A Beginner's Guide to the ESP8266

Pieter P, 08-03-2017

Some time ago, I wrote a Beginner's Guide to Arduino that seems to be very popular, so I decided to create a follow-up: **A Beginner's Guide to the ESP8266**. That's right, a tutorial on how to use the world's most popular $3 Wi-Fi board.

This is going to be a very in-depth tutorial, covering some networking concepts as well. If you're a beginner, and just want to go straight to the more exciting Wi-Fi part, feel free to do so, I included short *TL;DR's* in the longer, more technical parts.

A short overview of what I'll cover in this article:

1. **What is an ESP8266?** A short overview of what an ESP8266 is, and what you can do with it
2. **Deciding on what board to buy**: There's loads of different ESP8266 available these days, finding the one that's best for you can be hard
3. **Installing the software**: you need to install some software to program the ESP8266, and maybe a USB driver
4. **Setting up the hardware**: some modules and boards need some external components
5. **The ESP8266 as a microcontroller**: the ESP8266 can be used as a normal microcontroller, just like an Arduino
6. **Network protocols**: Before we start using the Wi-Fi capabilities of the ESP8266, I'll teach you some of the network protocols involved
7. **Setting up a Wi-Fi connection**: That's probably why you're reading this, right?
8. **Name resolution**: Find the ESP8266 on your local network using mDNS
9. **Setting up a simple web server**: This enables you to add web pages to the ESP8266, and browse them from your computer or phone
10. **Setting up an advanced web server**: a more advanced server with a real file system that allows you to upload new files over Wi-Fi
11. **OTA - uploading programs over Wi-Fi**: You don't have to upload programs over USB, you can use Wi-Fi instead
12. **Wirelessly controlling your RGB lighting**: Change the color of your LED strips using your phone or computer
13. **Getting the time**: Connect to a time server using NTP and sync the ESP's clock
14. **Monitoring sensors**: log the temperature in your living room, save it in flash memory and show it in a fancy graph in your browser
15. **Getting email notifications**: Turn on a notification light when you've got unread emails
16. **Advanced features**: use DNS, captive portals, Wi-Fi connector libraries, OSC ...

This guide expects some basic knowledge of microcontrollers like the Arduino. If that's something you're not already familiar with, I'd recommend you to read my Beginner's Guide to Arduino first, it covers a lot of the basics that I won't go into in this article.
I really want to focus on the ESP8266-specific things, like Wi-Fi and other network protocols, the ESP's hardware, software, IoT, etc ...

What is an ESP8266?

The ESP8266 is a System on a Chip (SoC), manufactured by the Chinese company Espressif. It consists of a Tensilica L106 32-bit **micro controller** unit (MCU) and a **Wi-Fi transceiver**. It has **11 GPIO pins*** (General Purpose Input/Output pins), and an **analog input** as well. This means that you can program it like any normal Arduino or other microcontroller. And on top of that, you get Wi-Fi communication, so you can use it to connect to your Wi-Fi network, connect to the Internet, host a web server with real web pages, let your smartphone connect to it, etc ... The possibilities are endless! It's no wonder that this chip has become the most popular IOT device available.

There are many different modules available, standalone modules like the ESP-## series by AI Thinker, or complete development boards like the NodeMCU DevKit or the WeMos D1. Different boards may have different pins broken out, have different Wi-Fi antennas, or a different amount of flash memory on board.

(*) The ESP8266 chip itself has 17 GPIO pins, but 6 of these pins (6-11) are used for communication with the on-board flash memory chip.

Programming

There are different ways to program the ESP8266, but I'll only cover the method using the Arduino IDE. This is really easy for beginners, and it's a very familiar environment if you've used Arduino boards before.
Just keep in mind that it's not limited to this option: there's also an official SDK available to program it in real C, this is very useful if you want to optimize your code or do some advanced tricks that aren't supported by the Arduino IDE. Another possibility is to flash it with a LUA interpreter, so you can upload and run LUA scripts. Or maybe you're more familiar with Python? Then you should check out the MicroPython firmware to interpret MicroPython scripts. I'm sure there's other languages available as well, so just do a quick Google search and write your code in the language of your choice.

Requirements

You'll need a couple of things in order to follow this guide:

- An ESP8266 board
- A computer that can run the Arduino IDE (Windows, Mac or Linux)
- A USB-to-Serial converter, it is very important that you use a **3.3V** model*
- A USB cable
- A 3.3V power supply or voltage regulator*
- A Wi-Fi network to connect to

(*) Your board may already include these. More information can be found in the next chapter.

Hardware

Deciding on what board to buy

ESP8266 is just the name of the chip, many companies have designed their own boards that use this chip, so there are many different ESP8266 boards on the market. If you don't know the difference between all these different models, you might have a hard time deciding on what board to buy.

The easiest (and fastest) way to get an ESP8266 board is to buy one from a well-known electronics shop like Adafruit or SparkFun, but if you want it cheap, you should check out Ebay or other sites where you can order them directly from China.

Development boards

Some boards have all kinds of features on-board to help developing ESP8266 hardware and software: for example, a USB to Serial converter for programming, a 3.3V regulator for power, on-board LEDs for debugging, a voltage divider to scale the analog input ...
If you're a beginner, I would definitely recommend a development board. It's easier to get started if you don't have to worry about all these things.

Bare-bones AI Thinker boards

If you want to add an ESP8266 to a small project, or if you want a cheaper* board, you might want to buy a board that doesn't have these features. In that case, you can buy one of the many ESP-## modules developed by AI Thinker. They contain just the ESP8266 and the necessary components to run it.
To program the board, you'll need an external USB-to-Serial converter.
With some modules, you get an on-board antenna (PCB or ceramic) and an LED, some boards have just an antenna connector, or no LEDs at all. They also differ in physical size, and flash memory size. An important thing to notice, is that some boards do not break out all GPIO pins. For example, the ESP-01 only has 2 I/O pins available (apart from the TX and RX pins), while other modules like the ESP-07 or ESP-12 break out all available I/O pins.

(*) The board itself is cheaper, but you'll have to spend more on external parts.

Overview

Here's a table with some of the most popular ESP8266 development boards and their features:

Board	GPIO	3.3V Vreg	USB-to-Serial	Auto-Reset	Auto-Program	Flash	ADC range	Extra
SparkFun ESP8266 Thing	11	+	-	+	±*	512KB (4Mb)	0-1V	Battery charger, crypto element, temperature sensor, light sensor
SparkFun ESP8266 Thing - Dev Board	11	+	+	+	+	512KB (4Mb)	0-1V	
Node MCU	11	+	+	+	+	4MB (32Mb)	0-3.3V	
Adafruit Feather HUZZAH with ESP8266	11	+	+	+	+	4MB (32Mb)	0-1V	Battery charger
Adafruit HUZZAH ESP8266 Breakout	11	+	-	-	-	4MB (32Mb)	0-1V	5V-tolerant RX and Reset pins
ESP-##	4 - 11	-	-	-	-	512KB (4Mb) – 4MB (32Mb)	0-1V	Small and cheap

You can find the full list of ESP-## modules here.

As you can see, both the Node MCU and the Adafruit Feather HUZZAH are solid choices.

(*) When auto-program on the SparkFun ESP8266 Thing is enabled, you can't use the Serial Monitor.

Getting the hardware ready

There are two main categories of ESP8266 boards: development boards with a USB interface (USB-to-Serial convertor) on-board, and boards without a USB connection.

Development boards with a USB interface

For example: NodeMCU, SparkFun ESP8266 Thing - Dev Board, SparkFun Blynk Board, Adafruit Feather HUZZAH with ESP8266 Wi-Fi ...

These boards will show up in Device manager (Windows) or in lsusb (Linux) as soon as you plug them in. They have a 3.3V regulator on-board, and can be programmed over USB directly, so you don't need any external components to get it working.
The only thing you may need to do, is solder on some headers.

Bare-bones boards and boards without a USB interface

This category has 2 sub-categories: boards with a 3.3V regulator on-board, and boards with just the ESP8266 and a flash memory chip, without 3.3V regulator. If your board doesn't have a 5V to 3.3V regulator, buy one separately. You could use an LM1117-3.3 for example. The on-board 3.3V regulator of most Arduino boards is not powerful enough to power the ESP.

To program the board, you'll need a USB-to-Serial converter. The FTDI FT232RL is quite popular, because it can switch between 5V and 3.3V. **It is essential that the USB-to-Serial converter you buy operates at 3.3V. If you buy a 5V model, you will damage the ESP8266.**

Connecting the USB-to-Serial converter

1. Connect the ground (GND) of the USB-to-Serial converter to the ground of the ESP8266.
2. Connect the RX-pin of the USB-to-Serial converter to the TXD pin of the ESP8266. (On some boards, it's labelled TX instead of TXD, but it's the same pin.)
3. Connect the TX-pin of the USB-to-Serial converter to the RXD pin of the ESP8266. (On some boards, it's labelled RX instead of RXD, but it's the same pin.)
4. If your ESP8266 board has a DTR pin, connect it to the DTR pin of the USB-to-Serial converter. This enables auto-reset when uploading a sketch, more on that later.

Enabling the chip

If you're using a bare-bone ESP-## board by AI Thinker, you have to add some resistors to turn on the ESP8266, and to select the right boot mode.

1. Enable the chip by connecting the CH_PD (Chip Power Down, sometimes labeled CH_EN or chip enable) pin to V_{CC} through a 10KΩ resistor.
2. Disable SD-card boot by connecting GPIO15 to ground through a 10KΩ resistor.
3. Select normal boot mode by connecting GPIO0 to V_{CC} through a 10KΩ resistor.
4. Prevent random resets by connecting the RST (reset) pin to V_{CC} through a 10KΩ resistor.
5. Make sure you don't have anything connected to GPIO2 (more information in the next chapter).

Adding reset and program buttons

If your ESP8266 board doesn't have a reset button, you could add one by connecting a push button to between the RST pin and ground.

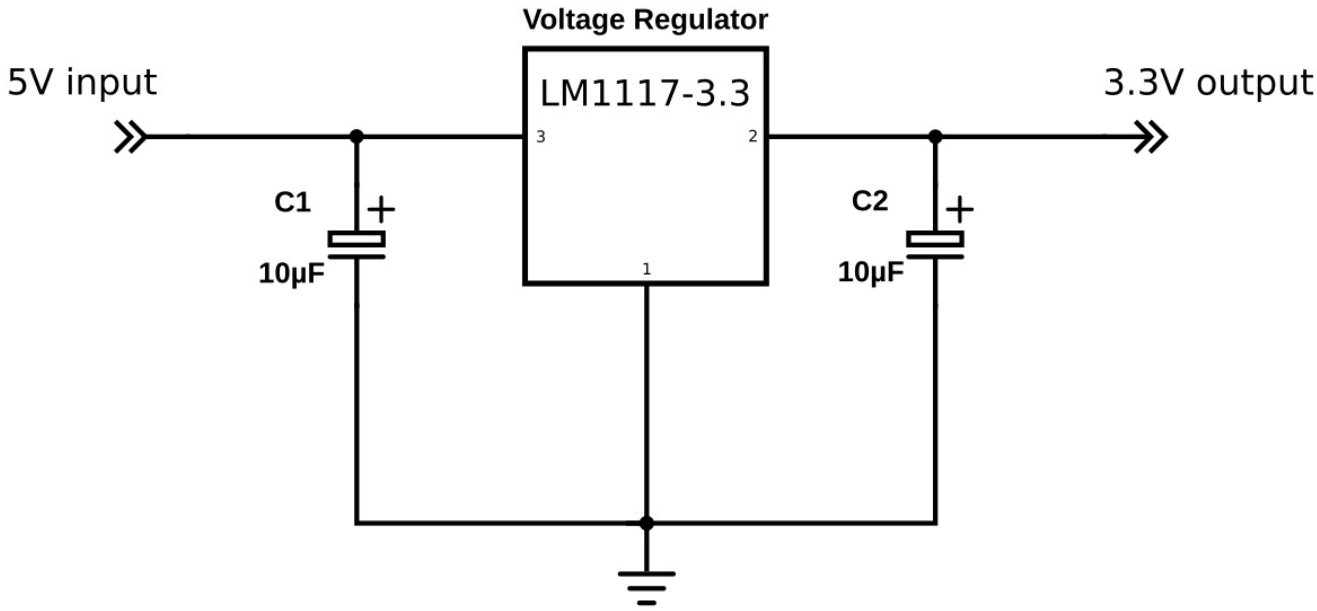
To put the chip into programming mode, you have to pull GPIO0 low during startup. That's why we also need a program button. Because it's possible to use GPIO0 as an output, we can't directly short it to ground, that could damage the chip. To prevent this, connect 470Ω resistor in series with the switch. It's important that this resistance is low enough, otherwise, it will be pulled high by the 10KΩ resistor we added in the previous paragraph.

Connecting the power supply

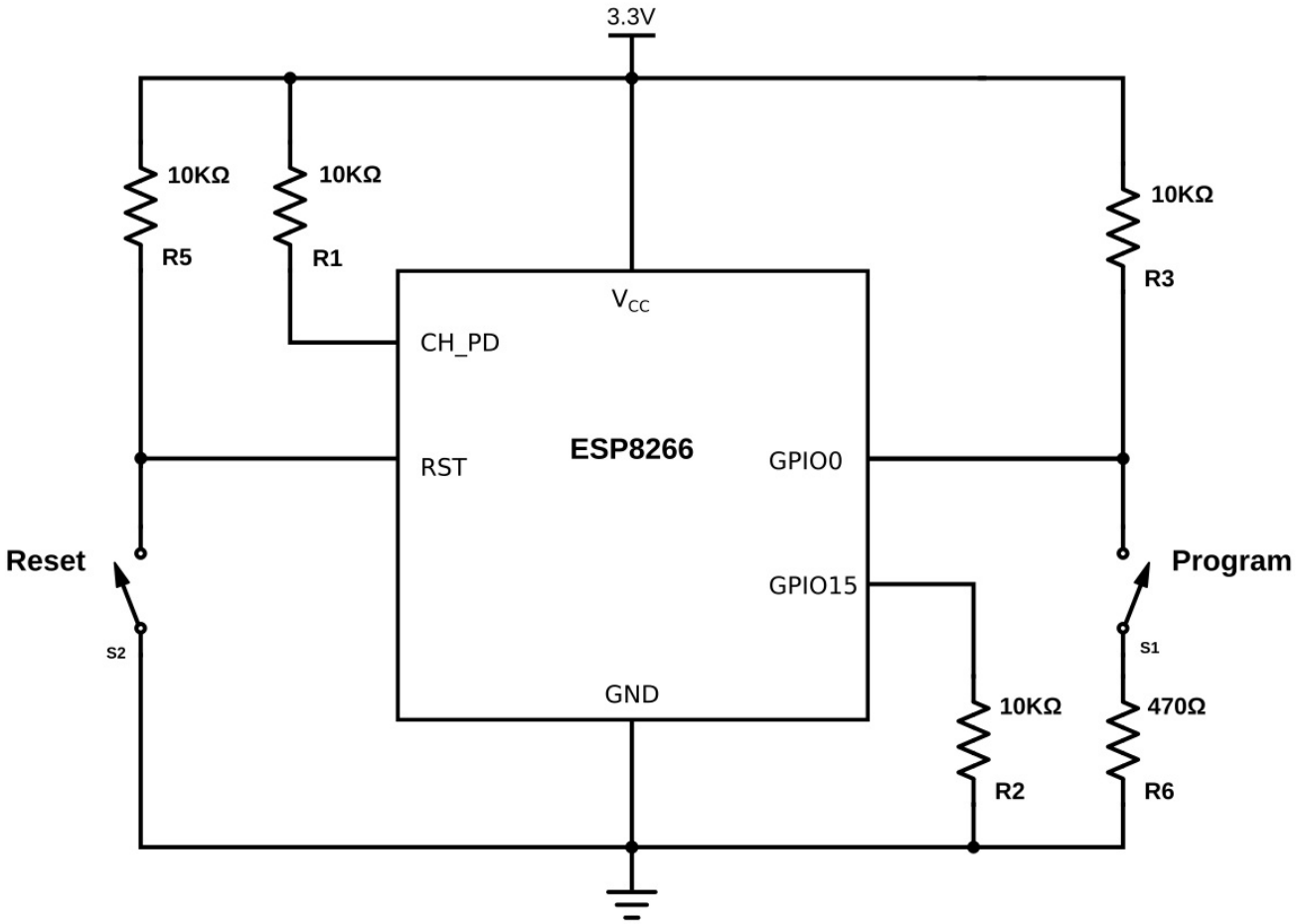
If the ESP8266 module you have doesn't have a 3.3V voltage regulator on board, you have to add one externally. You could use an LM1117-3.3 for example.

1. Connect the first pin of the regulator to ground.
2. Place a 10µF capacitor between pin 2 (V_{out}) and ground. Watch the polarity!
3. Place a 10µF capacitor between pin 3 (V_{in}) and ground.
4. Connect pin 2 to the 3.3V or $V_{CC}$of the ESP8266.
5. Connect pin 3 to a 5V power source, a USB port, for example.

ESP8266 3.3V Voltage Regulator



ESP8266 Wiring Diagram



Before you begin ...

There's a few things you have to look out for when using an ESP8266: The most important thing is that it runs at 3.3V, so if you connect it to a 5V power supply, you'll kill it. Unlike some 3.3V Arduino or Teensy boards, **the ESP8266's I/O pins are not 5V tolerant**, so if you use a 5V USB-to-Serial converter, or 5V sensors etc. you'll blow it up.
A second thing to keep in mind is that the ESP8266 can only source or sink **12mA per output pin**, compared to 20-40mA for most Arduinos.
The ESP8266 has one **analog to digital converter**, but it has a strange voltage range: **0 - 1V**, voltages above 1V might damage the board.

One last thing to keep in mind is that the ESP8266 has to share the system resources and CPU time between your sketch and the Wi-Fi driver. Also, features like PWM, interrupts or I²C are emulated in software, most Arduinos on the other hand, have dedicated hardware parts for these tasks.
For most applications however, this is not too much of an issue.

Software

Installation of the required software

The first step is to download and install the Arduino IDE. I explained this in A Beginner's Guide to Arduino. (As of February 7th 2017, the latest stable version of the IDE is 1.8.1.)

To program the ESP8266, you'll need a plugin for the Arduino IDE, it can be downloaded from GitHub manually, but it is easier to just add the URL in the Arduino IDE:

1. Open the Arduino IDE.
2. Go to File > Preferences.
3. Paste the URL http://arduino.esp8266.com/stable/package_esp8266com_index.json into the *Additional Board Manager URLs* field.
   (You can add multiple URLs, separating them with commas.)
4. Go to Tools > Board > Board Manager and search for 'esp8266'. Select the newest version, and click install. (As of February 7th 2017, the latest stable version is 2.3.0.)

You can check out the official install guide here.

Drivers

If you are using a board with the CH340(G) USB-to-Serial chip, like the NodeMCU, you'll probably have to install the USB drivers for it.
They can be found on GitHub.
If you are using a board with the CP2104 USB-to-Serial chip, like the Adafruit Feather HUZZAH board, you'll probably have to install USB drivers as well. You can find them on the Silicon Labs website.
Boards with an FTDI chip should work right out of the box, without the need of installing any drivers.

Python

If you want to use Over The Air updates on Windows, you have to install Python 2.7. You can download it from python.org. During the installation, you have to select the option to add Python to your path. If you don't do this, the Arduino IDE won't be able to find the Python executable.

Examples

You can find all examples used in this article on my GitHub. Just download it as a .ZIP file, unzip it to a convenient location, and you're good to go!

The ESP8266 as a microcontroller - Hardware

While the ESP8266 is often used as a 'dumb' Serial-to-WiFi bridge, it's a very powerful microcontroller on its own. In this chapter, we'll look at the non-Wi-Fi specific functions of the ESP8266.

Digital I/O

Just like a normal Arduino, the ESP8266 has digital input/output pins (I/O or GPIO, General Purpose Input/Output pins). As the name implies, they can be used as digital inputs to read a digital voltage, or as digital outputs to output either 0V (sink current) or 3.3V (source current).

Voltage and current restrictions

The ESP8266 is a 3.3V microcontroller, so its I/O operates at 3.3V as well. The pins are **not 5V tolerant, applying more than 3.6V on any pin will kill the chip.**

The maximum current that can be drawn from a single GPIO pin is **12mA**.

