MfHcNMD9rhBif8TRs_2whamY91tqX4kcKp5X---

Installing ngrok in a different Operating System

If you prefer using ngrok in a different operating system, you can download the installation files for all operating systems at the next link:

- <https://dashboard.ngrok.com/get-started/setup>

The command that makes the ESP web server available from anywhere is the same for all operating systems, but we'll use it on a Raspberry Pi.

Installing ngrok on Raspberry Pi

Having an SSH communication established with your Raspberry Pi:



```
pi@raspberrypi: ~
```

Send the following command to download the latest version of ngrok software:

```
pi@raspberry:~ $ wget http://randomnerdtutorials.com/ngrokARM
```

Unzip the ngrok software using:

```
pi@raspberry:~ $ unzip ngrokARM
```

Running ngrok

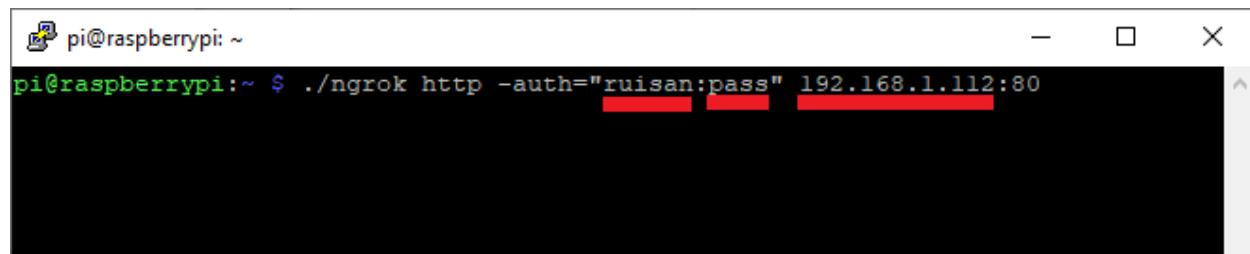
Enter the following command to authenticate your ngrok account (replace the red highlighted text with your own Authtoken):

```
pi@raspberry:~ $ ./ngrok authtoken 3V1MFHcNMD9rhBif8TRs_2whamY91tqX4kcKp5X---
```

In your terminal window, enter the following command and replace the red text with your desired credentials and ESP IP address:

```
pi@raspberry:~ $ ./ngrok http -auth="ruisan:pass" 192.168.1.112:80
```

When you try to access the web server, you'll be asked to enter a username (**ruisan**) and a password (**pass**).



```
pi@raspberrypi: ~
```

```
pi@raspberrypi:~ $ ./ngrok http -auth="ruisan:pass" [REDACTED] 192.168.1.112:80
```

When you run the previous command, a new window shows up:

```
pi@raspberrypi: ~/Desktop/tt
ngrok by @inconshreveable
Session Status          online
Account                 Rui Filipe Fernandes Santos (Plan: Free)
Version                2.3.35
Region                 United States (us)
Web Interface          http://127.0.0.1:4040
Forwarding             http://d11e777593b7.ngrok.io -> http://localhost:1880
Forwarding             https://d11e777593b7.ngrok.io -> http://localhost:1880
Connections            ttl     opn     rtl     rt5      p50      p90
                        0       0      0.00    0.00    0.00    0.00
```

Copy your unique link (it is highlighted in the preceding figure):

<https://d11e777593b7.ngrok.io>

Important: do not use the **http** link, because that link is not encrypted:

<http://d11e777593b7.ngrok.io>

Important: you should always use the **https** link:

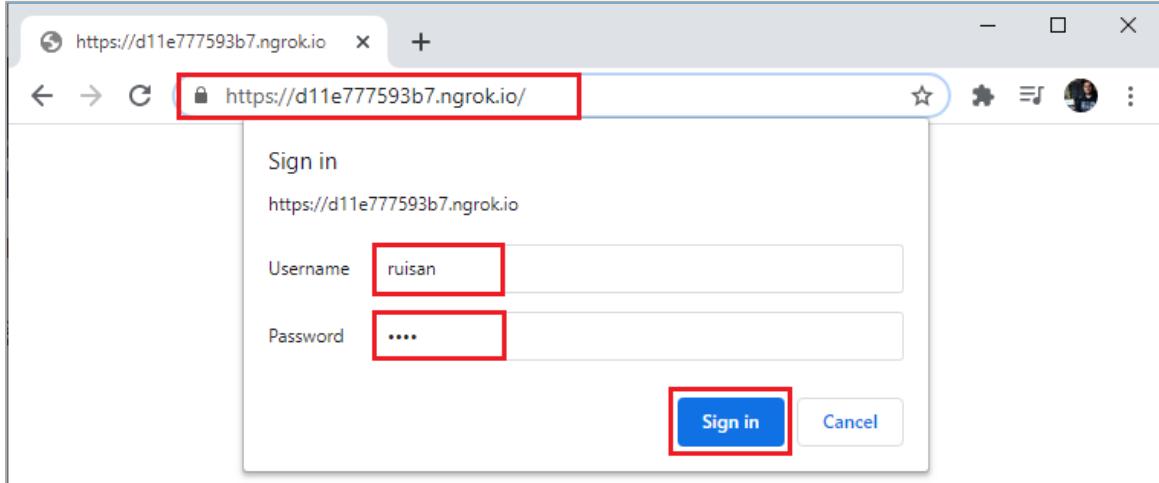
<https://d11e777593b7.ngrok.io>

Accessing Your Web Server from Anywhere

If everything runs smoothly, you can open the ESP web server from any web browser in the world. You just need to enter the URL you've got, as shown below:

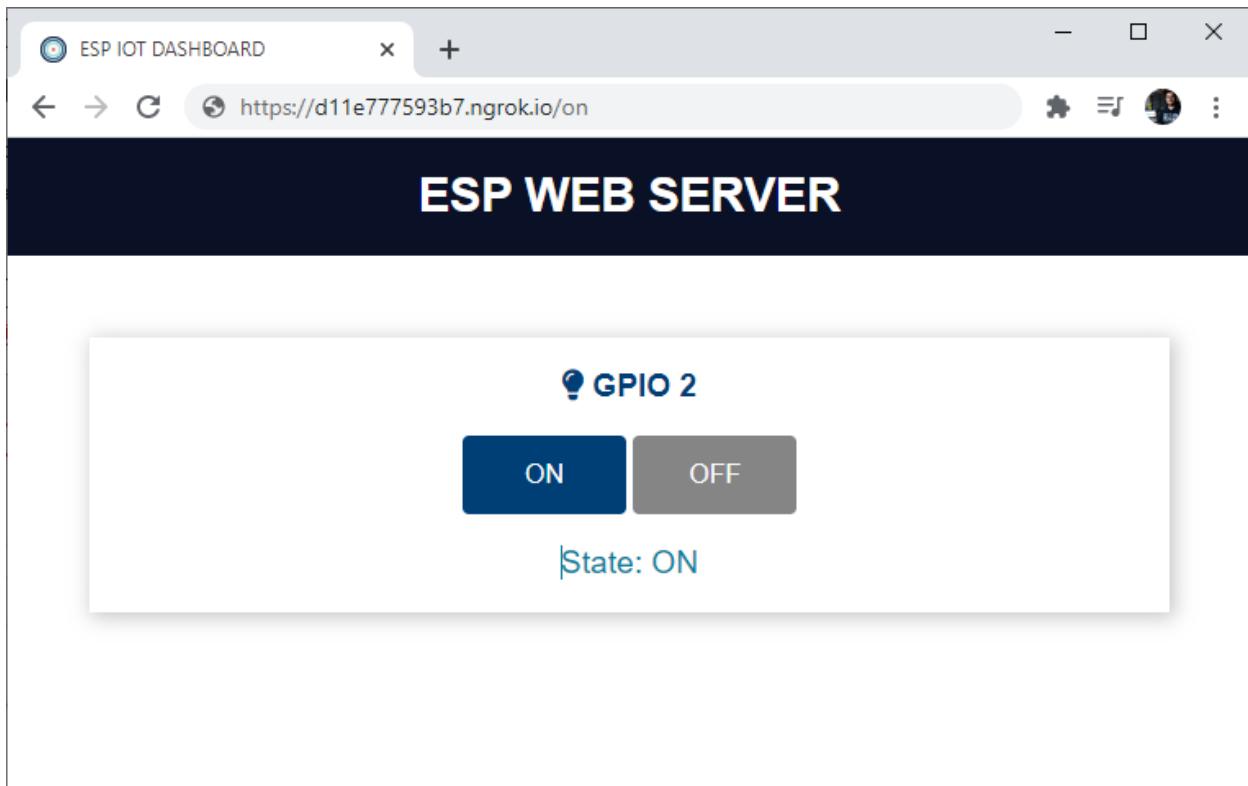
<https://d11e777593b7.ngrok.io/>

You'll be asked to enter a username (**ruisan**) and password (**pass**):



After entering the correct credentials for your tunnel, your ESP web server loads.