Usable pins

The ESP8266 has 17 GPIO pins (0-16), however, you can only use 11 of them, because 6 pins (GPIO 6 - 11) are used to connect the flash memory chip. This is the small 8-legged chip right next to the ESP8266. If you try to use one of these pins, you might crash your program.

GPIO 1 and 3 are used as TX and RX of the hardware Serial port (UART), so in most cases, you can't use them as normal I/O while sending/receiving serial data.

Boot modes

As mentioned in the previous chapter, some I/O pins have a special function during boot: They select 1 of 3 boot modes:

GPIO15	GPIO0	GPIO2	Mode
0V	0V	3.3V	Uart Bootloader
0V	3.3V	3.3V	Boot sketch (SPI flash)
3.3V	x	x	SDIO mode (not used for Arduino)

Note: you don't have to add an external pull-up resistor to GPIO2, the internal one is enabled at boot.

We made sure that these conditions are met by adding external resistors in the previous chapter, or the board manufacturer of your board added them for you. This has some implications, however:

- GPIO15 is always pulled low, so you can't use the internal pull-up resistor. You have to keep this in mind when using GPIO15 as an input to read a switch or connect it to a device with an open-collector (or open-drain) output, like I²C.
- GPIO0 is pulled high during normal operation, so you can't use it as a Hi-Z input.
- GPIO2 can't be low at boot, so you can't connect a switch to it.

Internal pull-up/-down resistors

GPIO 0-15 all have a built-in pull-up resistor, just like in an Arduino. GPIO16 has a built-in pull-down resistor.

PWM

Unlike most Atmel chips (Arduino), the ESP8266 doesn't support hardware PWM, however, software PWM is supported on all digital pins. The default PWM range is 10-bits @ 1kHz, but this can be changed (up to >14-bit@1kHz).

Analog input

The ESP8266 has a single analog input, with an input range of 0 - 1.0V. If you supply 3.3V, for example, you will damage the chip. Some boards like the NodeMCU have an on-board resistive voltage divider, to get an easier 0 - 3.3V range. You could also just use a trimpot as a voltage divider.

The ADC (analog to digital converter) has a resolution of 10 bits.

Communication

Serial

The ESP8266 has two hardware UARTS (Serial ports):
UART0 on pins 1 and 3 (TX0 and RX0 resp.), and UART1 on pins 2 and 8 (TX1 and RX1 resp.), however, GPIO8 is used to connect the flash chip. This means that UART1 can only transmit data.

UART0 also has hardware flow control on pins 15 and 13 (RTS0 and CTS0 resp.). These two pins can also be used as alternative TX0 and RX0 pins.

I²C

The ESP doesn't have a hardware TWI (Two Wire Interface), but it is implemented in software. This means that you can use pretty much any two digital pins. By default, the I²C library uses pin 4 as SDA and pin 5 as SCL. (The data sheet specifies GPIO2 as SDA and GPIO14 as SCL.) The maximum speed is approximately 450kHz.

SPI

The ESP8266 has one SPI connection available to the user, referred to as HSPI. It uses GPIO14 as CLK, 12 as MISO, 13 as MOSI and 15 as Slave Select (SS). It can be used in both Slave and Master mode (in software).

GPIO overview

GPIO	Function	State	Restrictions
0	Boot mode select	3.3V	No Hi-Z
1	TX0	-	Not usable during Serial transmission
2	Boot mode select TX1	3.3V (boot only)	Don't connect to ground at boot time Sends debug data at boot time
3	RX0	-	Not usable during Serial transmission
4	SDA (I²C)	-	-
5	SCL (I²C)	-	-
6 - 11	Flash connection	x	Not usable, and not broken out
12	MISO (SPI)	-	-
13	MOSI (SPI)	-	-
14	SCK (SPI)	-	-
15	SS (SPI)	0V	Pull-up resistor not usable
16	Wake up from sleep	-	No pull-up resistor, but pull-down instead Should be connected to RST to wake up

The ESP8266 as a microcontroller - Software

Most of the microcontroller functionality of the ESP uses exactly the same syntax as a normal Arduino, making it really easy to get started.

Digital I/O

Just like with a regular Arduino, you can set the function of a pin using `pinMode(pin, mode);` where `pin` is the GPIO number*, and `mode` can be either `INPUT`, which is the default, `OUTPUT`, or `INPUT_PULLUP` to enable the built-in pull-up resistors for GPIO 0-15. To enable the pull-down resistor for GPIO16, you have to use `INPUT_PULLDOWN_16`.

(*) NodeMCU uses a different pin mapping, read more here. To address a NodeMCU pin, e.g. pin 5, use D5: for instance: `pinMode(D5, OUTPUT);`

To set an output pin high (3.3V) or low (0V), use `digitalWrite(pin, value);` where `pin` is the digital pin, and `value` either 1 or 0 (or `HIGH` and `LOW`).

To read an input, use `digitalRead(pin);`

To enable PWM on a certain pin, use `analogWrite(pin, value);` where `pin` is the digital pin, and `value` a number between 0 and 1023.

You can change the range (bit depth) of the PWM output by using `analogWriteRange(new_range);`

The frequency can be changed by using `analogWriteFreq(new_frequency);`. `new_frequency` should be between 100 and 1000Hz.

Analog input

Just like on an Arduino, you can use `analogRead(A0)` to get the analog voltage on the analog input. (0 = 0V, 1023 = 1.0V).

The ESP can also use the ADC to measure the supply voltage (V_{CC}). To do this, include `ADC_MODE(ADC_VCC);` at the top of your sketch, and use `ESP.getVcc();` to actually get the voltage.
If you use it to read the supply voltage, you can't connect anything else to the analog pin.

Communication

Serial communication

To use UART0 (TX = GPIO1, RX = GPIO3), you can use the `Serial` object, just like on an Arduino: `Serial.begin(baud)`.

To use the alternative pins (TX = GPIO15, RX = GPIO13), use `Serial.swap()` after `Serial.begin`.

To use UART1 (TX = GPIO2), use the `Serial1` object.

All Arduino Stream functions, like `read, write, print, println, ...` are supported as well.

I²C and SPI

You can just use the default Arduino library syntax, like you normally would.

Sharing CPU time with the RF part

One thing to keep in mind while writing programs for the ESP8266 is that your sketch has to share resources (CPU time and memory) with the Wi-Fi- and TCP-stacks (the software that runs in the background and handles all Wi-Fi and IP connections).
If your code takes too long to execute, and don't let the TCP stacks do their thing, it might crash, or you could lose data. It's best to keep the execution time of you loop under a couple of hundreds of milliseconds.

Every time the main loop is repeated, your sketch yields to the Wi-Fi and TCP to handle all Wi-Fi and TCP requests.

If your loop takes longer than this, you will have to explicitly give CPU time to the Wi-Fi/TCP stacks, by using including `delay(0);` or `yield();`. If you don't, network communication won't work as expected, and if it's longer than 3 seconds, the soft WDT (Watch Dog Timer) will reset the ESP. If the soft WDT is disabled, after a little over 8 seconds, the hardware WDT will reset the chip.

From a microcontroller's perspective however, 3 seconds is a very long time (240 million clockcycles), so unless you do some extremely heavy number crunching, or sending extremely long strings over Serial, you won't be affected by this. Just keep in mind that you add the `yield();` inside your `for` or `while` loops that could take longer than, say 100ms.

Sources

This is where I got most of my information to writ this article, there's some more details on the GitHub pages, if you're into some more advanced stuff, like EEPROM or deep sleep etc.

- https://github.com/esp8266/Arduino/issues/2942
- https://github.com/esp8266/Arduino/pull/2533/files
- https://github.com/esp8266/Arduino/blob/master/doc/libraries.md
- https://github.com/esp8266/Arduino/blob/master/doc/reference.md
- https://github.com/esp8266/Arduino/blob/master/doc/boards.md

Wi-Fi

Using the ESP8266 as a simple microcontroller is great, but the reason why most people use it, is its Wi-Fi capabilities. In this chapter, we'll dive into the wonderful world of network protocols, like Wi-Fi, TCP, UDP, HTTP, DNS ... All these acronyms might intimidate you, but I'll try my best to explain them step-by-step and in an easy way.

Some paragraphs are in *italic*. These provide some extra information, but are not critical to understanding the ESP's Wi-Fi functions, so don't get frustrated if there are things you don't understand.

It's really hard to give a clear explanation, without over-complicating things and while keeping it short enough as well. If you've got any feedback or remarks, be sure to leave a comment to help improve this article. Thanks!

The TCP/IP stack

The system most people refer to as 'The Internet' isn't just one protocol: it's an entire stack of layers of protocols, often referred to as the **TCP/IP stack**. We'll go over these different layers, because we need to understand how our ESP8266 communicates with other devices on the network.

Layer	Protocols
Application	HTTP, FTP, mDNS, WebSocket, OSC ...
Transport	TCP, UDP
Internet	IP
Link	Ethernet, Wi-Fi ...

The Link layer

The link layer contains the physical link between two devices, an Ethernet cable, for example, or a Wi-Fi connection. This is the layer that is closest to the hardware.
To connect an ESP8266 to the network, you have to create a Wi-Fi link. This can happen in two different ways:

1. The ESP8266 connects to a wireless access point (WAP or simply AP). The AP can be built-in to your modem or router, for example.
   In this configuration, the ESP acts like a wireless **station.**
2. The ESP8266 acts as an **access point** and wireless stations can connect to it. These stations could be your laptop, a smartphone, or even another ESP in station mode.

Once the Wi-Fi link is established, the ESP8266 is part of a **local area network** (LAN). All devices on a LAN can communicate with each other.
Most of the time, the AP is connected to a physical Ethernet network as well, this means that the ESP8266 can also communicate with devices that are connected to the AP (modem/router) via a wired Ethernet connection (desktop computers, gaming consoles and set-top boxes, for instance).
If the ESP8266 is in access point mode, it can communicate with any station that is connected to it, and two stations (e.g. a laptop and a smartphone) can also communicate with each other.

The ESP can be used in AP-only, station-only, or AP+station mode.

TL;DR

The link layer is the physical link between devices: in the case of the ESP8266, this is a WiFi connection. The ESP can act as a station and connect to an access point, or act as an access point and let other devices connect to it.

The Internet or Network layer

Although the devices are now physically connected (either through actual wires (Ethernet) or through radio waves (Wi-Fi)), they can't actually talk to each other yet, because they have no way of knowing where to send the message to.
That's where the **Internet Protocol** (IP) comes in. Every device on the network has a personal IP address. The DHCP server (Dynamic Host Configuration Protocol Server) makes sure that these addresses are unique.

This means that you can now send a message to a specific address.

There are two versions of the Internet Protocol: IPv4 and IPv6. IPv6 is an improved version of IPv4 and has much more addresses than IPv4 (because there are much more devices than available IPv4 addresses). In this article, we'll only talk about IPv4 addresses, since most LANs still use them.

The IP address consists of 4 numbers, for example *192.168.1.5* is a valid IPv4 address. It actually consists of two parts: the first part is *192.168.1*, this is the address of the local network. The last digit, 5 in this case, is specific to the device.

By using IP addresses, we can find the ESP8266 on the network, and send messages to it. The ESP can also find our computer or our phone, if it knows their respective IP addresses.

Sub-net mask *(optional)*

This subdivision of the IP address is determined by the sub-net mask, often written as 255.255.255.0. You can see that it consists of four numbers, just like the IP address. If a part of the sub-net mask is 255, it means that the corresponding part of the IP address is part of the network address, if it's 0, the corresponding IP part is part of the address of the specific address. A different notation to "IP: 192.168.1.5, sub-net mask: 255.255.255.0" would be "192.168.1.5/24", because the binary representation of the sub-net mask is 11111111.11111111.11111111.00000000, and it has 24 ones.
If you want to know more about sub-nets, I'd recommend you to read the Wikipedia article.
(A quick tip to help you remember: it's called the sub-net mask, because if you perform a bitwise AND operation on the IP address and the sub-net mask (i.e. use the sub-net mask as a mask for the IP address), you get the address of the sub-net.)

MAC addresses and ARP *(optional)*

It is actually impossible to send packets directly to another machine using only the IP address. To send a packet to a specific device on the LAN (Wi-Fi or Ethernet), you have to know its MAC-address. The MAC address is a unique number that is unique for every network device, and it never changes, it's hardwired in the network chip. This means that every ESP8266, every network card, every smartphone ... ever made, has a different MAC address.
So before the ESP can send a packet to your smartphone for example, it has to know its MAC address. It doesn't know this yet, the ESP only knows the IP address of the smartphone, say 192.168.1.6. To do this, the ESP sends a broadcast message (i.e. a message addressed to all devices on the LAN) saying "I'm looking for the MAC address of the device with the IP address 192.168.1.6". The ESP also includes its own IP and MAC address with the message. When the smartphone receives this broadcast message, it recognizes its own IP address, and responds to the ESP by sending its own MAC address. Now the ESP and the phone both know each other's IP and MAC addresses, and they can communicate using IP addresses. This method is called the Addres Resolution Protocol, or ARP.

What about the Internet?

As you might have noticed, I only talked about the *local* area network, these are the computers in your own house. So how can the ESP8266 communicate with the Internet, you may ask? Well, there's a lot of network infrastructure involved in 'The Internet', and they all obey the IP rules, to make sure most of your packets arrive at there destination. It's not that simple of course, there's a lot of things going on, like routing and Network Address Translation (NAT), but that falls outside the scope of this article, and it's not really something most people have to worry about.

TL;DR

The Internet layer uses IP addresses in order to know where it should send the data. This means that two devices can now send packets of data to each other, even over the Internet.

The Transport layer

The different devices in the network do their best to deliver these IP packets to the addressee, however, it's not uncommon for a packet to get lost, so it will never arrive. Or the packet might get corrupted on the way: the data is no longer correct. IP also can't guarantee that the packets arrive in the same order they were sent in.
This means that we can't *reliably* send messages yet by only using the link and the Internet layer, since we can never know when and whether a packet will arrive, or know for certain that a received packet is correct.

We need a third layer on top of the Internet layer: the Transport layer.

There are mainly two protocols that make up this third layer: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

- **TCP** makes sure that all packets are received, that the packets are **in order**, and that corrupted packets are re-sent. This means that it can be used for communication between multiple applications, without having to worry about **data integrity** or packet loss. This is why it's used for things like downloading webpages, sending email, uploading files etc.
- **UDP** on the other hand, doesn't guarantee that every packet reaches its destination, it does **check for errors** however, but when it finds one, it just destroys the packet, without re-sending it. This means that it's **not as reliable as TCP**, but it's faster, and has a much **lower latency**, because it doesn't require an open connection to send messages, like TCP does. That's why it's used in voice and video chats, and for example in online games.

If you want to know more about the differences between TCP and UDP, check out this video.

TL;DR

The IP protocol is not reliable, and has no error checking. TCP solves this by re-sending lost or corrupt packages, and orders packets that are received in the wrong order. UDP also checks for corrupt packages, but doesn't re-send them, so it has less latency than TCP.

The Application layer

We now have reliable communication using TCP, but there's still one problem. Think of it this way: you are sending a letter, and TCP guarantees that it will arrive at its destination, but if the receiver doesn't understand the language it's written in, he won't know what to do with it.
In other words, we need a fourth layer of protocols, for two programs to be able to communicate with each other.
There's lots of different protocols out there, but we'll mostly focus on the protocols for web servers and browsers.

HyperText Transfer Protocol

The HyperText Transfer Protocol, or HTTP, is the protocol (cfr. language) that is used by both web servers and web clients in order to communicate. It uses text to perform send requests and responses from the client to the server and back again.
For example, when you type http://www.google.com into the address bar of a web browser (client), it will send an HTTP GET request to the Google web server. The server understands this HTTP request, and will send the Google webpage as a response. Or when you upload an image to Instagram, your browser sends an HTTP POST request with your selfie attached to the Instagram server. The server understands the request, saves the image and adds it into the database, sends the URL of the new image back to your browser, and the browser will add the image on the webpage.
As you can see, neither the client nor the server has to worry about the integrity of the messages they send, and they know that the recipient understands their language, and that it will know what to do with a certain HTTP request.
Most modern sites use a secure version of HTTP, called HTTPS. This secure connection encrypts the data, for security reasons. (You don't want anyone reading the packets from your mail server, or the packets you sent to your bank, for instance.)

WebSocket

HTTP is great for things like downloading webpages, uploading photos etc. but it's quite slow: every time you send an HTTP request, you have to start a new TCP connection to the server, then send your request, wait for the server to respond, and download the response. Wouldn't it be great if we didn't have to open a new connection every time we want to send some data, and if we could send and receive data at the same time at any moment we'd like? That's where WebSocket comes to the rescue: you can keep the TCP connection with the server open at all times, you get perfect TCP reliability, and it's pretty fast.

Open Sound Control

HTTP and WebSocket both use TCP connections. What if you want lower latency? Well, Open Sound Control, or OSC, uses UDP to send small pieces of data, like ints, floats, short text etc ... with very low latency. It was originally designed for controlling low latency audio applications, but it's a very flexible protocol, so it's often used for low-latency tasks other than audio control.

Domain Name System

As mentioned before, you can only send a message to another computer if you know its IP address. But when you browse the Internet, you only know a website's domain name (e.g. www.google.com). Your computer uses the Domain Name System to translate this domain name to the right IP address. More on this later.

Sources

- https://en.wikipedia.org/wiki/Internet_protocol_suite
- https://en.wikipedia.org/wiki/Port_(computer_networking)
- https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- https://en.wikipedia.org/wiki/Internet_Protocol
- https://en.wikipedia.org/wiki/User_Datagram_Protocol

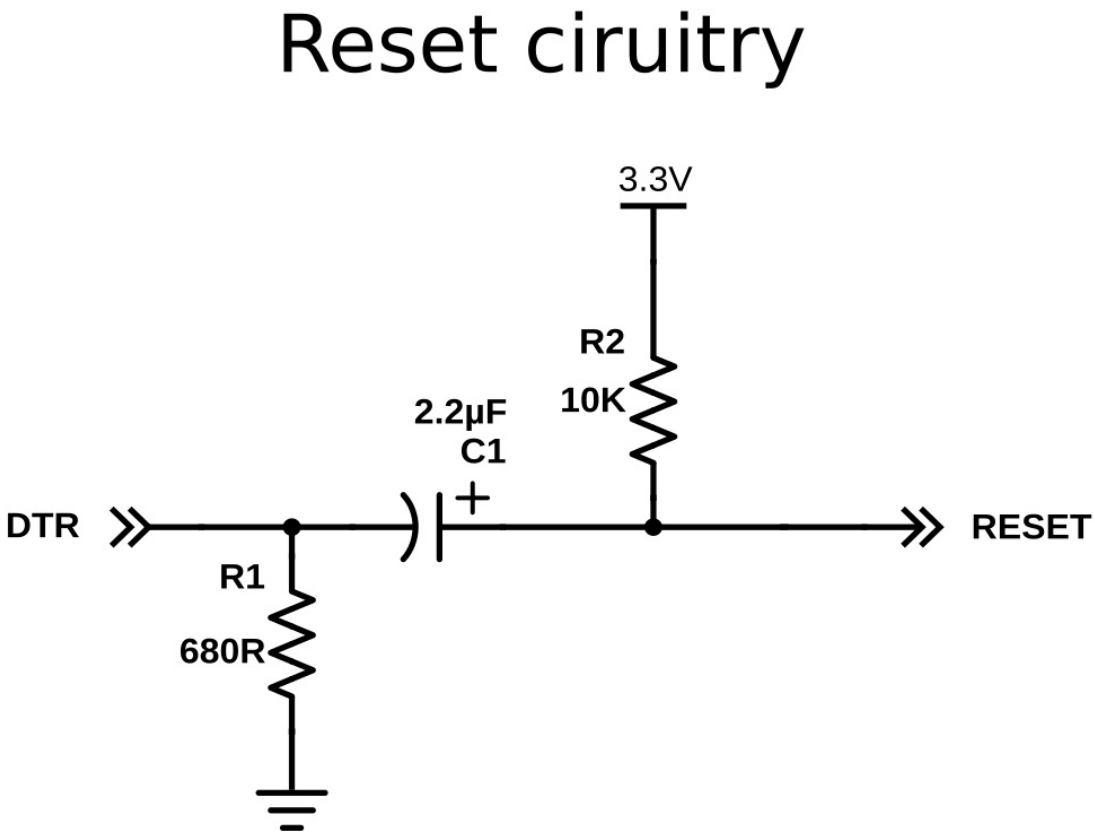
Uploading sketches to the ESP8266

The upload procedure for ESP8266 boards is a little different from the normal Arduino procedure. Most Arduinos will automatically reset when a new program is being uploaded, and will automatically enter programming mode.
On some ESP boards you have to manually enter programming mode, and on the bare-bones modules, you even have to reset them manually.
However, there are some simple circuits you can use to get automatic uploads.

Auto-reset

This only applies to boards without an on-board USB-to-Serial converter.

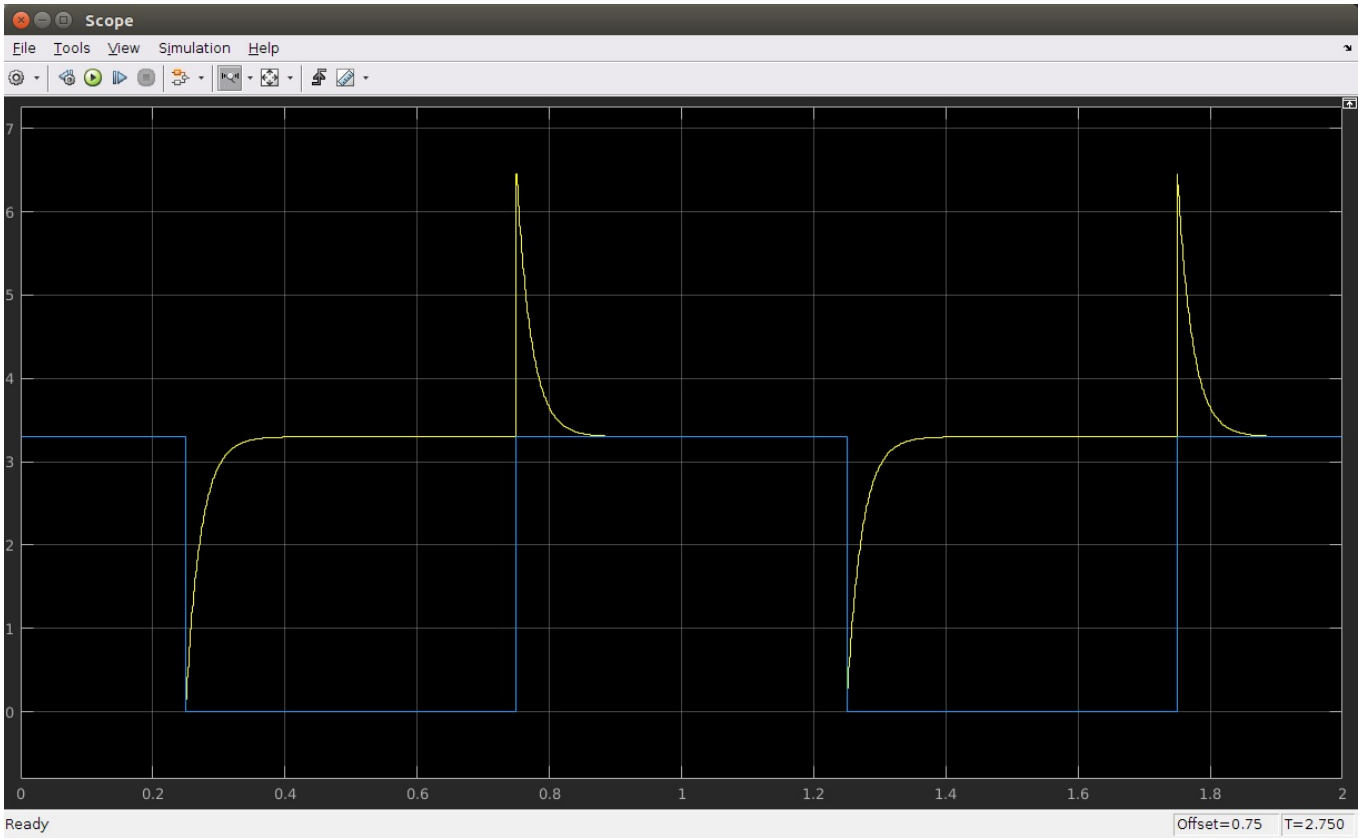
If the USB-to-Serial converter you're using has a DTR flow control line, you can automate the reset signal. When sending data to the ESP, the DTR line goes low, and stays low for some time. To reset the ESP, we need a low pulse on the RST pin. The problem is that the DTR pin *stays* low. To solve this, we're going to build a crude edge detector circuit, using a capacitor. Take a look at the following schematic:



You might recognize that this is basically a low-cut filter. In normal conditions, DTR is high (3.3V), and the reset line is also high, because of the pull-up resistor R2. This means that the voltage across the capacitor is 0V. When DTR suddenly drops (to 0V), the voltage across the capacitor is still 0V, meaning that the reset line will be at 0V + 0V = 0V, and a reset is triggered.

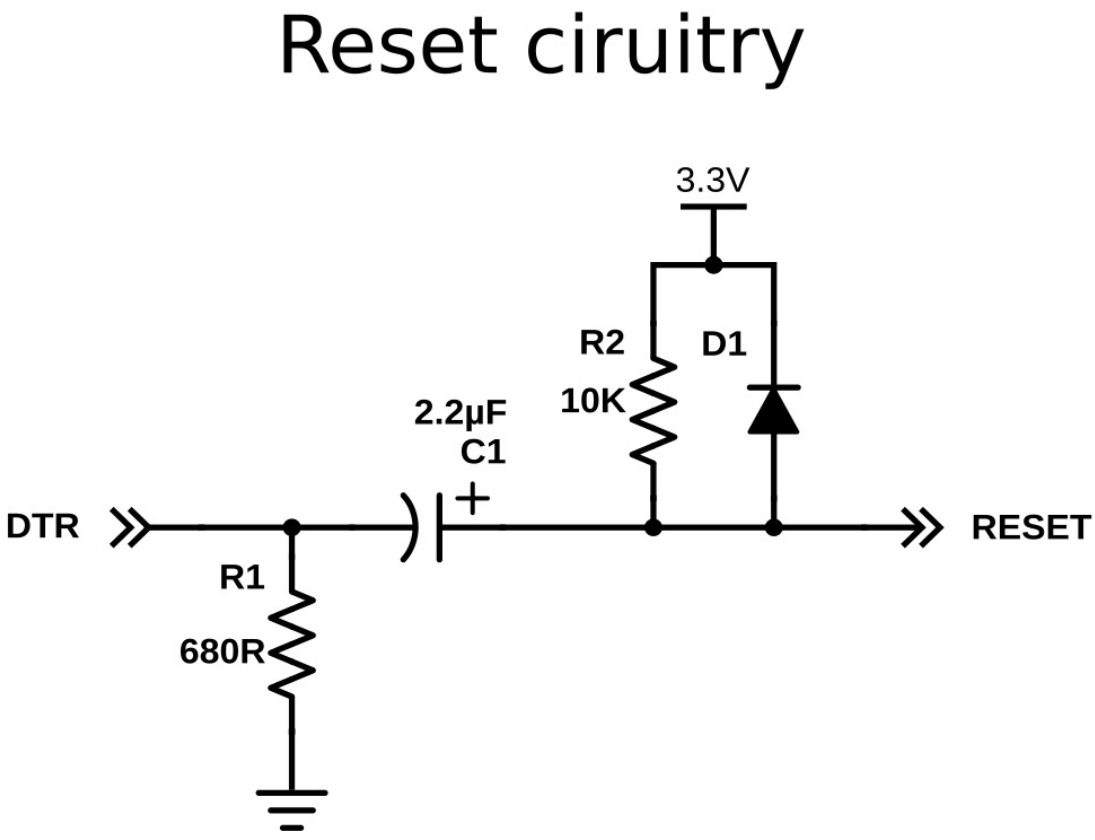
However, C1 immediately starts charging through R2, and reaches 3.3V. At this point, DTR is still at 0V, meaning that there's now 3.3V across the capacitor. When DTR rises again, the reset line will be at 3.3V + 3.3V = 6.6V, and then immediately starts to discharge through R2, finally reaching 3.3V again, with 0V across C1.
This is a problem: 6.6V can damage the ESP, so we have to find a way to get rid of the positive peak.

One glance at this MATLAB simulation shows us the problem even better:

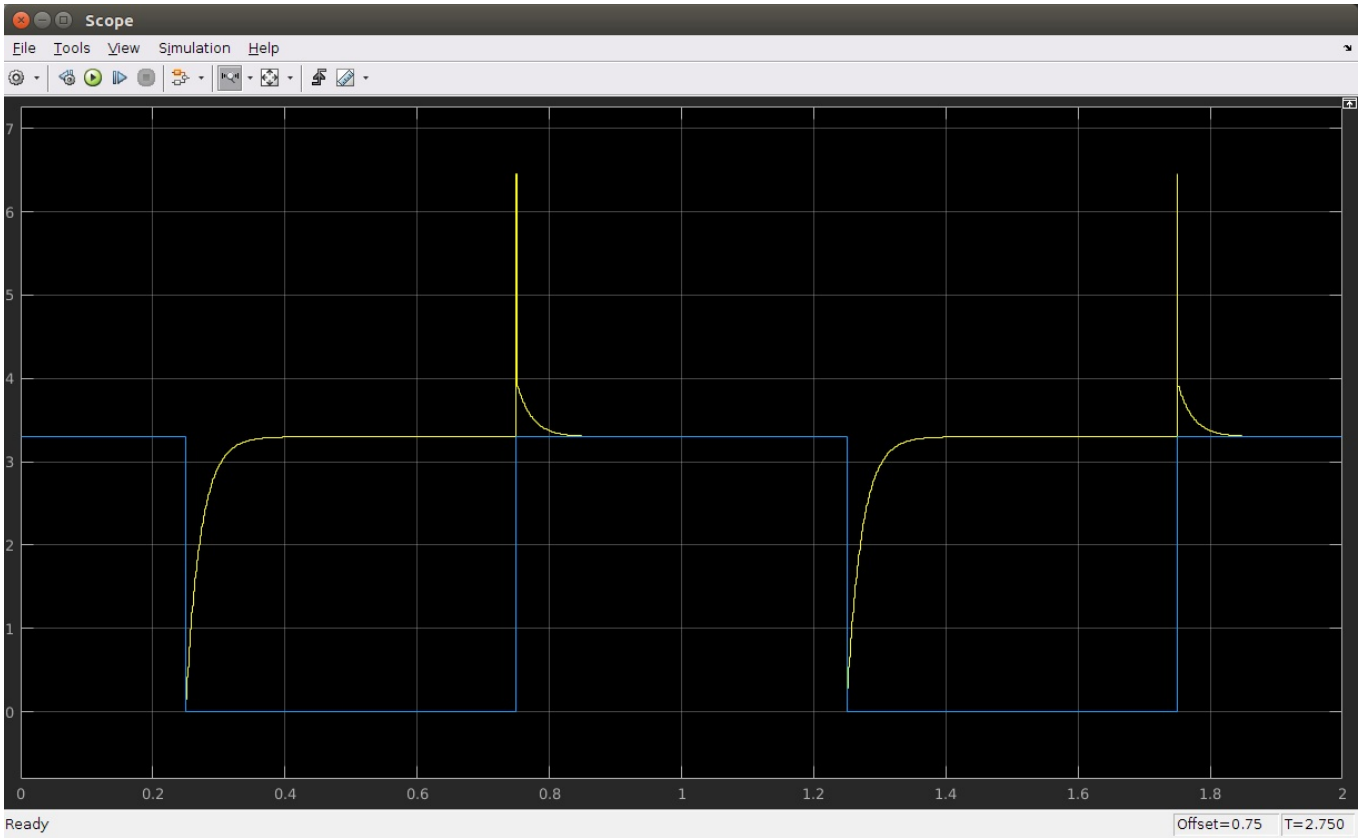


The blue signal is the voltage on the DTR pin, and the yellow signal is the voltage on the reset pin.

The solution is to add a diode: while charging the capacitor, it shouldn't change anything, so it should be reverse biased (just a fancy way of saying that it's not conducting any current because the polarity is the other way around), and while the capacitor is discharging, it should discharge the capacitor "immediately".
Here's what that looks like:



Let's run the simulation again to check if our problem is solved:



As you can see, the 6.6V peak is now very narrow, just like we wanted. It's impossible to discharge the capacitor instantly, that would require a capacitor and a diode with 0Ω of series resistance, and an infinite current, which is impossible, obviously. There's also a smaller but relatively wide peak of approximately 3.9V. This is because a diode only conducts when the voltage across it is higher than ~600mV. This means that the last 0.6V that's left in the capacitor (from 3.9 to 3.3V) will still be discharged through R2 only.
Nevertheless, the voltage peak is much lower and narrower than without the diode, and it's safe to connect to the ESP8266.

This exact circuit is also used in the Arduino Uno, for example.

Note: if you followed the instructions in the hardware step correctly, you should already have added R2 to your ESP.

How to use Auto-reset

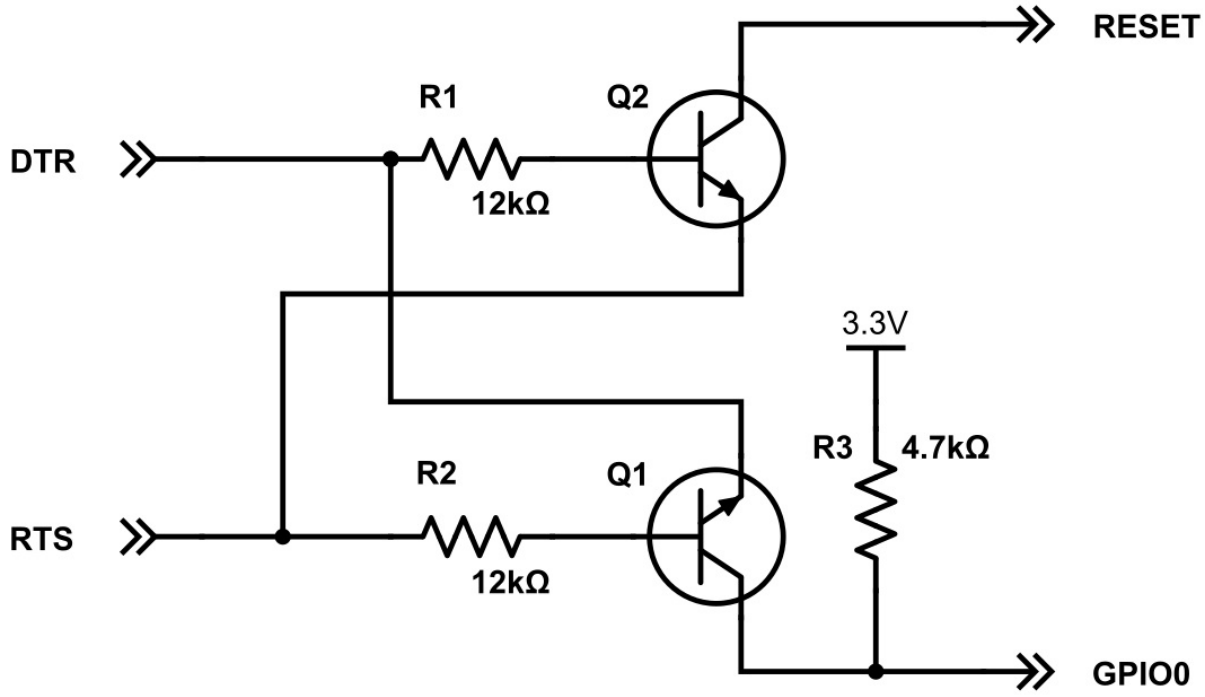
To use this auto-reset circuit, connect it to the DTR line of your USB-to-Serial converter, and to the reset line of the ESP, as shown in the diagram. Then click compile (just because the first compilation can take quite some time). Go to Tools > Reset and select 'ck'. When it's done compiling, hold down the program button we added in the hardware step, and click upload. Wait for it to say "Uploading..." and then release the program button.

Auto-reset and Auto-program

This only applies to boards without an on-board USB-to-Serial converter.

The method above still requires you to press a button to upload a new sketch. If your USB-to-Serial converter has a RTS line as well as a DTR line, you can automate the entire process.

Auto-Reset & Auto-Program



You may find out that the 4.7kΩ resistor doesn't work for you. In that case, try some other value, like 10kΩ, for example.

This method was first used in the NodeMCU, so go to Tools > Reset Method, and select "nodemcu". This will drive the DTR&RTS pins high and low in the right sequence to get it in programming mode before uploading.

This is by far the best method, but the problem is that you need access to both the RTS and DTR pins, while most USB-to-Serial adapters break out only one of the two.

Manual reset and manual program

This only applies to boards without an on-board USB-to-Serial converter.

If you don't have a USB-to-Serial converter with DTR and RTS lines, you could also just use the reset and program buttons we added in the hardware chapter. To get the ESP in program mode, GPIO0 must be low while booting:

1. press and hold the reset button
2. press and hold the program button
3. release the reset button, the ESP will boot in program mode
4. release the program button
5. upload the sketch

If you want to get out of program mode without uploading, just press reset (without pressing the program button).

Board options

If your specific board is in the Tools > Board list (e.g. NodeMCU, SparkFun and Adafruit boards), you can just select it, and you will get the right settings. When your board isn't in the list, you'll have to select a Generic ESP8266. In that case there's lots of new options in the Tools menu of the Arduino IDE, so let's go over them and pick the right settings.

Flash Mode

Like I said before, the ESP8266 uses an external flash chip for storage. You can communicate with this chip over 2 datalines (DIO), or over all 4 datalines (QIO). Using 4 lines is two times faster than 2 lines, so in most cases, you should choose QIO. (If you're doing some advanced stuff and you need 2 more GPIO pins, you could use 2 lines instead of 4, and use the 2 lines as I/O. Most modules don't give you access to these pins, though.)

Flash Size

Different boards/modules have different sizes of flash chips on board. There are boards with 512kB, 1MB, 2MB and 4MB of flash. To know how much flash your board has, you can try the Examples > ESP8266 > CheckFlashConfig to see if your flash setting is correct, or you can check the specifications of your specific board online.
You can also select the SPIFFS (SPI Flash File System) size. The SPIFFS partition is a small file system to store files. If you're not using it, you can select the minimum. Later on in the article, we'll use SPIFFS, and I'll remind you to select a larger SPIFFS size, but for now, it doesn't really matter.

Debug port

There's a load of things going on when the ESP is running: Things like Wi-Fi connections, TCP connections, DNS lookups ... you name it. All these small tasks produce a whole lot of debug output to help you troubleshoot. However, in a normal situation, where your program is behaving as expected, you don't need all those debug messages to flood the Serial Monitor, so you can just turn them off by selecting 'Disabled'.
If you do wish to receive debug messages, you can select the port to send them to. (Serial on pins 1 and 3, or Serial1 on pin 2)

Debug level

This allows you to choose what kind of debug messages you want to show.

Reset Method

As mentioned in the paragraphs above, there are different methods for auto-reset and auto-program. If you're using the first method (using the edge detector), you should use 'ck', if you use the two-transistor circuit, select 'nodemcu'.

Flash Frequency

If you need some extra memory speed, you could change the flash frequency from 40MHz to 80MHz. This is the clock frequency of the SPI/SDIO link.

CPU Frequency

If you need some extra CPU performance, you can double the clock speed from 80MHz to 160MHz. It's actually an overclock, but I've never had any issues or instability.

Upload Speed

The baud rate for uploading to the ESP. The default is 115200 baud, but you can go higher (if you're changing your sketch a lot, it might be too slow). 921600 baud works most of the time, but you may get an error sometimes, if that's the case, switching back to 115200 will probably solve all problems.

Establishing a Wi-Fi connection

Like I mentioned in the previous chapter, the ESP8266 can operate in three different modes: Wi-Fi station, Wi-Fi access point, and both at the same time. We'll start by looking at the configuration of a Wi-Fi station.

Station mode

Connecting to one specific network

```
#include <ESP8266WiFi.h>        // Include the Wi-Fi library

const char* ssid     = "SSID";         // The SSID (name) of the Wi-Fi network you want to connect to
const char* password = "PASSWORD";     // The password of the Wi-Fi network

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  WiFi.begin(ssid, password);             // Connect to the network
  Serial.print("Connecting to ");
  Serial.print(ssid); Serial.println(" ...");

  int i = 0;
  while (WiFi.status() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
    delay(1000);
    Serial.print(++i); Serial.print(' ');
  }

  Serial.println('\n');
  Serial.println("Connection established!");
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());         // Send the IP address of the ESP8266 to the computer
}

void loop() { }
```

The code to connect to a wireless access point is relatively straightforward: enter the SSID and the password of the network you want to connect to, and call the `WiFi.begin` function. Then wait for the connection to complete, *et voilà*, your ESP8266 is now connected to your Local Area Network.