Now, you have full control over your ESP from anywhere in the world.



Important: you can close the SSH window, but you can't quit the ngrok software. Otherwise, your tunnel stops. You also need to leave your Raspberry Pi on and running ngrok in order to maintain the tunnel online.

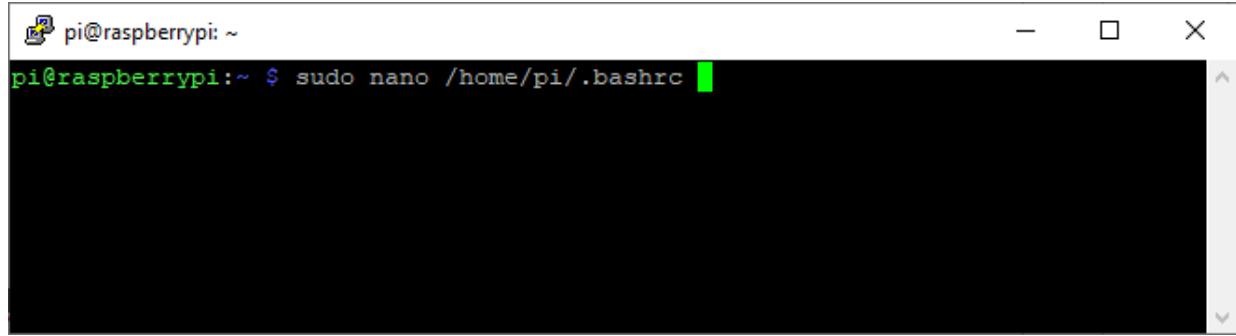
Autostart ngrok on boot

To make ngrok software autostart when the Raspberry Pi first boots, you need to install the screen software that allows you to run commands in the background:

```
pi@raspberry:~ $ sudo apt install screen -y
```

Finally, you need to edit the file `/home/pi/.bashrc` to tell your Raspberry Pi to run ngrok on start. Type the next command in your terminal window:

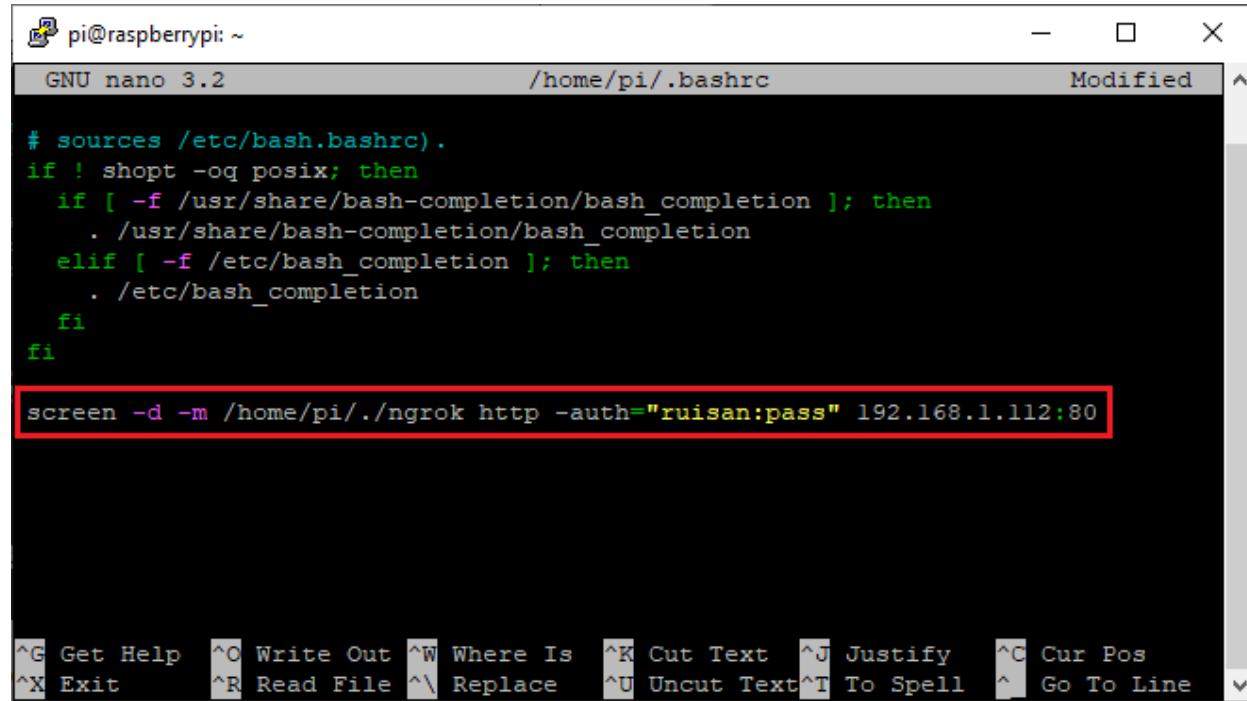
```
pi@raspberry:~ $ sudo nano /home/pi/.bashrc
```



```
pi@raspberrypi: ~
pi@raspberrypi:~ $ sudo nano /home/pi/.bashrc
```