Don't believe me? I'll prove it to you: open the Serial monitor (CTRL+SHIFT+M) and upload the sketch. You should see something like this:

```
Connecting to SSID ...
1 2 3 4 5 6 ...

Connection established!
IP address: 192.168.1.3
```

Now go to your computer and open up a terminal: On Windows, search for "Command Prompt", on Mac or Linux, search for "Terminal". You could also use the shortcuts: on Windows, hit + R, type "cmd" and hit enter, on Linux, use `CTRL+ALT+T` .

Next, type `ping` , and then the IP address you received in the Serial monitor. If you're on Mac or Linux, use `CTRL+C` to stop it after a couple of lines. The output should look something like this:

```
user@computername:~$ ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=128 time=6.38 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=128 time=45.2 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=128 time=69.1 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=128 time=94.0 ms
64 bytes from 192.168.1.3: icmp_seq=5 ttl=128 time=20.5 ms
64 bytes from 192.168.1.3: icmp_seq=6 ttl=128 time=7.37 ms
^C
--- 192.168.1.3 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5003ms
rtt min/avg/max/mdev = 6.384/40.463/94.047/32.588 ms
```

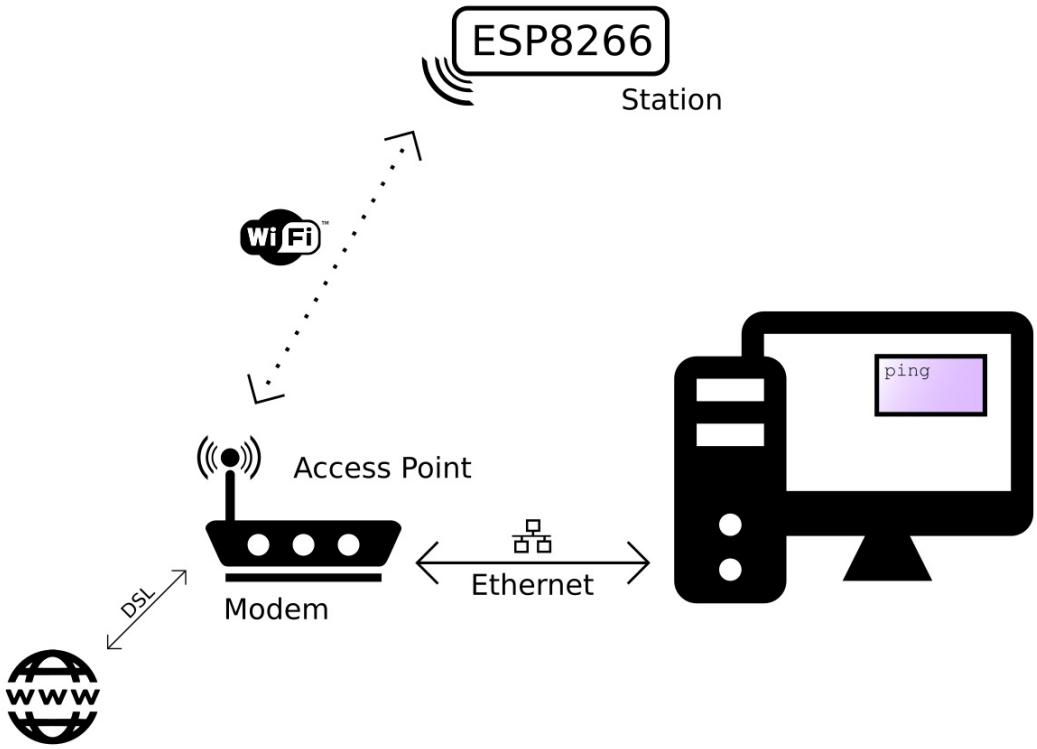
The ping command sends small packets to the IP address of the ESP8266. When the ESP receives such a

packet, it sends it back to the sender. Ping is part of the second layer of the TCP/IP stack, the Internet layer. It relies on both the Data Link layer (Wi-Fi) and the Internet Protocol*.
You can see that in the example above, we sent 6 packets to the ESP, and we also received 6 response (echo) packets. This tells us that the Data Link, the Wi-Fi connection, and the Internet Protocol are working correctly.

We now know that the ESP can successfully communicate with other devices on the network, and if your local network is online (if it is connected to the Internet via your modem), the ESP can also communicate with any device on the web !

Ping is a great tool to check if the ESP (or any device, really) is still connected to the network, and if it's still working fine.
One drawback is that IP addresses can change over time, but that's a problem we'll address in one of the following chapters ...

(*) I'm simplifying things a bit here. Actually, ping is part of the Internet Control Message Protocol (ICMP), that's also part of the second layer, just like the Internet Protocol. Don't worry too much about it, just remember that if you can send ping packets to a device, you can also send IP packets.



The device with the antenna serves many different purposes:

- **Access point**: Other Wi-Fi devices can connect to it, to be part of the local network.
- **Router**: It routes IP packets to the right sub-nets so that they will arrive at their destination. E.g. if the computer sends a message that is meant for the ESP over the Ethernet sub-net, the router will send the packet to the Wi-Fi sub-net, because it knows that's where the ESP is.
- **Modem**: if the router can't find the addressee on the local network, the packet will be passed on to the integrated modem, and it will be sent to the Internet Service Provider over a DSL line, heading for the Internet, where lots of other routers will try to get the packet to the right destination.

But in reality, you don't have to worry too much about it, because it's all done for you, in a fraction of a second without you even noticing it!

Automatically connect to the strongest network

The sketch above might be enough for your specific application, but if you need to be able to connect to multiple Wi-Fi networks, for example the Wi-Fi at home and the Wi-Fi at the office, it won't work.
To solve this problem, we'll use the Wi-Fi-Multi library: You can add as many networks as you like, and it automatically connects to the one with the strongest signal.

```
#include <ESP8266WiFi.h>        // Include the Wi-Fi library
#include <ESP8266WiFiMulti.h>   // Include the Wi-Fi-Multi library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(1000);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer
}

void loop() { }
```

Access Point mode

To configure the ESP8266 as an access point, to allow other devices like smartphones or laptops to connect to it, you can use the softAP function:

```
#include <ESP8266WiFi.h>        // Include the Wi-Fi library

const char *ssid = "ESP8266 Access Point"; // The name of the Wi-Fi network that will be created
const char *password = "thereisnospoon";   // The password required to connect to it, leave blank for an
open network

void setup() {
  Serial.begin(115200);
  delay(10);
  Serial.println('\n');

  WiFi.softAP(ssid, password);             // Start the access point
  Serial.print("Access Point \"");
  Serial.print(ssid);
  Serial.println("\" started");

  Serial.print("IP address:\t");
  Serial.println(WiFi.softAPIP());         // Send the IP address of the ESP8266 to the computer
}

void loop() { }
```

To see if it works, open the Wi-Fi settings on your computer, look for a network called "ESP8266 Access Point", enter the password "thereisnospoon", and connect to it. Then open a terminal, and ping to 192.168.4.1 (this is the default IP address of our ESP AP). You'll see that the ESP responds to your pings.

However, if you try to go to an online website, you'll get a timeout or a DNS error. This is because the ESP itself is not connected to the internet. The sub-net that consists of the ESP and the computer is not connected to any other networks, so there's no way for a packet on this network to make it to the Internet.

If you connected a second station to the ESP access point on the other hand, you would be able to ping from one station to the other without problems, because they're on the same network.

Multicast Domain Name System

DNS

Let's face it, constantly typing IP addresses is really cumbersome, and it would be impossible to remember all your favorite websites' addresses, especially if they use IPv6.
That's why domain names were introduced: a simple string of text that's easy to remember, for example www.google.com.

However, to send a request to a website, your computer still needs to know its IP address. That's where DNS comes in. It stands for Domain Name System, and is a way to translate a website's domain name to its IP address. On the Internet, there are a lot of DNS servers. Each DNS server has a long list of domain names and their corresponding IP addresses. Devices can connect to a DNS server and send a domain name, the DNS server will then respond with the IP address of the requested site.
You could compare it to a telephone directory: you can look up a name to find the corresponding phone number.

The DNS lookup happens completely in the background: when you go to a website in your browser, it will first send a request to a DNS server (this implies that the computer knows the IP address of the DNS server itself), wait for the response of the lookup, and then send the actual request to the right IP address.

mDNS

DNS works great for normal sites on the Internet, but most local networks don't have their own DNS server. This means that you can't reach local devices using a domain name, and you're stuck using IP addresses ...

Fortunately, there's another way: multicast DNS, or mDNS.
mDNS uses domain names with the *.local* suffix, for example http://esp8266.local. If your computer needs to send a request to a domain name that ends in *.local*, it will send a multicast query to all other devices on the LAN that support mDNS, asking the device with that specific domain name to identify itself. The device with the right name will then respond with another multicast and send its IP address. Now that your computer knows the IP address of the device, it can send normal requests.

Luckily for us, the ESP8266 Arduino Core supports mDNS:

```
#include <ESP8266WiFi.h>        // Include the Wi-Fi library
#include <ESP8266WiFiMulti.h>   // Include the Wi-Fi-Multi library
#include <ESP8266mDNS.h>        // Include the mDNS library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(1000);
    Serial.print(++i); Serial.print(' ');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (!MDNS.begin("esp8266")) {             // Start the mDNS responder for esp8266.local
    Serial.println("Error setting up MDNS responder!");
  }
  Serial.println("mDNS responder started");
}

void loop() { }
```

Upload it and open ping again. Try to ping to *esp8266.local:*

```
user@computername:~$ ping esp8266.local
PING esp8266.local (10.92.237.128) 56(84) bytes of data.
64 bytes from 10.92.237.128: icmp_seq=1 ttl=128 time=5.68 ms
64 bytes from 10.92.237.128: icmp_seq=2 ttl=128 time=3.41 ms
64 bytes from 10.92.237.128: icmp_seq=3 ttl=128 time=2.55 ms
64 bytes from 10.92.237.128: icmp_seq=4 ttl=128 time=2.19 ms
64 bytes from 10.92.237.128: icmp_seq=5 ttl=128 time=2.29 ms
64 bytes from 10.92.237.128: icmp_seq=6 ttl=128 time=2.74 ms
^C
--- esp8266.local ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5007ms
rtt min/avg/max/mdev = 2.190/3.148/5.687/1.202 ms
```

As you can see, ping will automatically find the IP address of the ESP for you.

mDNS is supported on Windows, OSX, Linux and iOS, but not (yet?) on Android.
It's a real shame that Android doesn't support it, you can help by starring this issue report for the Chromium project to ask for mDNS support in Chrome on Android.

Of course, you can change the domain name of the ESP by changing the parameter of `MDNS.begin`.

ESP8266 Web Server

Being able to ping the ESP is quite an achievement if you look at it from a technical point of view, but for most people, it's not that exciting, and not really useful.
In this chapter, I'll cover the basics of a web server, and teach you how to host a web page on the ESP.

Web servers

A web server is an Internet-connected device that stores and serves files. Clients can request such a file or another piece of data, and the server will then send the right data/files back to the client. Requests are made using HTTP.

HTTP

HTTP or the Hypertext Transfer Protocol is the text-based protocol used to communicate with (web) servers. There are multiple HTTP request methods, but I'll only cover the two most widely used ones: GET and POST.

HTTP GET

GET requests are used to retrieve data from a server, a web page for instance. It shouldn't change anything on the server, it just *gets* the data from the server, without side effects.

When you open a webpage in your browser, it will take the URL and put it in an HTTP GET request. This is just plain text. Then it will send the request to the right server using TCP. The server will read the request, check the URL, and send the right HTTP response for that URL back to the browser.

The anatomy of a GET request

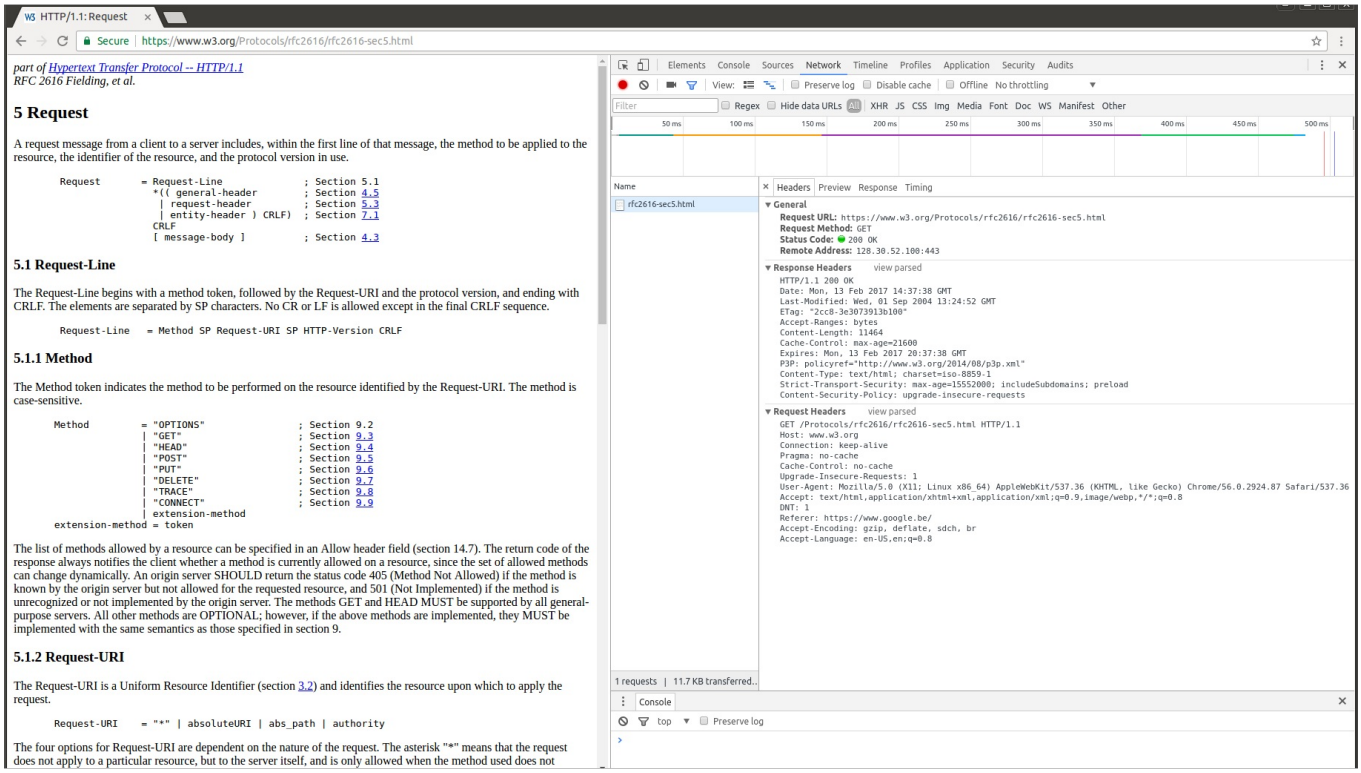
The most important parts of a GET request are the **request line** and the **host header**. Let's take a look at an example:
If you click the following link: https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html, your browser will send out the following HTTP request:

```
GET /Protocols/rfc2616/rfc2616-sec5.html HTTP/1.1
Host: www.w3.org
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87
Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
DNT: 1
Referer: https://www.google.be/
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: en-US,en;q=0.8
```

The first line is the **request line**: it contains the **request method:** GET, in this case, the **URI** or Uniform Resource Identifier: /Protocols/rfc2616/rfc2616-sec5.html, and the HTTP version: 1.1.
The second line is the **host header**, it specifies the domain name of the host (server).
There are many other headers as well, but they're not really important when using an ESP8266.
Most servers will check if the URI is a file on their file system, and if that's the case, they'll send that file as a response.

Viewing HTTP headers in the browser

If you want to check the headers your browser sends, you can press F12, go to the network tab, reload the page, and click the request you want to inspect. If you want, you can click 'view source', this will show you the actual HTTP text.
Here's what that looks like in Chrome:



Sending extra information to the server

Sometimes, you might want to add extra information to the GET request. You can send key-value pairs by adding a question mark (?) to the URI, followed by key=value. Multiple pairs are separated by an ampersand (&).
For example:

```
GET /get-phone-number.php?firstName=John&lastName=Doe HTTP/1.1
Host: www.phonebook.example.com
...
```

If you use any special characters in the key or value names, you have to URL-encode them.

HTTP POST

POST requests are used to send data to the server, for example, to send your user name and password to the server when you log in, or when you upload a photo. Unlike GET, POST can change the data on the server or the state of the server.
POST has a body that can contain data that is sent to the server.

The anatomy of a POST request

For example, the login page of your favorite site might send something like this when you enter your credentials and click the *login* button:

```
POST /login.php HTTP/1.1
Host: www.example.com
Connection: keep-alive
Content-Length: 480
Origin: http://www.example.com
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryQNEJOasMvgAOg8Kt
...
```

As you can see, the **request line** now has the POST method in it, and is still followed by a URI, /login.php, and the HTTP version, 1.1. The **host header** still contains just the domain name.

The real difference is the **request body**: a GET request has no payload, while you can add a lot of data to the body of a POST request. This data could be normal key-value pairs, like a username and a password, or actual files that are being uploaded.
Also note the Content-Type header: it tells the server what kind of data can be found in the body of the POST request.

Let's take a look at the body of the login example:

```
------WebKitFormBoundaryQNEJOasMvgAOg8Kt
```

```
Content-Disposition: form-data; name="username"

John Doe
------WebKitFormBoundaryQNEJOasMvgAOg8Kt
Content-Disposition: form-data; name="password"

p@ssw0rd123
------WebKitFormBoundaryQNEJOasMvgAOg8Kt
Content-Disposition: form-data; name="token"

9i9ZoLHl5pkRAeuKCEu76TbaCnMphwYkPEovEUY9PHk=
------WebKitFormBoundaryQNEJOasMvgAOg8Kt--
```

As you can see, there are three parameters inside the body, every parameter has a name (e.g. username), and a value (e.g. John Doe).

You could also use the same syntax we used before when adding parameters to a GET request:

```
POST /add-user.php HTTP/1.1
Host: www.example.com
Content-Length: 27
Content-Type: application/x-www-form-urlencoded
...
```

And the payload:

```
firstName=John&lastName=Doe
```

As you can see, the `Content-Type`header is different, indicating that the encoding of the values in the payload is different.

HTTP status codes

A server should answer all requests with an HTTP status code. This is a 3-digit number indicating if the request was successful or telling the client what went wrong. Here's a table with some of the most important and useful ones.

Status Code	Meaning
200	OK: the request was successful
303	See Other: used to redirect to a different URI, after a POST request, for instance
400	Bad Request: the server couldn't understand the request, because the syntax was incorrect
401	Unauthorized: user authentication is required
403	Forbidden: the server refuses to execute the request, authorization won't help
404	Not Found: the requested URI was not found
500	Internal Server Error: The server encountered an unexpected condition and couldn't fulfill the request

TCP & UDP Ports

In most cases, one device has many different services, for example, a web server, an email server, an FTP server, a Spotify streaming service, ...
If the device had just an IP address, it would be impossible to know which application a packet was sent to. That's why every service has a port number. It's an identifier for all different services or applications on a single device. In the example above, the web server will only listen for requests on port 80, the email server only on port 25, the FTP server only on port 20, Spotify will only receive streams on port 4371 ...
To specify a certain port, you can use a colon after the IP address of after the domain name. But most of the time, you don't have to add it explicitly. For example, all web servers listen on port 80, so a web browser will always connect to port 80.