Scroll down to the bottom of the `.bashrc` file and add the following command:

```
screen -d -m /home/pi/./ngrok http -auth="ruisan:pass" 192.168.1.112:80
```



```
pi@raspberrypi: ~
GNU nano 3.2          /home/pi/.bashrc          Modified

# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

screen -d -m /home/pi/./ngrok http -auth="ruisan:pass" 192.168.1.112:80
```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^L Go To Line

Press **Ctrl+X**, followed by **Y** and **Enter** key to save the file. Then, restart your Raspberry Pi to test if it's auto starting ngrok on boot:

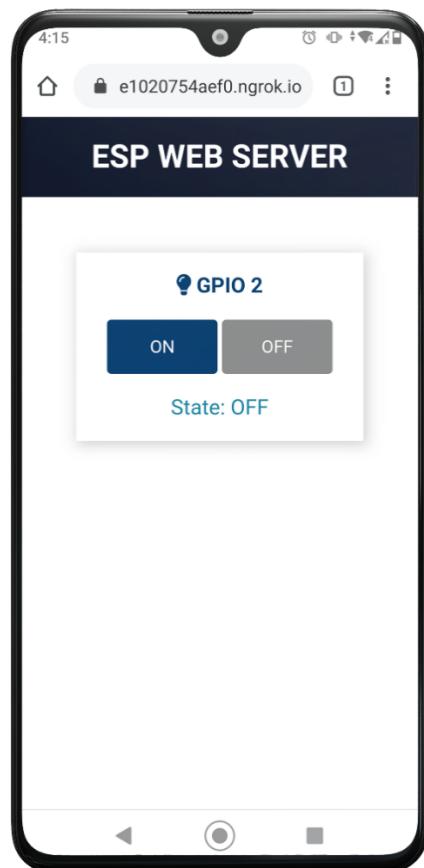
```
pi@raspberry:~ $ sudo reboot
```

Wait a few minutes for your Raspberry Pi to fully start all services. Then, if you go to ngrok Dashboard under the "Tunnels Online" menu, you should see the new URL that you must access to view your web server.

The screenshot shows the ngrok dashboard interface. On the left, there's a sidebar with options: Getting Started, Status (which has a red box around it), Tunnels (which is highlighted with a blue box), Sessions, Endpoints, and Authentication. The main area is titled "Tunnels Online". It includes a search bar labeled "Filter tunnels..." and a table with columns: Region, URL, and Client IP. There are two entries in the table:

Region	URL	Client IP
US	https://f3efd5f6175d.ngrok.io	2001:8a0:e3c1:a000:2600:8b00:3966:
US	http://f3efd5f6175d.ngrok.io	2001:8a0:e3c1:a000:2600:8b00:3966:

When you enter the preceding URL in your browser, you'll be asked to enter your username and password to open your web server. Now, you have control over your boards from anywhere!