- http://stackoverflow.com/questions/176264/what-is-the-difference-between-a-uri-a-url-and-a-urn
- https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

ESP8266 First Web Server

The actual implementation of a web server is much easier than it sounds, because the ESP8266 Arduino Core includes some great libraries that handle pretty much everything for you. Let's look at a basic Hello World! example.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>   // Include the WebServer library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

void handleRoot();              // function prototypes for HTTP handlers
void handleNotFound();

void setup(void){
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (MDNS.begin("esp8266")) {              // Start the mDNS responder for esp8266.local
    Serial.println("mDNS responder started");
  } else {
    Serial.println("Error setting up MDNS responder!");
  }

  server.on("/", handleRoot);               // Call the 'handleRoot' function when a client requests URI
"/"
  server.onNotFound(handleNotFound);        // When a client requests an unknown URI (i.e. something
other than "/"), call function "handleNotFound"

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

void loop(void){
  server.handleClient();                    // Listen for HTTP requests from clients
}

void handleRoot() {
  server.send(200, "text/plain", "Hello world!");   // Send HTTP status 200 (Ok) and send some text to
the browser/client
}

void handleNotFound(){
  server.send(404, "text/plain", "404: Not found"); // Send HTTP status 404 (Not Found) when there's no
handler for the URI in the request
}
```

There's a lot of code that's the same as in the Wi-Fi and mDNS examples.
The actual server code is pretty straightforward. First, we create a server instance that listens for HTTP requests on port 80. This is the default port for web servers. In the setup, we tell the server what to do with certain HTTP requests. If the URI '/' is requested, the server should reply with a HTTP status code of 200 (Ok) and then send a response with the words 'Hello world!'. We put the code for generating a response in a separate function, and the we tell the server to execute it when '/' is requested, using the `server.on` function.

We haven't specified what the server should do if the client requests any URI other than '/'. It should

respond with an HTTP status 404 (Not Found) and a message for the user. We put this in a function as well, and use `server.onNotFound` to tell it that it should execute it when it receives a request for a URI that wasn't specified with `server.on` .

Then we start listening for HTTP requests by using `server.begin` .
During the loop, we constantly check if a new HTTP request is received by running `server.handleClient` . If handleClient detects new requests, it will automatically execute the right functions that we specified in the setup.

To test it out, upload the sketch, open a new browser tab, and browse to http://esp8266.local. You should get a webpage saying `Hello world!` . If you try to go to a different page, http://esp8266.local/test, for instance, you should get a 404 error: `404: Not found` .

Turning on and off an LED over Wi-Fi

We can use the web server to serve interactive pages, and to react to certain POST request. In the following example, the ESP8266 hosts a web page with a button. When the button is pressed, the browser sends a POST request to */LED*. When the ESP receives such a POST request on the /LED URI, it will turn on or off the LED, and then redirect the browser back to the home page with the button.

In order to perform this redirect, the ESP has to add a **Location header** to the response, and use a 303 (See Other) HTTP status code.

The button to send the POST request in the browser is part of an HTML form. You have to specify the target URI to send the request to, and the request method, in this case this is "/LED" and POST respectively.

Note that I changed the content type of the response from "text/plain" to "text/html". If you send it as plain text, the browser will display it as text instead of interpreting it as HTML and showing it as a button.

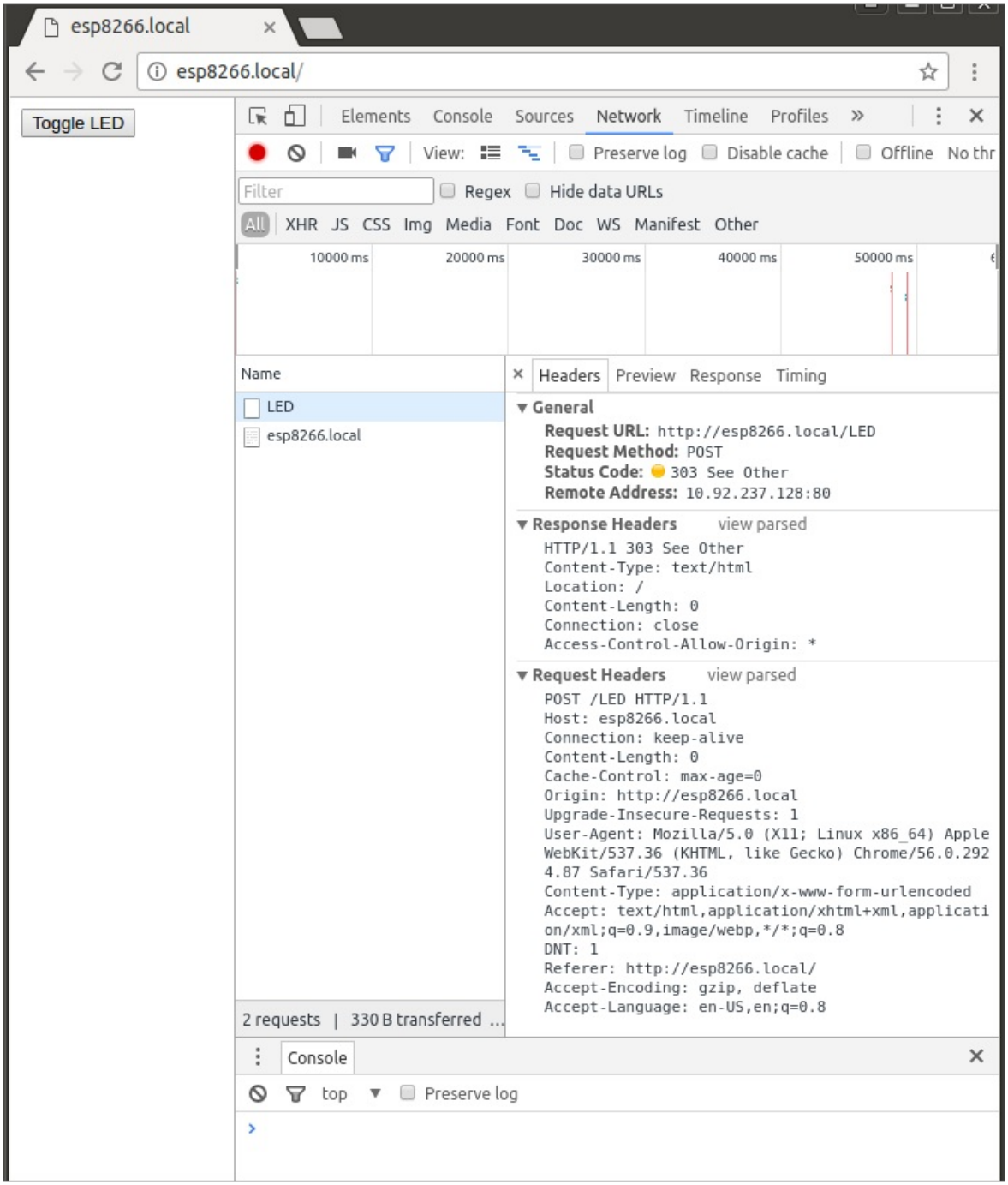
```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

const int led = 2;

void handleRoot();              // function prototypes for HTTP handlers
void handleLED();
void handleNotFound();

void setup(void){
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  pinMode(led, OUTPUT);

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (MDNS.begin("esp8266")) {              // Start the mDNS responder for esp8266.local
    Serial.println("mDNS responder started");
  } else {
    Serial.println("Error setting up MDNS responder!");
  }

  server.on("/", HTTP_GET, handleRoot);     // Call the 'handleRoot' function when a client requests URI
"/"
  server.on("/LED", HTTP_POST, handleLED);  // Call the 'handleLED' function when a POST request is made
```

```
to URI "/LED"
  server.onNotFound(handleNotFound);        // When a client requests an unknown URI (i.e. something
other than "/"), call function "handleNotFound"

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

void loop(void){
  server.handleClient();                    // Listen for HTTP requests from clients
}

void handleRoot() {                         // When URI / is requested, send a web page with a button to
toggle the LED
  server.send(200, "text/html", "<form action=\"/LED\" method=\"POST\"><input type=\"submit\"
value=\"Toggle LED\"></form>");
}

void handleLED() {                          // If a POST request is made to URI /LED
  digitalWrite(led,!digitalRead(led));      // Change the state of the LED
  server.sendHeader("Location","/");        // Add a header to respond with a new location for the
browser to go to the home page again
  server.send(303);                         // Send it back to the browser with an HTTP status 303 (See
Other) to redirect
}

void handleNotFound(){
  server.send(404, "text/plain", "404: Not found"); // Send HTTP status 404 (Not Found) when there's no
handler for the URI in the request
}
```

As you can see, the `server.on` function now takes three parameters: the URI, the request method (GET or POST) and the function to execute.

Connect an LED to GPIO2, and upload the sketch. Then go to http://esp8266.local/ and click the button to turn the LED on or off.

You can open the developer options in Chrome (F12) to check the HTTP request that are made when you click the button: you'll see that it first send a POST request, and then receives a 303 (See Other) HTTP status as a response. The response also has a Location header containing the URI "/", so the browser will send a GET request to the URI of this new location:



If you check the page source (CTRL+U), you can see the simple HTML form that's used:

```
<form action="/LED" method="POST">
    <input type="submit" value="Toggle LED">
</form>
```

Sending data to the ESP using HTTP POST

In the previous example, we sent an empty POST request to the ESP8266. In the previous chapter however, I explained that it's possible to send all kinds of data in the body of the POST request.

In this example, I'll show you how to send a username and a password to the ESP. The ESP will then check if they are correct, and respond to the request with the appropriate page.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

void handleRoot();              // function prototypes for HTTP handlers
void handleLogin();
```

```
void handleNotFound();

void setup(void){
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (MDNS.begin("esp8266")) {              // Start the mDNS responder for esp8266.local
    Serial.println("mDNS responder started");
  } else {
    Serial.println("Error setting up MDNS responder!");
  }

  server.on("/", HTTP_GET, handleRoot);        // Call the 'handleRoot' function when a client requests
URI "/"
  server.on("/login", HTTP_POST, handleLogin); // Call the 'handleLogin' function when a POST request is
made to URI "/login"
  server.onNotFound(handleNotFound);           // When a client requests an unknown URI (i.e. something
other than "/"), call function "handleNotFound"

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

void loop(void){
  server.handleClient();                    // Listen for HTTP requests from clients
}

void handleRoot() {                         // When URI / is requested, send a web page with a button
to toggle the LED
  server.send(200, "text/html", "<form action=\"/login\" method=\"POST\"><input type=\"text\"
name=\"username\" placeholder=\"Username\"></br><input type=\"password\" name=\"password\"
placeholder=\"Password\"></br><input type=\"submit\" value=\"Login\"></form><p>Try 'John Doe' and
'password123' ...</p>");
}

void handleLogin() {                        // If a POST request is made to URI /login
  if( ! server.hasArg("username") || ! server.hasArg("password")
      || server.arg("username") == NULL || server.arg("password") == NULL) { // If the POST request
doesn't have username and password data
    server.send(400, "text/plain", "400: Invalid Request");         // The request is invalid, so send
HTTP status 400
    return;
  }
  if(server.arg("username") == "John Doe" && server.arg("password") == "password123") { // If both the
username and the password are correct
    server.send(200, "text/html", "<h1>Welcome, " + server.arg("username") + "!</h1><p>Login
successful</p>");
  } else {                                                                              // Username and
password don't match
    server.send(401, "text/plain", "401: Unauthorized");
  }
}

void handleNotFound(){
  server.send(404, "text/plain", "404: Not found"); // Send HTTP status 404 (Not Found) when there's no
handler for the URI in the request
}
```

The HTML in handleRoot is:

```
<form action="/login" method="POST">
    <input type="text" name="username" placeholder="Username"></br>
    <input type="password" name="password" placeholder="Password"></br>
    <input type="submit" value="Login">
</form>
<p>
    Try 'John Doe' and 'password123' ...
</p>
```

Upload the sketch and go to http://esp8266.local/, then type 'John Doe' into the username field, and 'password123' into the password field, and click 'Login'. You should get a welcome screen. If you leave on or both of the fields blank, you should get a 400 (Bad Request) error. If you enter a wrong username or password, you should get a 401 (Unauthorized) error.

The data of the POST body can be accessed using `server.arg("key")` , and you can check if a specific key exists using `server.hasArg("key")` . The key name on the ESP8266 corresponds to the name argument in the HTML form on the web page.

When we get a POST request, we first check if the necessary arguments (username and password) are present. If that's not the case, we send a 400 (Invalid Request) status.
Then we check if the credentials match 'John Doe' & 'password123'. If that's the case, we respond with a status 200 (Ok) and a welcome page. If the username and/or password doesn't match, we send a 401 (Unauthorized) status.

Inline functions

In the previous examples, we passed `handleRoot` and `handleNotFound` to the `server.on` function as a parameter (callback function). In some cases however, it's more readable to just write the definition of the function inline, like so:

```
void setup(){
  // ...
  server.onNotFound([](){
    server.send(404, "text/plain", "404: Not found");
  });
}
```

SPI Flash File System

Up until now, we've always included the HTML for our web pages as string literals in our sketch. This makes our code very hard to read, and you'll run out of memory rather quickly.
If you remember the introduction, I mentioned the Serial Peripheral Interface Flash File System, or SPIFFS for short. It's a light-weight file system for microcontrollers with an SPI flash chip. The on-board flash chip of the ESP8266 has plenty of space for your webpages, especially if you have the 1MB, 2MB or 4MB version.

SPIFFS let's you access the flash memory as if it was a normal file system like the one on your computer (but much simpler of course): you can read and write files, create folders ...

The easiest way to learn how to use SPIFFS is to look at some examples. But a file server with no files to serve is pretty pointless, so I'll explain how to upload files to the SPIFFS first.

Uploading files to SPIFFS

To select the right files to upload, you have to place them in a folder called *data*, inside the sketch folder of your project: Open your sketch in the Arduino IDE, and hit CTRL+K. Wait for a file explorer window to open, and create a new folder named *data*. Copy your files over to this folder. (Only use small files like text files or icons. There's not enough space for large photos or videos.)
Next, select all files in the folder (CTRL+A) and check the size of all files combined (don't forget subfolders). Go to the Arduino IDE again, and under Tools > Flash Size, select an option with the right flash size for your board, and a SPIFFS size that is larger than the size of your data folder.
Then upload the sketch. When that's finished, make sure that the Serial Monitor is closed, then open the *Tools* menu, and click *ESP8266 sketch data upload*. If your ESP has auto-reset and auto-program, it should work automatically, if you don't have auto-program, you have to manually enter program mode before uploading the data to SPIFFS. The procedure is exactly the same as entering program mode before uploading a sketch.

If you get an error saying `SPIFFS_write error(-10001): File system is full`, this means that your files are too large to fit into the SPIFFS memory. Select a larger SPIFFS size under Tools > Flash Size, or delete some files.
Even if your computer says that the files are smaller than the selected SPIFFS size, you can still get this error: this has to do with block sizes, and metadata like file and folder names that take up space as well.

If you change the SPIFFS size, you have to reupload your sketch, because when you change the SPIFFS size, the memory location will be different. The program has to know the updated SPIFFS address offset to be able to read the files.

SPIFFS File Server

The following example is a very basic file server: it just takes the URI of the HTTP request, checks if the URI points to a file in the SPIFFS, and if it finds the file, it sends it as a response.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>
#include <FS.h>   // Include the SPIFFS library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

String getContentType(String filename); // convert the file extension to the MIME type
bool handleFileRead(String path);       // send the right file to the client (if it exists)

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
```

```
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (MDNS.begin("esp8266")) {              // Start the mDNS responder for esp8266.local
    Serial.println("mDNS responder started");
  } else {
    Serial.println("Error setting up MDNS responder!");
  }

  SPIFFS.begin();                           // Start the SPI Flash Files System

  server.onNotFound([]() {                              // If the client requests any URI
    if (!handleFileRead(server.uri()))                  // send it if it exists
      server.send(404, "text/plain", "404: Not Found"); // otherwise, respond with a 404 (Not Found)
error
  });

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

void loop(void) {
  server.handleClient();
}

String getContentType(String filename) { // convert the file extension to the MIME type
  if (filename.endsWith(".html")) return "text/html";
  else if (filename.endsWith(".css")) return "text/css";
  else if (filename.endsWith(".js")) return "application/javascript";
  else if (filename.endsWith(".ico")) return "image/x-icon";
  return "text/plain";
}

bool handleFileRead(String path) { // send the right file to the client (if it exists)
  Serial.println("handleFileRead: " + path);
  if (path.endsWith("/")) path += "index.html";         // If a folder is requested, send the index file
  String contentType = getContentType(path);            // Get the MIME type
  if (SPIFFS.exists(path)) {                            // If the file exists
    File file = SPIFFS.open(path, "r");                 // Open it
    size_t sent = server.streamFile(file, contentType); // And send it to the client
    file.close();                                       // Then close the file again
    return true;
  }
  Serial.println("\tFile Not Found");
  return false;                                         // If the file doesn't exist, return false
}
```

As you can see, we don't use `server.on` in this example. Instead, we use `server.onNotFound` : this will match any URI, since we didn't declare any specific URI handlers like in the previous server examples.
When a URI is requested, we call the function `handleFileRead` . This function checks if the URI of the HTTP request is the path to an existing file in the SPIFFS. If that's the case, it sends the file back to the client. If the path doesn't exist, it returns false, and a 404 (Not Found) HTTP status will be sent.

The MIME type for the different files is based on the file extension.
You could add other file types as well. For instance:

```
String getContentType(String filename){
  if(filename.endsWith(".htm")) return "text/html";
  else if(filename.endsWith(".html")) return "text/html";
  else if(filename.endsWith(".css")) return "text/css";
  else if(filename.endsWith(".js")) return "application/javascript";
  else if(filename.endsWith(".png")) return "image/png";
  else if(filename.endsWith(".gif")) return "image/gif";
  else if(filename.endsWith(".jpg")) return "image/jpeg";
  else if(filename.endsWith(".ico")) return "image/x-icon";
  else if(filename.endsWith(".xml")) return "text/xml";
  else if(filename.endsWith(".pdf")) return "application/x-pdf";
  else if(filename.endsWith(".zip")) return "application/x-zip";
  else if(filename.endsWith(".gz")) return "application/x-gzip";
  return "text/plain";
}
```

This example is adapted from the FSBrowser example by Hristo Gochkov.

Compressing files

The ESP8266's flash memory isn't huge, and most text files, like html, css etc. can be compressed by quite a large factor. Modern web browsers accept compressed files as a response, so we'll take

advantage of this by uploading compressed versions of our html and icon files to the SPIFFS, in order to save space and bandwidth.

To do this, we need to add the GNU zip file type to our list of MIME types:

```
String getContentType(String filename){
  if(filename.endsWith(".html")) return "text/html";
  else if(filename.endsWith(".css")) return "text/css";
  else if(filename.endsWith(".js")) return "application/javascript";
  else if(filename.endsWith(".ico")) return "image/x-icon";
  else if(filename.endsWith(".gz")) return "application/x-gzip";
  return "text/plain";
}
```

And we need to change our `handleFileRead` function as well:

```
bool handleFileRead(String path){  // send the right file to the client (if it exists)
  Serial.println("handleFileRead: " + path);
  if(path.endsWith("/")) path += "index.html";           // If a folder is requested, send the index
file
  String contentType = getContentType(path);             // Get the MIME type
  String pathWithGz = path + ".gz";
  if(SPIFFS.exists(pathWithGz) || SPIFFS.exists(path)){  // If the file exists, either as a compressed
archive, or normal
    if(SPIFFS.exists(pathWithGz))                          // If there's a compressed version available
      path += ".gz";                                         // Use the compressed version
    File file = SPIFFS.open(path, "r");                    // Open the file
    size_t sent = server.streamFile(file, contentType);    // Send it to the client
    file.close();                                          // Close the file again
    Serial.println(String("\tSent file: ") + path);
    return true;
  }
  Serial.println(String("\tFile Not Found: ") + path);
  return false;                                          // If the file doesn't exist, return false
}
```

Now, try compressing some of the files to the GNU zip format (.gz), and uploading them to SPIFFS. Or you can just download the new data folder (unzip it first).
Every time a client requests a certain file, the ESP will check if a compressed version is available. If so, it will use that instead of the uncompressed file. The output in the Serial Monitor should look something like this:

```
handleFileRead: /
        Sent file: /index.html.gz
handleFileRead: /main.css
        Sent file: /main.css
handleFileRead: /JavaScript.js
        Sent file: /JavaScript.js
handleFileRead: /folder/JavaScript.js
        Sent file: /folder/JavaScript.js
handleFileRead: /favicon.ico
        Sent file: /favicon.ico.gz
```

It automatically detected that it had to send the compressed versions of index.html and favicon.ico.

Uploading files to the server

There are scenarios where you may want to upload new files to the server from within a browser, without having to connect to the ESP8266 over USB in order to flash a new SPIFFS image.
In this chapter, I'll show you how to use HTML forms and POST requests to upload or edit files to our little ESP server.

Client: HTML form

The easiest way to upload files is by using an HTML form, just like in the first server examples, where we used forms to turn on/off LEDs, and to send the login credentials back to the server. If you choose a file input, you automatically get a file picker, and the browser will send the right POST request to the server, with the file attached.