Setting the ESP32 and ESP8266 as an Access Point



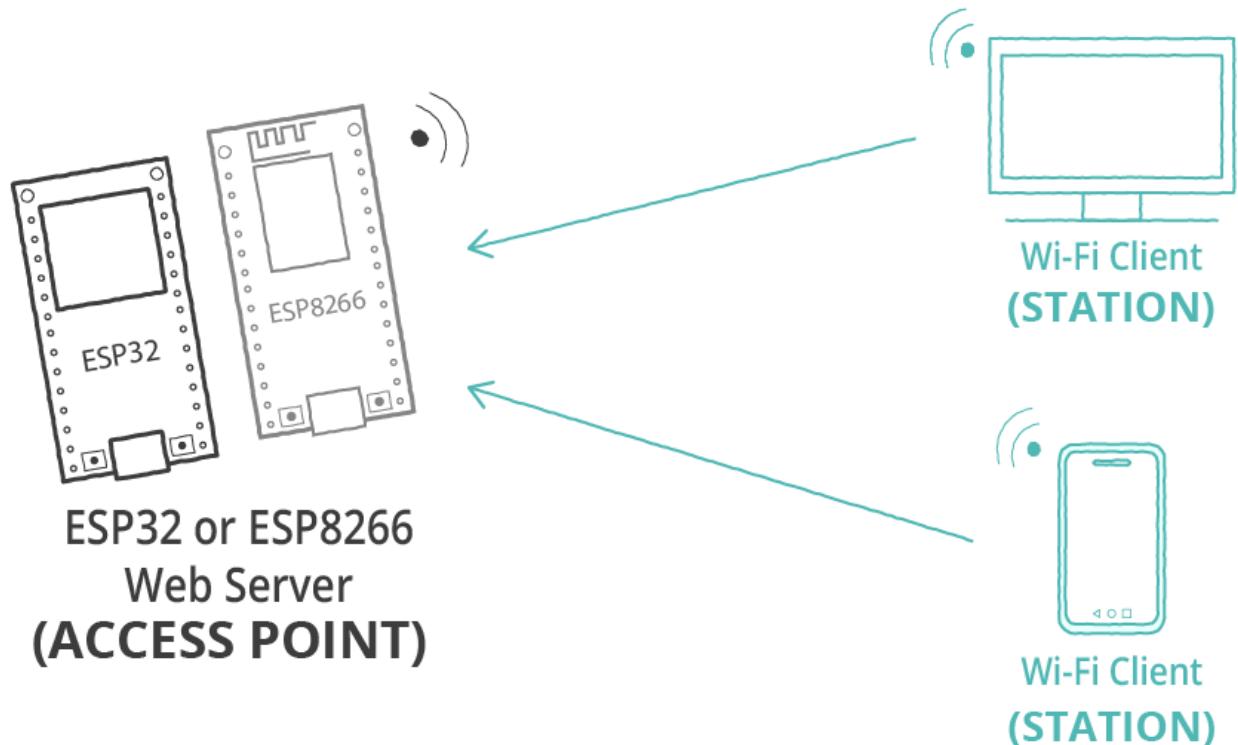
The ESP32 and ESP8266 boards can act as Wi-Fi Station, Access Point or both. In this section, you'll learn how to set the ESP32 and ESP8266 boards as an Access Point.

In all projects presented throughout the eBook , the ESP boards are set as a station. The boards are connected to a wireless router. This way, you can only access the boards through your local network. If you are in a location without access to a wireless router, you can still connect to and control your boards if you set them as Access Points.

Access Point

When you set your ESP boards as an access point, you can be connected to them using any device with Wi-Fi capabilities without having to connect to your router.

When you set the ESP32 or ESP8266 as an access point you create its own Wi-Fi network and nearby Wi-Fi devices (stations) like your smartphone or your computer can connect to it. So, you don't need to be connected to a router to control them.



Because the ESP doesn't connect further to a wired network like your router, it is called soft-AP (soft Access Point). This means that if you try to load libraries or use firmware from the internet, it will not work.

For example, when your ESP is acting as an Access Point, your script isn't able to load icons from the fontawesome website or download the Highcharts Javascript library to build charts. You would have had to do this earlier when connected through a router, storing whatever you need in the ESP filesystem for use later. Unfortunately, some libraries (like Highcharts) are too big to do this.

For you to understand how to set the ESP32/ESP8266 as an access point, we'll use the example from Unit 1.1.

Setting Up the Web Server

Follow the steps below to build the web server. You need to edit the *platformio.ini* file and the *main.cpp* file inside the *src* folder.

platformio.ini file (ESP32)

The *platformio.ini* file for the ESP32 should be like this.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

platformio.ini file (ESP8266)

The *platformio.ini* file for the ESP8266 should be like this.

```
[env:esp12e]
platform = espressif8266
board = esp12e
framework = arduino
monitor_speed = 115200
lib_deps = ESP Async WebServer
```

main.cpp (ESP32)

This section refers to the ESP32. If you're using an ESP8266, skip to the next section.

The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

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

// Replace with desired access point credentials
```

```

const char* ssid      = "ESP32-Access-Point";
const char* password = "123456789";

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

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
  <title>ESP Web Server</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,>
  <style>
    html {
      font-family: Arial;
      text-align: center;
    }
    body {
      max-width: 400px;
      margin: 0px auto;
    }
  </style>
</head>
<body>
  <h1>Hello World!</h1>
  <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
)rawliteral";

void initAP() {
  WiFi.mode(WIFI_AP);
  WiFi.softAP(ssid, password);
}

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

  server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
    request->send(200, "text/html", index_html);
  });

  // Start server
  server.begin();
}

void loop() {
  // put your main code here, to run repeatedly:
}

```

main.cpp (ESP8266)

This section refers to the ESP8266. If you're using an ESP32, go back to the previous section. The code is slightly different for each board.

Open the *main.cpp* file—it's in the *src* folder. Copy the following code to your *main.cpp* file. You can download the complete project folder at the end of the unit.

```
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <ESPAsyncWebServer.h>

// Replace with desired access point credentials
const char* ssid      = "ESP8266-Access-Point";
const char* password = "123456789";

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

const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE html>
<html>
<head>
    <title>ESP Web Server</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" href="data:,">
    <style>
        html {
            font-family: Arial;
            text-align: center;
        }
        body {
            max-width: 400px;
            margin: 0px auto;
        }
    </style>
</head>
<body>
    <h1>Hello World!</h1>
    <p>Congratulations!<br>This is your first Web Server with the ESP.</p>
</body>
</html>
)rawliteral";

void initAP() {
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password);
}
```

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

    server.on("/", HTTP_GET, [] (AsyncWebServerRequest *request){
        request->send(200, "text/html", index_html);
    });

    // Start server
    server.begin();
}

void loop() {
// put your main code here, to run repeatedly:
}
```

How The Code Works

Let's take a look at the code to see how to set up an access point.

Customize the SSID and Password

You need to define an SSID name and a password for the access point. In this example, we're setting the SSID name to `ESP32-Access-Point`. You can modify the name to whatever you want. The password is `123456789`, but you can and should also modify it.

```
// Replace with desired access point credentials
const char* ssid      = "ESP32-Access-Point";
const char* password = "123456789";
```

Setting the ESP as an Access Point

The `initAP()` function initializes the ESP as an access point with the credentials you've defined in the code.

```
void initAP() {
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password);
}
```

The `ssid` and `password` parameters are passed to the `softAP()` method. There are other optional parameters you can pass to the `softAP()` method. Here are all the parameters:

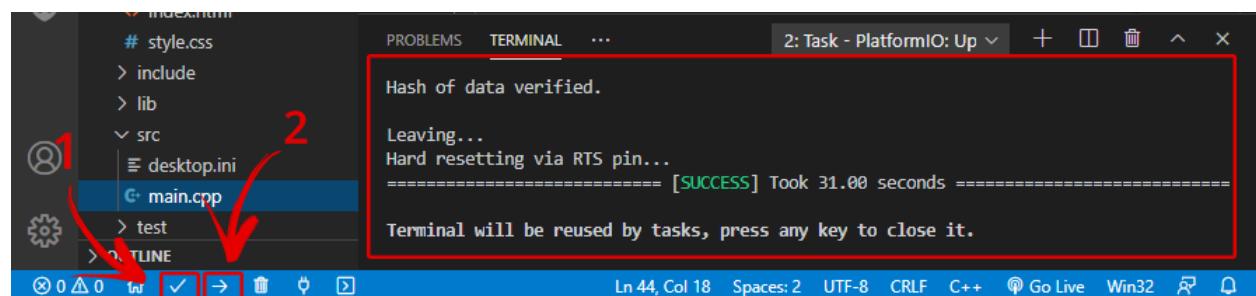
```
softAP(const char* ssid, const char* password, int channel, int ssid_hidden, int max_connection)
```

- `ssid` (defined earlier): maximum of 63 characters;
- `password` (defined earlier): minimum of 8 characters; set to `NULL` if you want the access point to be open
- `channel`: Wi-Fi channel number (1-13)
- `ssid_hidden`: (0 = broadcast SSID, 1 = hide SSID)
- `max_connection`: maximum simultaneous connected clients (1-4)

Then, you just need to call the `initAP()` function in the `setup()` and your web server projects will run in access point mode.