```
<form method="post" enctype="multipart/form-data">
    <input type="file" name="name">
    <input class="button" type="submit" value="Upload">
</form>
```

Server

In the ESP code, we have to add a handler to our server that handles POST requests to the */upload* URI. When it receives a POST request, it sends a status 200 (OK) back to the client to start receiving the file, and then write it to the SPIFFS. When the file is uploaded successfully, it redirects the client to a success page.
The relevant new code is found in the `setup` and the `handleFileUpload` function.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>
#include <FS.h>   // Include the SPIFFS library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

File fsUploadFile;              // a File object to temporarily store the received file

String getContentType(String filename); // convert the file extension to the MIME type
bool handleFileRead(String path);       // send the right file to the client (if it exists)
void handleFileUpload();                // upload a new file to the SPIFFS

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
    delay(1000);
    Serial.print(++i); Serial.print(' ');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  if (!MDNS.begin("esp8266")) {             // Start the mDNS responder for esp8266.local
    Serial.println("Error setting up MDNS responder!");
  }
  Serial.println("mDNS responder started");

  SPIFFS.begin();                           // Start the SPI Flash Files System

  server.on("/upload", HTTP_GET, []() {                 // if the client requests the upload page
    if (!handleFileRead("/upload.html"))                // send it if it exists
      server.send(404, "text/plain", "404: Not Found"); // otherwise, respond with a 404 (Not Found)
error
```

```
  });

  server.on("/upload", HTTP_POST,                       // if the client posts to the upload page
    [](){ server.send(200); },                          // Send status 200 (OK) to tell the client we
are ready to receive
    handleFileUpload                                    // Receive and save the file
  );

  server.onNotFound([]() {                              // If the client requests any URI
    if (!handleFileRead(server.uri()))                  // send it if it exists
      server.send(404, "text/plain", "404: Not Found"); // otherwise, respond with a 404 (Not Found)
error
  });

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

void loop() {
  server.handleClient();
}

String getContentType(String filename) { // convert the file extension to the MIME type
  if (filename.endsWith(".html")) return "text/html";
  else if (filename.endsWith(".css")) return "text/css";
  else if (filename.endsWith(".js")) return "application/javascript";
  else if (filename.endsWith(".ico")) return "image/x-icon";
  else if (filename.endsWith(".gz")) return "application/x-gzip";
  return "text/plain";
}

bool handleFileRead(String path) { // send the right file to the client (if it exists)
  Serial.println("handleFileRead: " + path);
  if (path.endsWith("/")) path += "index.html";          // If a folder is requested, send the index
file
  String contentType = getContentType(path);            // Get the MIME type
  String pathWithGz = path + ".gz";
  if (SPIFFS.exists(pathWithGz) || SPIFFS.exists(path)) { // If the file exists, either as a compressed
archive, or normal
    if (SPIFFS.exists(pathWithGz))                        // If there's a compressed version available
      path += ".gz";                                      // Use the compressed verion
    File file = SPIFFS.open(path, "r");                   // Open the file
    size_t sent = server.streamFile(file, contentType);   // Send it to the client
    file.close();                                         // Close the file again
    Serial.println(String("\tSent file: ") + path);
    return true;
  }
  Serial.println(String("\tFile Not Found: ") + path);   // If the file doesn't exist, return false
  return false;
}

void handleFileUpload(){ // upload a new file to the SPIFFS
  HTTPUpload& upload = server.upload();
  if(upload.status == UPLOAD_FILE_START){
    String filename = upload.filename;
    if(!filename.startsWith("/")) filename = "/"+filename;
    Serial.print("handleFileUpload Name: "); Serial.println(filename);
    fsUploadFile = SPIFFS.open(filename, "w");            // Open the file for writing in SPIFFS (create
if it doesn't exist)
    filename = String();
  } else if(upload.status == UPLOAD_FILE_WRITE){
    if(fsUploadFile)
      fsUploadFile.write(upload.buf, upload.currentSize); // Write the received bytes to the file
  } else if(upload.status == UPLOAD_FILE_END){
    if(fsUploadFile) {                                    // If the file was successfully created
      fsUploadFile.close();                               // Close the file again
      Serial.print("handleFileUpload Size: "); Serial.println(upload.totalSize);
      server.sendHeader("Location","/success.html");      // Redirect the client to the success page
      server.send(303);
    } else {
      server.send(500, "text/plain", "500: couldn't create file");
    }
  }
}
```

The `handleFileUpload` function just writes the file attached to the POST request to SPIFFS.

If you wan to use other file types as well, you can just add them to the `getContentType` function.

Uploading files

To upload a new file to the ESP, or to update an existing file, just go to http://esp8266.local/upload, click the *Choose File* button, select the file you wish to upload, and click *Upload*. You can now enter the URL into the URL bar, and open the new file.

A note on safety

This example isn't very secure (obviously). Everyone that can connect to the ESP can upload new files, or edit the existing files and insert XSS code, for example. There's also not a lot of error checking/handling, like checking if there's enough space in the SPIFFS to upload a new file, etc.

Advanced example

The code for these SPIFFS server examples comes (for the most part) from an example written by Hristo Gochkov. You can find it under File > Examples > ESP8266WebServer > FSBrowser. It has a web interface for browsing and editing files in your browser, and has some other nice features as well.

Over The Air Updates

Uploading over Serial is fine during development, when you have access to the Serial pins and the USB port. But once your project is finished, and you put it inside an enclosure, it not that easy to upload updates with bug fixes or new features.
A solution to this problem is Over The Air updating, or OTA for short. As the name implies, this technology allows you to upload new code over Wi-Fi, instead of Serial.
The only disadvantage is that you have to explicitly add the code for it to every sketch you upload. You also need a flash chip that is twice the size of your sketch, so it won't work for 512kB boards. (It has to download the new sketch while still running the old code.)
Let's take a look at an example ...

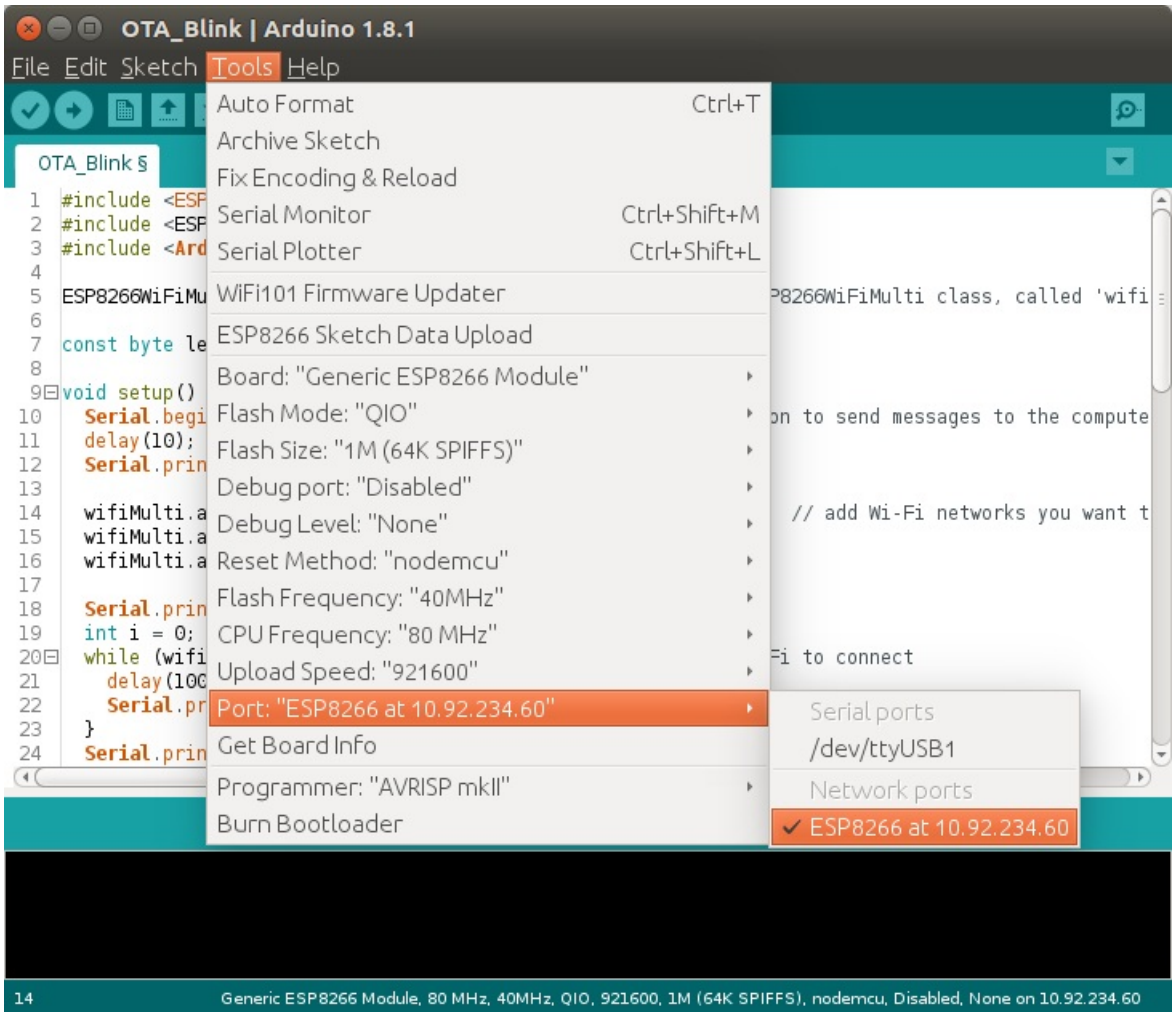
Blink OTA

The following example is basically Blink Without Delay, but with the necessary OTA and Wi-Fi code added as well.

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <ArduinoOTA.h>

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

const byte led = 13;

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer

  ArduinoOTA.setHostname("ESP8266");
  ArduinoOTA.setPassword("esp8266");

  ArduinoOTA.onStart([]() {
    Serial.println("Start");
  });
  ArduinoOTA.onEnd([]() {
    Serial.println("\nEnd");
  });
  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
  });
  ArduinoOTA.onError([](ota_error_t error) {
    Serial.printf("Error[%u]: ", error);
    if (error == OTA_AUTH_ERROR) Serial.println("Auth Failed");
    else if (error == OTA_BEGIN_ERROR) Serial.println("Begin Failed");
    else if (error == OTA_CONNECT_ERROR) Serial.println("Connect Failed");
    else if (error == OTA_RECEIVE_ERROR) Serial.println("Receive Failed");
    else if (error == OTA_END_ERROR) Serial.println("End Failed");
  });
  ArduinoOTA.begin();
  Serial.println("OTA ready");

  pinMode(led, OUTPUT);
  digitalWrite(led, 1);
}

unsigned long previousTime = millis();

const unsigned long interval = 1000;

void loop() {
  ArduinoOTA.handle();
```

```
  unsigned long diff = millis() - previousTime;
  if(diff > interval) {
    digitalWrite(led, !digitalRead(led));  // Change the state of the LED
    previousTime += diff;
  }
}
```

Add your Wi-Fi credentials, and upload the sketch over Serial. Connect an LED (+ resistor) to pin 13. Then restart the IDE (you have to close all windows). Go to Tools > Port, and you should get a new option: Network Ports: ESP8266 at 192.168.1.x. Select it.



Next, change the interval on line 59 from 1000 to 500 milliseconds, and click upload. You should get a password prompt: enter "esp8266". This password is set on line 31, so you can change it if you want to. You can also delete line 31 altogether to use it without a password, but it's not recommended - for obvious security reasons.

The sketch should upload just fine, and once the ESP has reset itself, the LED should blink twice as fast.

Once in a while, you might get an error saying `[ERROR]: No Answer` . If this happens, just enter the password, and try again.

Serial Monitor OTA

Using the Serial Monitor over Wi-Fi is not possible (yet?). When you try to open it, you'll be prompted a password, entering the password won't work, because there is no SSH support to access the ESP's console.

You can use a different program to get debug output from the physical Serial port. On Windows, you can try Portmon. On Linux, you can try GTKTerm (`sudo apt-get install gtkterm`) or Screen (`sudo apt-get install screen` to install, and `screen /dev/ttyUSB0 115200` or `screen /dev/ttyACM0 115200` to run; `CTRL+A, CTRL+D` to exit).

WebSocket communication

Up until now, we've always used links (GET) and HTML forms (POST) to get data from the ESP, or to send data to it. This always resulted in a browser navigation action. There are many situations where you want to send data to the ESP without refreshing the page.

One way to do this is by using AJAX and XMLHTTP requests. The disadvantage is that you have to establish a new TCP connection for every message you send. This adds a load of latency.
WebSocket is a technology that keeps the TCP connection open, so you can constantly send data back and forth between the ESP and the client, with low latency. And since it's TCP, you're sure that the packets will arrive intact.

Controlling RGB LEDs from a web interface using WebSocket

To learn how to use WebSockets, I created this comprehensive example, it uses pretty much everything we've covered so far.

The ESP hosts a webpage with three sliders to set the red, green and blue levels of an RGB LED (or LED strip). There's also a button to turn on a rainbow effect that cycles through the entire color wheel. Color data is transmitted from the browser to the ESP via a WebSocket connection.
You can connect to the ESP directly, using it as an AP, or let the ESP connect to a different AP. You can use mDNS to open the webpage, by browsing to http://esp8266.local.
All files are stored in the ESP's SPIFFS, and you can upload new files, or update files via a web interface. You can also use the OTA service to upload new firmware (sketches) over Wi-Fi.

Improving readability

When dealing with large and complicated programs, it's a good idea to make abstraction of some things, and create functions with a descriptive name instead of endless lines of meaningless code.

Even if you have lots of comments in your code, it'll be very hard to preserve an overview. Using functions will greatly improve the readability of your code.
So just split up the code into different parts and move all pieces to functions at the bottom of your sketch, or even to different files.

In the following example, the setup was very long and cluttered, so I split it up into several different functions: one to connect to the Wi-Fi, one to start the OTA update service, one to start the SPIFFS ... and so on.

Downloading WebSockets for Arduino

We'll be using the arduinoWebSockets library by Links2004. Download it from GitHub and install it. (Sketch > Include Library > Add .ZIP Library...)

Libraries, constants and globals

At the top of the sketch we'll include the necessary libraries, create some global server and file objects like in the previous examples, and some constants for the host name, AP ssid, passwords, LED pins ...

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <ArduinoOTA.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>
#include <FS.h>
#include <WebSocketsServer.h>

ESP8266WiFiMulti wifiMulti;       // Create an instance of the ESP8266WiFiMulti class, called
'wifiMulti'

ESP8266WebServer server(80);       // Create a webserver object that listens for HTTP request on port 80
WebSocketsServer webSocket(81);    // create a websocket server on port 81

File fsUploadFile;                 // a File variable to temporarily store the received file

const char *ssid = "ESP8266 Access Point"; // The name of the Wi-Fi network that will be created
const char *password = "thereisnospoon";   // The password required to connect to it, leave blank for an
open network

const char *OTAName = "ESP8266";           // A name and a password for the OTA service
const char *OTAPassword = "esp8266";
```

```
#define LED_RED     15            // specify the pins with an RGB LED connected
#define LED_GREEN   12
#define LED_BLUE    13

const char* mdnsName = "esp8266"; // Domain name for the mDNS responder
```

You should already be familiar with most of this code. The only new part is the WebSocket server library that is included, and the WebSocket server object, but this shouldn't be a problem.

Setup

```
void setup() {
  pinMode(LED_RED, OUTPUT);    // the pins with LEDs connected are outputs
  pinMode(LED_GREEN, OUTPUT);
  pinMode(LED_BLUE, OUTPUT);

  Serial.begin(115200);        // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println("\r\n");

  startWiFi();                 // Start a Wi-Fi access point, and try to connect to some given access
points. Then wait for either an AP or STA connection

  startOTA();                  // Start the OTA service

  startSPIFFS();               // Start the SPIFFS and list all contents

  startWebSocket();            // Start a WebSocket server

  startMDNS();                 // Start the mDNS responder

  startServer();               // Start a HTTP server with a file read handler and an upload handler

}
```

As you can see, the setup is now much more condensed and gives a much better overview of what it's doing. To understand the program, you don't have to know each individual step that is required to connect to a Wi-Fi network, it's enough to know that it *will* connect to a Wi-Fi network, because that's what the startWiFi function does.

Loop

```
bool rainbow = false;             // The rainbow effect is turned off on startup

unsigned long prevMillis = millis();
int hue = 0;

void loop() {
  webSocket.loop();                           // constantly check for websocket events
  server.handleClient();                      // run the server
  ArduinoOTA.handle();                        // listen for OTA events

  if(rainbow) {                               // if the rainbow effect is turned on
    if(millis() > prevMillis + 32) {
      if(++hue == 360)                        // Cycle through the color wheel (increment by one degree
every 32 ms)
        hue = 0;
      setHue(hue);                            // Set the RGB LED to the right color
      prevMillis = millis();
    }
  }
}
```

Same goes for the loop: most of the work is done by the first three functions that handle the WebSocket communication, HTTP requests and OTA updates. When such an event happens, the appropriate handler functions will be executed. These are defined elsewhere.

The second part is the rainbow effect. If it is turned on, it cycles through the color wheel and sets the color to the RGB LED.
If you don't understand why I use millis(), you can take a look at the Blink Without Delay example.

Setup functions

```
void startWiFi() { // Start a Wi-Fi access point, and try to connect to some given access points. Then
wait for either an AP or STA connection
  WiFi.softAP(ssid, password);             // Start the access point
  Serial.print("Access Point \"");
  Serial.print(ssid);
```

```
  Serial.println("\" started\r\n");

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting");
  while (wifiMulti.run() != WL_CONNECTED && WiFi.softAPgetStationNum() < 1) {  // Wait for the Wi-Fi to
connect
    delay(250);
    Serial.print('.');
  }
  Serial.println("\r\n");
  if(WiFi.softAPgetStationNum() == 0) {      // If the ESP is connected to an AP
    Serial.print("Connected to ");
    Serial.println(WiFi.SSID());             // Tell us what network we're connected to
    Serial.print("IP address:\t");
    Serial.print(WiFi.localIP());            // Send the IP address of the ESP8266 to the computer
  } else {                                   // If a station is connected to the ESP SoftAP
    Serial.print("Station connected to ESP8266 AP");
  }
  Serial.println("\r\n");
}

void startOTA() { // Start the OTA service
  ArduinoOTA.setHostname(OTAName);
  ArduinoOTA.setPassword(OTAPassword);

  ArduinoOTA.onStart([]() {
    Serial.println("Start");
    digitalWrite(LED_RED, 0);    // turn off the LEDs
    digitalWrite(LED_GREEN, 0);
    digitalWrite(LED_BLUE, 0);
  });
  ArduinoOTA.onEnd([]() {
    Serial.println("\r\nEnd");
  });
  ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
    Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
  });
  ArduinoOTA.onError([](ota_error_t error) {
    Serial.printf("Error[%u]: ", error);
    if (error == OTA_AUTH_ERROR) Serial.println("Auth Failed");
    else if (error == OTA_BEGIN_ERROR) Serial.println("Begin Failed");
    else if (error == OTA_CONNECT_ERROR) Serial.println("Connect Failed");
    else if (error == OTA_RECEIVE_ERROR) Serial.println("Receive Failed");
    else if (error == OTA_END_ERROR) Serial.println("End Failed");
  });
  ArduinoOTA.begin();
  Serial.println("OTA ready\r\n");
}

void startSPIFFS() { // Start the SPIFFS and list all contents
  SPIFFS.begin();                            // Start the SPI Flash File System (SPIFFS)
  Serial.println("SPIFFS started. Contents:");
  {
    Dir dir = SPIFFS.openDir("/");
    while (dir.next()) {                     // List the file system contents
      String fileName = dir.fileName();
      size_t fileSize = dir.fileSize();
      Serial.printf("\tFS File: %s, size: %s\r\n", fileName.c_str(), formatBytes(fileSize).c_str());
    }
    Serial.printf("\n");
  }
}

void startWebSocket() { // Start a WebSocket server
  webSocket.begin();                         // start the websocket server
  webSocket.onEvent(webSocketEvent);         // if there's an incomming websocket message, go to
function 'webSocketEvent'
  Serial.println("WebSocket server started.");
}

void startMDNS() { // Start the mDNS responder
  MDNS.begin(mdnsName);                        // start the multicast domain name server
  Serial.print("mDNS responder started: http://");
  Serial.print(mdnsName);
  Serial.println(".local");
}

void startServer() { // Start a HTTP server with a file read handler and an upload handler
  server.on("/edit.html",  HTTP_POST, []() {  // If a POST request is sent to the /edit.html address,
    server.send(200, "text/plain", "");
  }, handleFileUpload);                      // go to 'handleFileUpload'

  server.onNotFound(handleNotFound);         // if someone requests any other file or page, go to
function 'handleNotFound'
                                             // and check if the file exists

  server.begin();                            // start the HTTP server
```

```
  Serial.println("HTTP server started.");
}
```

These are the function definitions of the functions used in the setup. Nothing new here, apart from the startWebSocket function. You just have to start the WebSocket server using the begin method, and then give it a callback function that is executed when the ESP receives a WebSocket message.