Uploading Code

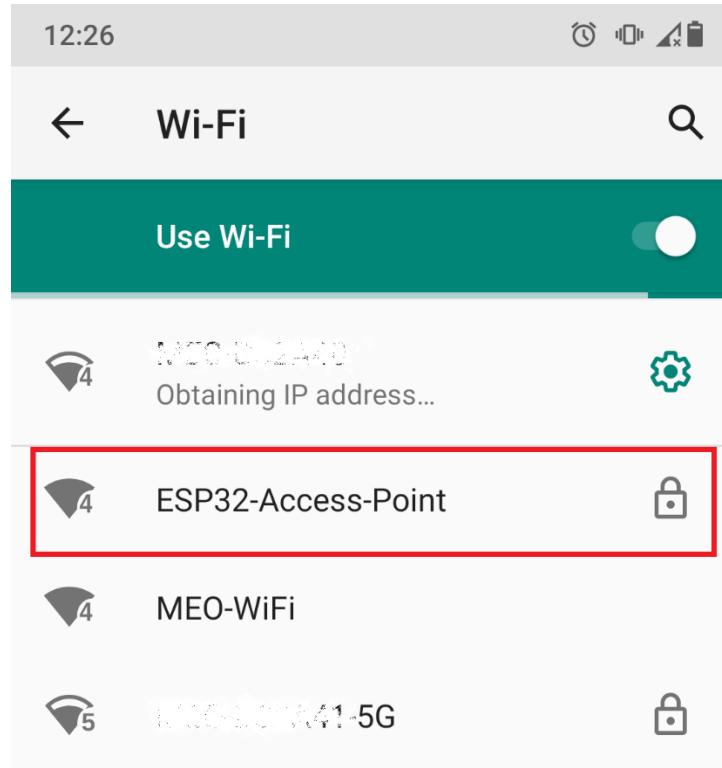
After adding the desired credentials for the access point, save the code. Click on the **Compile** icon and then on the **Upload** icon to upload code to your board.



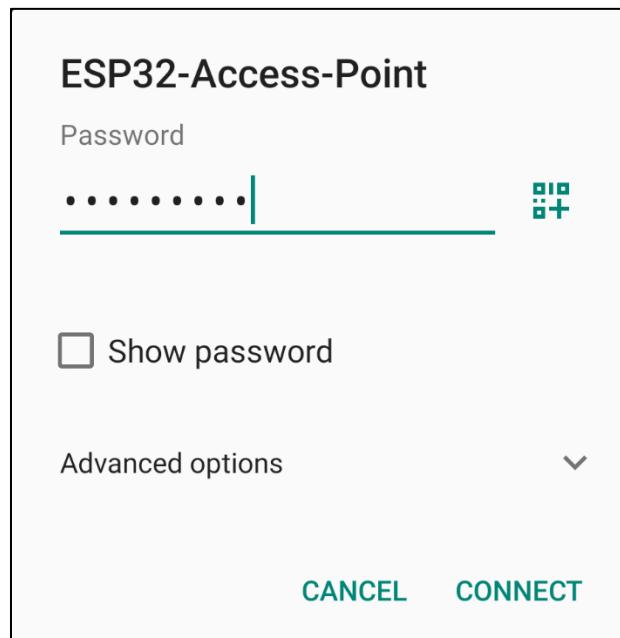
Connecting to the ESP Access Point

After uploading the code, you can connect to the ESP access point to access the web server. You don't need to connect to a router.

On your smartphone, open your Wi-Fi settings and tap the ESP Access Point network:



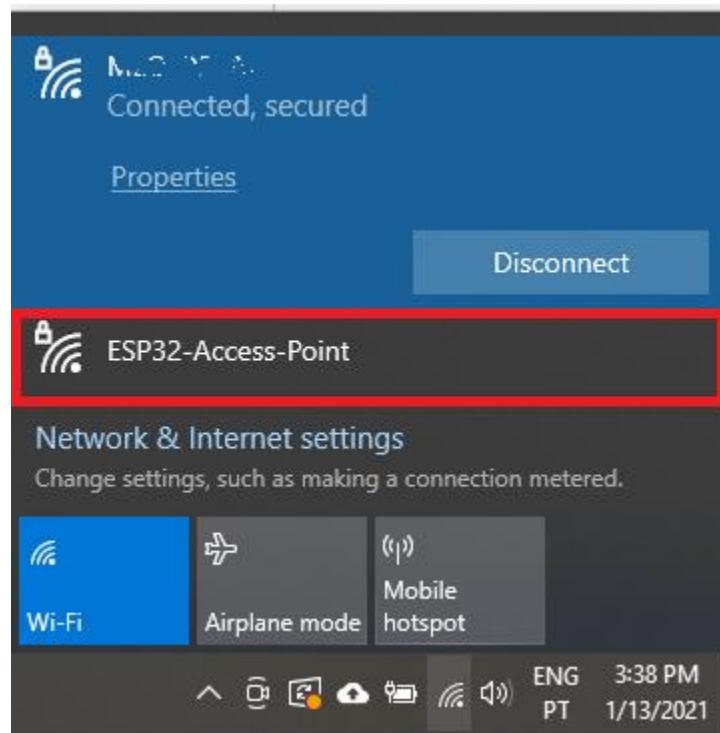
Type the password you've defined earlier in the code.