Server handlers

This is the code that is executed on certain server-related events, like when an HTTP request is received, when a file is being uploaded, when there's an incoming WebSocket message ... etc.

```
void handleNotFound(){ // if the requested file or page doesn't exist, return a 404 not found error
  if(!handleFileRead(server.uri())){          // check if the file exists in the flash memory (SPIFFS),
if so, send it
    server.send(404, "text/plain", "404: File Not Found");
  }
}

bool handleFileRead(String path) { // send the right file to the client (if it exists)
  Serial.println("handleFileRead: " + path);
  if (path.endsWith("/")) path += "index.html";          // If a folder is requested, send the index
file
  String contentType = getContentType(path);             // Get the MIME type
  String pathWithGz = path + ".gz";
  if (SPIFFS.exists(pathWithGz) || SPIFFS.exists(path)) { // If the file exists, either as a compressed
archive, or normal
    if (SPIFFS.exists(pathWithGz))                         // If there's a compressed version available
      path += ".gz";                                         // Use the compressed verion
    File file = SPIFFS.open(path, "r");                    // Open the file
    size_t sent = server.streamFile(file, contentType);    // Send it to the client
    file.close();                                          // Close the file again
    Serial.println(String("\tSent file: ") + path);
    return true;
  }
  Serial.println(String("\tFile Not Found: ") + path);   // If the file doesn't exist, return false
  return false;
}

void handleFileUpload(){ // upload a new file to the SPIFFS
  HTTPUpload& upload = server.upload();
  String path;
  if(upload.status == UPLOAD_FILE_START){
    path = upload.filename;
    if(!path.startsWith("/")) path = "/"+path;
    if(!path.endsWith(".gz")) {                          // The file server always prefers a compressed
version of a file
      String pathWithGz = path+".gz";                    // So if an uploaded file is not compressed,
the existing compressed
      if(SPIFFS.exists(pathWithGz))                      // version of that file must be deleted (if it
exists)
        SPIFFS.remove(pathWithGz);
    }
    Serial.print("handleFileUpload Name: "); Serial.println(path);
    fsUploadFile = SPIFFS.open(path, "w");            // Open the file for writing in SPIFFS (create if
it doesn't exist)
    path = String();
  } else if(upload.status == UPLOAD_FILE_WRITE){
    if(fsUploadFile)
      fsUploadFile.write(upload.buf, upload.currentSize); // Write the received bytes to the file
  } else if(upload.status == UPLOAD_FILE_END){
    if(fsUploadFile) {                                    // If the file was successfully created
      fsUploadFile.close();                               // Close the file again
      Serial.print("handleFileUpload Size: "); Serial.println(upload.totalSize);
      server.sendHeader("Location","/success.html");      // Redirect the client to the success page
      server.send(303);
    } else {
      server.send(500, "text/plain", "500: couldn't create file");
    }
  }
}

void webSocketEvent(uint8_t num, WStype_t type, uint8_t * payload, size_t lenght) { // When a WebSocket
message is received
  switch (type) {
    case WStype_DISCONNECTED:             // if the websocket is disconnected
      Serial.printf("[%u] Disconnected!\n", num);
      break;
    case WStype_CONNECTED: {              // if a new websocket connection is established
        IPAddress ip = webSocket.remoteIP(num);
        Serial.printf("[%u] Connected from %d.%d.%d.%d url: %s\n", num, ip[0], ip[1], ip[2], ip[3],
payload);
        rainbow = false;                  // Turn rainbow off when a new connection is established
      }
      break;
    case WStype_TEXT:                     // if new text data is received
      Serial.printf("[%u] get Text: %s\n", num, payload);
```

```
      if (payload[0] == '#') {            // we get RGB data
        uint32_t rgb = (uint32_t) strtol((const char *) &payload[1], NULL, 16);   // decode rgb data
        int r = ((rgb >> 20) & 0x3FF);                     // 10 bits per color, so R: bits 20-29
        int g = ((rgb >> 10) & 0x3FF);                     // G: bits 10-19
        int b =          rgb & 0x3FF;                      // B: bits  0-9

        analogWrite(LED_RED,   r);                         // write it to the LED output pins
        analogWrite(LED_GREEN, g);
        analogWrite(LED_BLUE,  b);
      } else if (payload[0] == 'R') {                      // the browser sends an R when the rainbow
effect is enabled
        rainbow = true;
      } else if (payload[0] == 'N') {                      // the browser sends an N when the rainbow
effect is disabled
        rainbow = false;
      }
      break;
  }
}
```

Again, most of the code is adapted from the previous examples, only the WebSocket part is new.

There are different types of WebSocket messages, but we're only interested in the text type, because the JavaScript code at the client side sends the color data in text format, as a hexadecimal number, starting with a '#' sign.
Each color is a 10-bit number, so in total, it gives us a 30-bit number for the RGB value.

When the rainbow function is enabled, JavaScript sends an 'R' character, and when it's disabled, it sends a 'N' character.

Let's take a look at the HTML and JavaScript code as well:

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>LED Control</title>
  <link href='https://fonts.googleapis.com/css?family=Roboto:300' rel='stylesheet' type='text/css'>
  <link href='main.css' rel='stylesheet' type='text/css'>
  <link rel="apple-touch-icon" sizes="180x180" href="/apple-touch-icon-180x180.png">
  <link rel="icon" type="image/png" sizes="144x144"  href="/favicon-144x144.png">
  <link rel="icon" type="image/png" sizes="48x48" href="/favicon.ico">
  <link rel="manifest" href="/manifest.json">
  <meta name="theme-color" content="#00878f">
  <meta content='width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0'
name='viewport'>
  <script src="WebSocket.js" type="text/javascript"></script>
</head>

<body>
  <center>
    <header>
      <h1>LED Control</h1>
    </header>
    <div>
      <table>
        <tr>
          <td style="width:14.4px; text-align: right">R: </td>
          <td><input class="enabled" id="r" type="range" min="0" max="1023" step="1"
oninput="sendRGB();" value="0"></td>
        </tr>
        <tr>
          <td style="width:14.4px; text-align: right">G: </td>
          <td><input class="enabled" id="g" type="range" min="0" max="1023" step="1"
oninput="sendRGB();" value="0"></td>
        </tr>
        <tr>
          <td style="width:14.4px; text-align: right">B: </td>
          <td><input class="enabled" id="b" type="range" min="0" max="1023" step="1"
oninput="sendRGB();" value="0"></td>
        </tr>
      </table>
      <p style="margin:8px 0px">
        <button id="rainbow" class="button" style="background-color:#999"
onclick="rainbowEffect();">Rainbow</button>
      </p>
    </div>
  </center>
</body>
</html>
```

There's really not much to it, just 3 sliders and a button linked to JavaScript functions.

JavaScript

```
var rainbowEnable = false;
var connection = new WebSocket('ws://' + location.hostname + ':81/', ['arduino']);
connection.onopen = function () {
  connection.send('Connect ' + new Date());
};
connection.onerror = function (error) {
  console.log('WebSocket Error ', error);
};
connection.onmessage = function (e) {
  console.log('Server: ', e.data);
};
connection.onclose = function () {
  console.log('WebSocket connection closed');
};

function sendRGB () {
  var r = document.getElementById('r').value** 2 / 1023;
  var g = document.getElementById('g').value** 2 / 1023;
  var b = document.getElementById('b').value** 2 / 1023;

  var rgb = r << 20 | g << 10 | b;
  var rgbstr = '#' + rgb.toString(16);
  console.log('RGB: ' + rgbstr);
  connection.send(rgbstr);
}

function rainbowEffect () {
  rainbowEnable = ! rainbowEnable;
  if (rainbowEnable) {
    connection.send("R");
    document.getElementById('rainbow').style.backgroundColor = '#00878F';
    document.getElementById('r').className = 'disabled';
    document.getElementById('g').className = 'disabled';
    document.getElementById('b').className = 'disabled';
    document.getElementById('r').disabled = true;
    document.getElementById('g').disabled = true;
    document.getElementById('b').disabled = true;
  } else {
    connection.send("N");
    document.getElementById('rainbow').style.backgroundColor = '#999';
    document.getElementById('r').className = 'enabled';
    document.getElementById('g').className = 'enabled';
    document.getElementById('b').className = 'enabled';
    document.getElementById('r').disabled = false;
    document.getElementById('g').disabled = false;
    document.getElementById('b').disabled = false;
    sendRGB();
  }
}
```

We just create a WebSocket connection object to send data to the ESP.
Then every time a slider is moved, we take the values of the three sliders and we square the color values to get a smoother and more natural curve. We then combine them into a 30-bit number (10 bits per color). Finally, the RGB value gets converted to a hexadecimal string, a '#' is added, and it's sent to the ESP.
When the rainbow button is pressed, the sliders are disabled, and an 'R' is sent to the ESP. When the rainbow button is pressed again, the sliders are enabled, and an 'N' is sent.

Helper functions

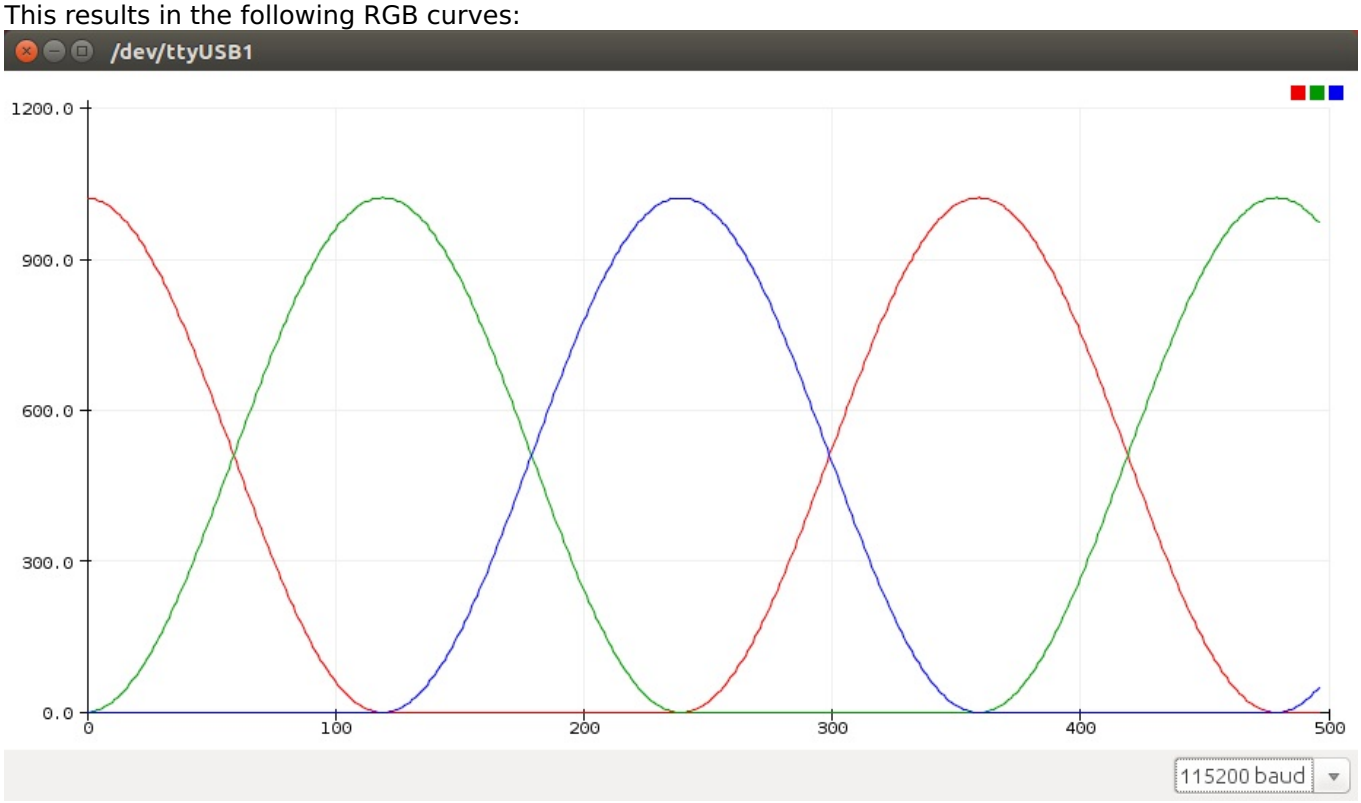
Back to the ESP8266 Arduino code again. We need some other functions as well, to convert bytes to KB and MB, to determine file types based on file extensions and to convert a hue angle to RGB values.

```
String formatBytes(size_t bytes) { // convert sizes in bytes to KB and MB
  if (bytes < 1024) {
    return String(bytes) + "B";
  } else if (bytes < (1024 * 1024)) {
    return String(bytes / 1024.0) + "KB";
  } else if (bytes < (1024 * 1024 * 1024)) {
    return String(bytes / 1024.0 / 1024.0) + "MB";
  }
}

String getContentType(String filename) { // determine the filetype of a given filename, based on the
extension
  if (filename.endsWith(".html")) return "text/html";
  else if (filename.endsWith(".css")) return "text/css";
  else if (filename.endsWith(".js")) return "application/javascript";
  else if (filename.endsWith(".ico")) return "image/x-icon";
  else if (filename.endsWith(".gz")) return "application/x-gzip";
  return "text/plain";
}
```

```
void setHue(int hue) { // Set the RGB LED to a given hue (color) (0° = Red, 120° = Green, 240° = Blue)
  hue %= 360;                  // hue is an angle between 0 and 359°
  float radH = hue*3.142/180;  // Convert degrees to radians
  float rf, gf, bf;

  if(hue>=0 && hue<120){       // Convert from HSI color space to RGB
    rf = cos(radH*3/4);
    gf = sin(radH*3/4);
    bf = 0;
  } else if(hue>=120 && hue<240){
    radH -= 2.09439;
    gf = cos(radH*3/4);
    bf = sin(radH*3/4);
    rf = 0;
  } else if(hue>=240 && hue<360){
    radH -= 4.188787;
    bf = cos(radH*3/4);
    rf = sin(radH*3/4);
    gf = 0;
  }
  int r = rf*rf*1023;
  int g = gf*gf*1023;
  int b = bf*bf*1023;

  analogWrite(LED_RED,   r);    // Write the right color to the LED output pins
  analogWrite(LED_GREEN, g);
  analogWrite(LED_BLUE,  b);
}
```

To convert from hue to RGB, we use sines and cosines, because the sum of their squares is always one, so the total intensity will always be more or less the same.

This results in the following RGB curves:



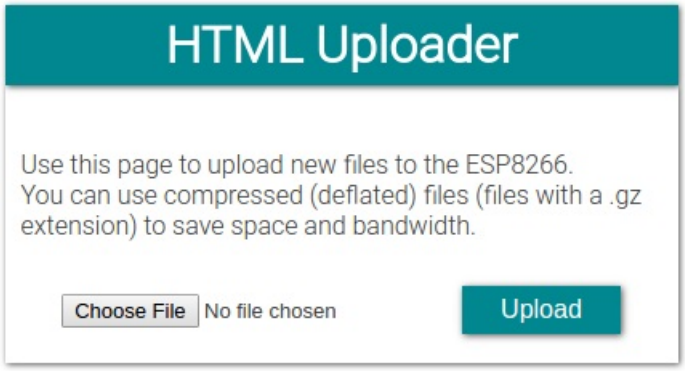
Using the example

Download the example from GitHub and open it in the Arduino IDE. Then add your Wi-Fi credentials (lines 83-85).
Connect an RGB LED with red to pin 15, green to pin 12 and blue to pin 13. Don't forget the current limiting resistors!
Select the SPIFFS size (64KB should be enough, but if you want to upload more files later, you should set it higher). Upload the sketch over Serial, and then upload the SPIFFS files using Tools > ESP8266 Sketch Data Upload.

Wait for it to connect to a Wi-Fi network, or connect to the ESP8266 Access Point using the password 'thereisnospoon', and go to http://esp8266.local. You should get a page that looks like this:

Use the sliders to adjust the color levels of the LED, and press the Rainbow button to enable the rainbow effect.

If you go to http://esp8266.local/edit.html, you can upload or update files:



Network Time Protocol

There are many applications where you want to know the time. In a normal Arduino project, you would have to get a RTC module, set the right time, sacrifice some Arduino pins for communication ... And when the RTC battery runs out, you have to replace it.

On the ESP8266, all you need is an Internet connection: you can just ask a time server what time it is. To do this, the Network Time Protocol (**NTP**) is used.

In the previous examples (HTTP, WebSockets) we've only used TCP connections, but NTP is based on **UDP**. There are a couple of differences, but it's really easy to use, thanks to the great libraries that come with the ESP8266 Arduino Core.

The main difference between TCP and UDP is that TCP needs a connection to send messages: First a handshake is sent by the client, the server responds, and a connection is established, and the client can send its messages. After the client has received the response of the server, the connection is closed (except when using WebSockets). To send a new message, the client has to open a new connection to the server first. This introduces latency and overhead.

UDP doesn't use a connection, a client can just send a message to the server directly, and the server can just send a response message back to the client when it has finished processing. There is, however, no guarantee that the messages will arrive at their destination, and there's no way to know whether they arrived or not (without sending an acknowledgement, of course). This means that we can't halt the program to wait for a response, because the request or response packet could have been lost on the Internet, and the ESP8266 will enter an infinite loop.

Instead of waiting for a response, we just send multiple requests, with a fixed interval between two requests, and just regularly check if a response has been received.

Getting the time

Let's take a look at an example that uses UDP to request the time from a NTP server.

Libraries, constants and globals

```
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <WiFiUdp.h>

ESP8266WiFiMulti wifiMulti;      // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

WiFiUDP UDP;                     // Create an instance of the WiFiUDP class to send and receive

IPAddress timeServerIP;          // time.nist.gov NTP server address
const char* NTPServerName = "time.nist.gov";

const int NTP_PACKET_SIZE = 48;  // NTP time stamp is in the first 48 bytes of the message

byte NTPBuffer[NTP_PACKET_SIZE]; // buffer to hold incoming and outgoing packets
```

To use UDP, we have to include the WiFiUdp library, and create a UDP object. We'll also need to allocate memory for a buffer to store the UDP packets. For NTP, we need a buffer of 48 bytes long.
To know where to send the UDP packets to, we need the hostname of the NTP server, this is time.nist.gov.

Setup

```
void setup() {
  Serial.begin(115200);          // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println("\r\n");

  startWiFi();                   // Try to connect to some given access points. Then wait for a
connection

  startUDP();

  if(!WiFi.hostByName(NTPServerName, timeServerIP)) { // Get the IP address of the NTP server
    Serial.println("DNS lookup failed. Rebooting.");
    Serial.flush();
    ESP.reset();
  }
```

```
  Serial.print("Time server IP:\t");
  Serial.println(timeServerIP);

  Serial.println("\r\nSending NTP request ...");
  sendNTPpacket(timeServerIP);
}
```

In the setup, we just start our Serial and Wi-Fi, as usual, and we start UDP as well. We'll look at the implementation of this function later.
We need the IP address of the NTP server, so we perform a DNS lookup with the server's hostname. There's not much we can do without the IP address of the time server, so if the lookup fails, reboot the ESP.
If we do get an IP, send the first NTP request, and enter the loop.

Loop

```
unsigned long intervalNTP = 60000; // Request NTP time every minute
unsigned long prevNTP = 0;
unsigned long lastNTPResponse = millis();
uint32_t timeUNIX = 0;

unsigned long prevActualTime = 0;

void loop() {
  unsigned long currentMillis = millis();

  if (currentMillis - prevNTP > intervalNTP) { // If a minute has passed since last NTP request
    prevNTP = currentMillis;
    Serial.println("\r\nSending NTP request ...");
    sendNTPpacket(timeServerIP);               // Send an NTP request
  }

  uint32_t time = getTime();                   // Check if an NTP response has arrived and get the
(UNIX) time
  if (time) {                                  // If a new timestamp has been received
    timeUNIX = time;
    Serial.print("NTP response:\t");
    Serial.println(timeUNIX);
    lastNTPResponse = currentMillis;
  } else if ((currentMillis - lastNTPResponse) > 3600000) {
    Serial.println("More than 1 hour since last NTP response. Rebooting.");
    Serial.flush();
    ESP.reset();
  }

  uint32_t actualTime = timeUNIX + (currentMillis - lastNTPResponse)/1000;
  if (actualTime != prevActualTime && timeUNIX != 0) { // If a second has passed since last print
    prevActualTime = actualTime;
    Serial.printf("\rUTC time:\t%d:%d:%d   ", getHours(actualTime), getMinutes(actualTime),
getSeconds(actualTime));
  }
}
```

The first part of the loop sends a new NTP request to the time server every minute. This is based on Blink Without Delay.
Then we call the getTime function to check if we've got a new response from the server. If this is the case, we update the timeUNIX variable with the new timestamp from the server.
If we don't get any responses for an hour, then there's something wrong, so we reboot the ESP.
The last part prints the actual time. The actual time is just the last NTP time plus the time since we received that NTP message.

Setup functions

Nothing special here, just a function to connect to Wi-Fi, and a new function to start listening for UDP messages on port 123.