Open your web browser and type the IP address **192.168.4.1**. You should get access to the Hello World Web Server.



To connect to the access point using your computer, go to the Network and Internet Settings, select the ESP Access Point and insert the password.



And it's done! Now, to access the web server page, you just need to type the IP address **192.168.4.1** in your browser.

Download Project Folder

You can download the complete project folder for this project using the links below.

- [ESP32 Project Folder](#)
- [ESP8266 Project Folder](#)

Wrapping Up

In this tutorial, you've learned how to set the ESP32/ESP8266 as an access point for your web server projects. When the ESP is set as an access point, devices with Wi-Fi capabilities like your smartphone can connect directly to the ESP without the need to connect to a router.

ESP32/ESP8266 Static IP Address



This section shows how to set a static/fixed IP address for your ESP32/ESP8266 boards. If you're running a web server or Wi-Fi client with the ESP and every time you restart your board, it has a new IP address, you can follow this Unit to assign a static/fixed IP address.

Setting Static IP Address

Before the `setup()` and `loop()` functions, define the following variables with your own static IP address and corresponding gateway IP address.

```
// Set your Static IP address
IPAddress local_IP(192, 168, 1, 184);

// Set your Gateway IP address
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 0, 0);

IPAddress primaryDNS(8, 8, 8, 8);    //optional
IPAddress secondaryDNS(8, 8, 4, 4); //optional
```

By default, the previous snippet assigns the IP address 192.168.1.184 that works in the 192.168.1.1 gateway.

In the `setup()`, you need to call the `WiFi.config()` method to assign the configurations to your ESP board (before starting Wi-Fi).

```
if(!WiFi.config(local_IP, gateway, subnet, primaryDNS, secondaryDNS)) {  
    Serial.println("STA Failed to configure");  
}  
WiFi.begin(ssid, password);
```

The `primaryDNS` and `secondaryDNS` parameters are optional and you can remove them.

And that's it. You just need to follow the previous instructions to set a static IP address to your ESP32 or ESP8266 boards.

CONGRATULATIONS

For Completing This eBook

Congratulations for Completing this eBook!

If you followed all the projects presented in this eBook, now you should be able to build your web server projects with the ESP32 and ESP8266 boards. You know how to control the ESP outputs and monitor sensors through the web pages served by your web server. You also know how to create your own web pages using HTML, CSS, and JavaScript. Here's a summary of what you've learned:

- Install and use VS Code Text Editor to write your HTML, CSS and JavaScript files;
- Use VS Code with PlatformIO IDE extension to program the ESP32/ESP8266;
- Upload files to the ESP filesystem;
- Create HTML, CSS, and JavaScript files to build web pages to interface with the ESP boards;
- Use different client-server communication protocols: HTTP polling, WebSocket, and Server-Sent Events.
- Use different HTML elements to control the ESP outputs: buttons, toggle switches, and sliders;
- Display sensor readings on the web pages in paragraphs, tables, and charts;
- Add OTA (over-the-air) capabilities to your projects;
- And much more...

We hope you had fun following this eBook and you've learned something new!

If you have something you would like to share (your projects, suggestions, feedback) use the [Private Forum](#) or [Facebook group](#).

Good luck with all your projects,

Rui Santos and Sara Santos

Other RNT Courses/eBooks

[Random Nerd Tutorials](#) is an online resource with electronics projects, tutorials, and reviews. Currently, Random Nerd Tutorials has more than [250 free blog posts](#) with complete tutorials using open-source hardware that anyone can read, remix and apply to their projects.

To keep free tutorials coming, there's also paid content or what we like to call "premium content". To support Random Nerd Tutorials, you can [download premium content here](#). If you enjoyed this eBook, make sure you [check all our courses and resources](#).

Here's a list of all the courses/eBooks available to download at Random Nerd Tutorials:

- [Learn ESP32 with Arduino IDE](#) (Bestseller)
- [Build ESP32-CAM Projects using Arduino IDE](#)
- [Home Automation Using ESP8266](#)
- [MicroPython Programming with ESP32/ESP8266](#)
- [20 Easy Raspberry Pi Projects](#) (available on Amazon)
- [Build a Home Automation System for \\$100](#)
- [Arduino Step-by-step Projects Course](#)
- [Android Apps for Arduino with MIT App Inventor 2](#)
- [Electronics For Beginners eBook](#)