```
void startWiFi() { // Try to connect to some given access points. Then wait for a connection
  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting");
  while (wifiMulti.run() != WL_CONNECTED) {  // Wait for the Wi-Fi to connect
    delay(250);
    Serial.print('.');
  }
  Serial.println("\r\n");
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());             // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.print(WiFi.localIP());            // Send the IP address of the ESP8266 to the computer
```

```
  Serial.println("\r\n");
}

void startUDP() {
  Serial.println("Starting UDP");
  UDP.begin(123);                          // Start listening for UDP messages on port 123
  Serial.print("Local port:\t");
  Serial.println(UDP.localPort());
  Serial.println();
}
```

Helper functions

```
uint32_t getTime() {
  if (UDP.parsePacket() == 0) { // If there's no response (yet)
    return 0;
  }
  UDP.read(NTPBuffer, NTP_PACKET_SIZE); // read the packet into the buffer
  // Combine the 4 timestamp bytes into one 32-bit number
  uint32_t NTPTime = (NTPBuffer[40] << 24) | (NTPBuffer[41] << 16) | (NTPBuffer[42] << 8) |
NTPBuffer[43];
  // Convert NTP time to a UNIX timestamp:
  // Unix time starts on Jan 1 1970. That's 2208988800 seconds in NTP time:
  const uint32_t seventyYears = 2208988800UL;
  // subtract seventy years:
  uint32_t UNIXTime = NTPTime - seventyYears;
  return UNIXTime;
}

void sendNTPpacket(IPAddress& address) {
  memset(NTPBuffer, 0, NTP_PACKET_SIZE);  // set all bytes in the buffer to 0
  // Initialize values needed to form NTP request
  NTPBuffer[0] = 0b11100011;   // LI, Version, Mode
  // send a packet requesting a timestamp:
  UDP.beginPacket(address, 123); // NTP requests are to port 123
  UDP.write(NTPBuffer, NTP_PACKET_SIZE);
  UDP.endPacket();
}

inline int getSeconds(uint32_t UNIXTime) {
  return UNIXTime % 60;
}

inline int getMinutes(uint32_t UNIXTime) {
  return UNIXTime / 60 % 60;
}

inline int getHours(uint32_t UNIXTime) {
  return UNIXTime / 3600 % 24;
}
```

In the getTime function, we first try to parse the UDP packet. If there's no packet available, the function just returns 0. If there is a UDP packet available however, read it into the buffer. The NTP timestamp is 32 bits or 4 bytes wide, so we combine these bytes into one long number. This number is the number of seconds since Jan 1, 1900, 00:00:00, but most applications use UNIX time, the number of seconds since Jan 1, 1970, 00:00:00 (UNIX epoch). To convert from NTP time to UNIX time, we just subtract 70 years worth of seconds.

To request the time from the NTP server, you have to send a certain sequence of 48 bytes. We don't need any fancy features, so just set the first byte to request the time, and leave all other 47 bytes zero. To actually send the packet, you have to start the packet, specifying the IP address of the server, and the NTP port number, port 123. Then just write the buffer to the packet, and send it with endPacket.

The last three functions are just some simple math to convert seconds to hours, minutes and seconds.

Using the example

Enter your Wi-Fi credentials on lines 79-81, and hit upload. If you have a working Internet connection, you should get an output that looks like this:

```
Connecting
.........

Connected to Wi-Fi SSID
IP address:     192.168.1.2

Starting UDP
Local port:     123

Time server IP: 216.229.0.179
```

```
Sending NTP request ...
NTP response:   1488378061
UTC time:       14:21:53
Sending NTP request ...
NTP response:   1488378114
UTC time:       14:22:53
Sending NTP request ...
NTP response:   1488378174
UTC time:       14:23:53
Sending NTP request ...
NTP response:   1488378234
UTC time:       14:24:53
Sending NTP request ...
NTP response:   1488378294
UTC time:       14:25:53
...
```

You should see the time update every second, and `Sending NTP request ...` should show up every minute. If you don't have an Internet connection, the DNS lookup of the time server will fail:

```
Connecting
.........

Connected to Wi-Fi SSID
IP address:     192.168.1.2

Starting UDP
Local port:     123

DNS lookup failed. Rebooting.

 ets Jan  8 2013,rst cause:2, boot mode:(3,6)
```

If your connection is not reliable, or if there's heavy traffic, you might encounter some dropped packets:

```
Sending NTP request ...
NTP response:   1488378780
UTC time:       14:33:54
Sending NTP request ...
UTC time:       14:34:54
Sending NTP request ...
NTP response:   1488378895
UTC time:       14:35:0
```

As you can see, the ESP never received a response to the second NTP request. That's not really an issue, as long as at least some packets make it through.

Local time and daylight savings

An NTP server returns the UTC time. If you want local time, you have to compensate for your time zone and daylight savings. For example, if you want CET (Central European Time), you have to add 3600 to the UNIX time during winter, (3600 s = 1 h), and 7200 during summer (DST).

Data logging

A common use for IoT devices like the ESP8266 is monitoring sensors. Using the code in the previous example, we can request the time, and save some sensor values to a file. If we run a server as well, we can show this data in a pretty graph in a webpage.

Temperature logger

In the following example, we'll use a DS18S20 temperature sensor to log the temperature over time and save it to the SPIFFS. It can then be displayed in a graph in the browser.

Installing libraries

First, download the Dallas Temperature library by Miles Burton and the OneWire library by Jim Studt: Go to Sketch > Include Library ... > Manage Libraries and search for 'Dallas Temperature' and 'OneWire' (make sure you download the correct version).

Hardware

Connect the ground of the DS18S20 temperature sensor (pin 1) to the ground of the ESP, connect the data pin (pin 2) to GPIO5, and V_{CC} (pin 3) to the 3.3V of the ESP. Finally, connect a 4k7Ω resistor between the data pin and V_{CC}.

Libraries, constants and globals

```
#include <OneWire.h>
#include <DallasTemperature.h>
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <WiFiUdp.h>
#include <ArduinoOTA.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>
#include <FS.h>

#define ONE_HOUR 3600000UL

#define TEMP_SENSOR_PIN 5

OneWire oneWire(TEMP_SENSOR_PIN);        // Set up a OneWire instance to communicate with OneWire
devices

DallasTemperature tempSensors(&oneWire); // Create an instance of the temperature sensor class

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

File fsUploadFile;                                    // a File variable to temporarily store the
received file

ESP8266WiFiMulti wifiMulti;    // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

const char *OTAName = "ESP8266";         // A name and a password for the OTA service
const char *OTAPassword = "esp8266";

const char* mdnsName = "esp8266";        // Domain name for the mDNS responder

WiFiUDP UDP;                    // Create an instance of the WiFiUDP class to send and receive UDP
messages

IPAddress timeServerIP;         // The time.nist.gov NTP server's IP address
const char* ntpServerName = "time.nist.gov";

const int NTP_PACKET_SIZE = 48;          // NTP time stamp is in the first 48 bytes of the message

byte packetBuffer[ NTP_PACKET_SIZE];     // A buffer to hold incoming and outgoing packets
```

The only new things here are the OneWire and DallasTemperature libraries, to get the temperature from the sensor.

Setup

```
void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
```

```
  delay(10);
  Serial.println("\r\n");

  tempSensors.setWaitForConversion(false); // Don't block the program while the temperature sensor is
reading
  tempSensors.begin();                     // Start the temperature sensor

  if (tempSensors.getDeviceCount() == 0) {
    Serial.printf("No DS18x20 temperature sensor found on pin %d. Rebooting.\r\n", TEMP_SENSOR_PIN);
    Serial.flush();
    ESP.reset();
  }

  startWiFi();                 // Start a Wi-Fi access point, and try to connect to some given access
points. Then wait for either an AP or STA connection

  startOTA();                  // Start the OTA service

  startSPIFFS();               // Start the SPIFFS and list all contents

  startMDNS();                 // Start the mDNS responder

  startServer();               // Start a HTTP server with a file read handler and an upload handler

  startUDP();                  // Start listening for UDP messages to port 123

  WiFi.hostByName(ntpServerName, timeServerIP); // Get the IP address of the NTP server
  Serial.print("Time server IP:\t");
  Serial.println(timeServerIP);

  sendNTPpacket(timeServerIP);
}
```

In the setup, there's not much new either, we just start the temperature sensor, and check if we can communicate with it. If no temperature sensor is found, the ESP resets.

Getting the temperature from the sensor may take some time (up to 750ms). We don't want our loop to take longer than a couple of milliseconds, so we can't wait 750ms. If we did, the HTTP server etc. would start to misbehave.
The solution is to request the temperature first. The sensor will then start reading the analog temperature, and stores it in its memory. In the meantime, the loop just keeps on running, the server refreshes etc. After 750ms, we contact the sensor again, and read the temperature from its memory.
To tell the library that we don't want to wait for the analog to digital conversion of the sensor, we use setWaitForConversion.

Loop

```
const unsigned long intervalNTP = ONE_HOUR; // Update the time every hour
unsigned long prevNTP = 0;
unsigned long lastNTPResponse = millis();

const unsigned long intervalTemp = 60000;   // Do a temperature measurement every minute
unsigned long prevTemp = 0;
bool tmpRequested = false;
const unsigned long DS_delay = 750;         // Reading the temperature from the DS18x20 can take up to
750ms

uint32_t timeUNIX = 0;                      // The most recent timestamp received from the time server

void loop() {
  unsigned long currentMillis = millis();

  if (currentMillis - prevNTP > intervalNTP) { // Request the time from the time server every hour
    prevNTP = currentMillis;
    sendNTPpacket(timeServerIP);
  }

  uint32_t time = getTime();                   // Check if the time server has responded, if so, get the
UNIX time
  if (time) {
    timeUNIX = time;
    Serial.print("NTP response:\t");
    Serial.println(timeUNIX);
    lastNTPResponse = millis();
  } else if ((millis() - lastNTPResponse) > 24UL * ONE_HOUR) {
    Serial.println("More than 24 hours since last NTP response. Rebooting.");
    Serial.flush();
    ESP.reset();
  }

  if (timeUNIX != 0) {
    if (currentMillis - prevTemp > intervalTemp) {  // Every minute, request the temperature
      tempSensors.requestTemperatures(); // Request the temperature from the sensor (it takes some time
to read it)
      tmpRequested = true;
```

```
      prevTemp = currentMillis;
      Serial.println("Temperature requested");
    }
    if (currentMillis - prevTemp > DS_delay && tmpRequested) { // 750 ms after requesting the
temperature
      uint32_t actualTime = timeUNIX + (currentMillis - lastNTPResponse) / 1000;
      // The actual time is the last NTP time plus the time that has elapsed since the last NTP response
      tmpRequested = false;
      float temp = tempSensors.getTempCByIndex(0); // Get the temperature from the sensor
      temp = round(temp * 100.0) / 100.0; // round temperature to 2 digits

      Serial.printf("Appending temperature to file: %lu,", actualTime);
      Serial.println(temp);
      File tempLog = SPIFFS.open("/temp.csv", "a"); // Write the time and the temperature to the csv
file
      tempLog.print(actualTime);
      tempLog.print(',');
      tempLog.println(temp);
      tempLog.close();
    }
  } else {                                  // If we didn't receive an NTP response yet, send another
request
    sendNTPpacket(timeServerIP);
    delay(500);
  }

  server.handleClient();                    // run the server
  ArduinoOTA.handle();                      // listen for OTA events
}
```

The loop looks a lot more complex, but it's actually pretty simple. It's all based on Blink Without Delay. There's two things going on:

1. Every hour, the ESP requests the time from an NTP server. Then it constantly checks for a response, and updates the time if it gets an NTP response. If it hasn't received any responses for over 24 hours, there's something wrong, and the ESP resets itself.
2. Every minute, the ESP requests the temperature from the DS18x20 sensor, and sets the 'tmpRequested' flag. The sensor will start the analog to digital conversion.
   750ms after the request, when the conversion should be finished, the ESP reads the temperature from the sensor, and resets the flag (otherwise, it would keep on reading the same temperature over and over again). Then it writes the time and the temperature to a file in SPIFFS.
   By saving it as a CSV file in the filesystem, we can easily download it to the client (using the web server that is running), and it's easy to parse with JavaScript.

If we miss the first NTP response, timeUNIX will be zero. If that's the case, we send another NTP request (otherwise, the next request would be an hour later, and the temperature logging only starts when the time is known).

We also need to run the server and OTA functions to handle HTTP and OTA requests.

Setup functions, server handlers and helper functions

These functions haven't change since the previous example, so there's no need to cover them here. You do need them to get the program running, though. Download the ZIP archive with examples for the full sketch.

HTML and JavaScript

There's some HTML and JavaScript files to plot the temperature using Google Graphs. I won't cover it here, but if you wan't to know how it works, you can find the files in the ZIP archive.

Using the example

Set the SPIFFS size to 64KB or larger if you plan to use it for prolonged periods of time. (You could also increase the logging interval on line 80 to save space.)
Enter your Wi-Fi credentials on lines 138-140, and hit upload. Then upload the webpages and scripts to SPIFFS using Tools > ESP8266 Sketch Data Upload.
Make sure you have the temperature sensor connected, as described at the top of this page. Open a terminal to see if it works. You should see something like this:

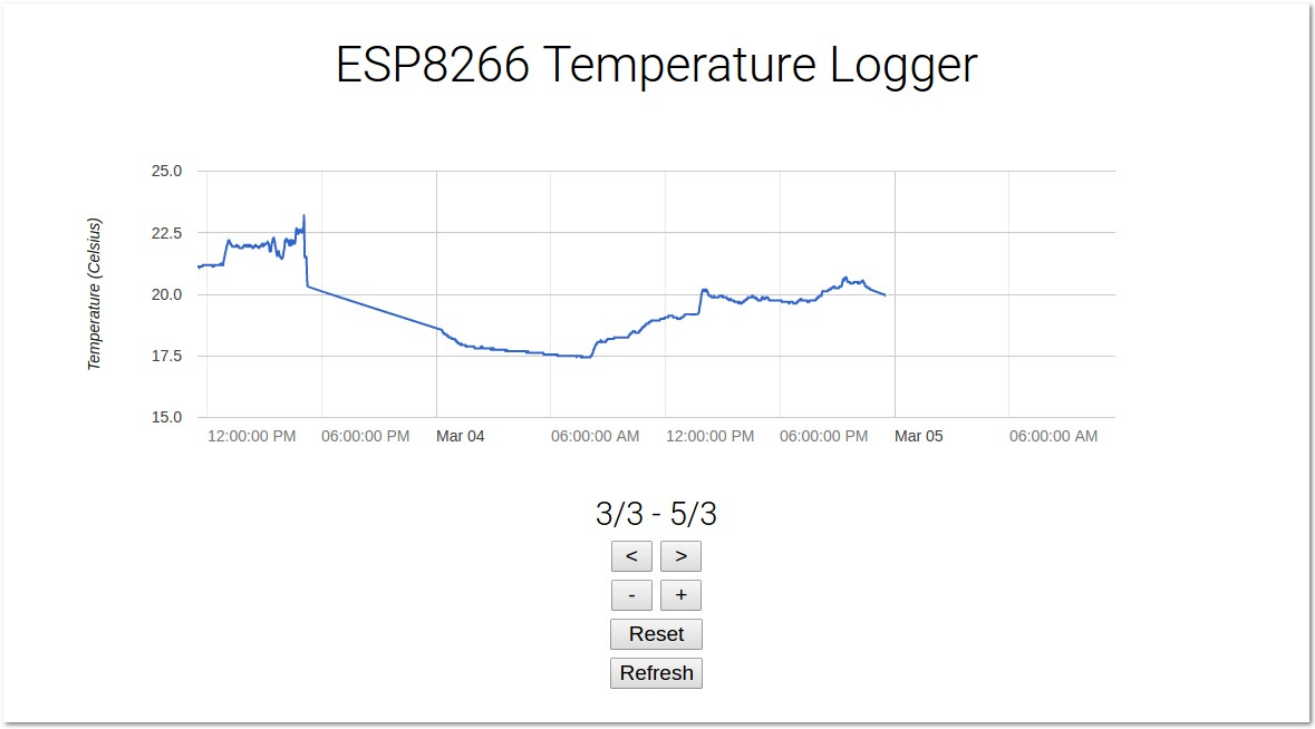
```
Connecting
..........

Connected to SSID
IP address:     192.168.1.2

OTA ready
```

```
SPIFFS started. Contents:
        FS File: /favicon-144x144.png, size: 2.81KB
        FS File: /temperatureGraph.js.gz, size: 1.17KB
        FS File: /temp.csv, size: 42.50KB
        FS File: /success.html.gz, size: 456B
        FS File: /edit.html.gz, size: 700B
        FS File: /main.css.gz, size: 349B
        FS File: /index.html.gz, size: 795B
        FS File: /manifest.json, size: 169B
        FS File: /favicon.ico.gz, size: 1.91KB

mDNS responder started: http://esp8266.local
HTTP server started.
Starting UDP
Local port:     123
Time server IP: 216.229.0.179
Sending NTP request
NTP response:   1488666586
Temperature requested
Appending temperature to file: 1488666627,20.00
Temperature requested
Appending temperature to file: 1488666687,19.94
Temperature requested
...
```

Let it run for a couple of minutes, to gather some temperature data. Then open a web browser, and go to http://esp8266.local/.
You should get a graph showing the temperature curve. You can use the arrow buttons to travel through time, and the + and - buttons to zoom in or out. The reset button resets the zoom, and jumps back to the present. Refresh requests the latest temperature data.

If you want, you can still go to http://esp8266.local/edit.html to upload new files.

The web interface should look like this:



It works on Windows, Linux and Android, but iOS seems to have some problems rendering the graph (in both Chrome and Safari).

Email notifier

Another great use for IoT devices is displaying things like traffic information, weather forecast, social media updates ... This requires us to send an HTTP GET request to the server of the service we'd like to access. Most popular services have API (Application Programming Interface) documents that explain that explain how you can retrieve certain information, and what format that information is in. In the following example, we'll look at Gmail specifically, but the code should be similar for other services.

Showing the number of unread emails

To communicate with Google's Gmail servers, we have to establish a secure connection to the server and send a secure HTTPS request with our email address and password. Gmail will then respond with an XML document containing all kinds of information, like (parts of) your most recent messages and the number of unread emails.

To make sure we don't send our Google password to a malicious server, we have to check the server's identity, using the SHA-1 fingerprint of the SSL certificate. This is a unique sequence of hexadecimal characters that identifies the server.

Allowing access to the email feed

The only way (I know of) to get email information from Google on the ESP currently is the Google Atom Feed. This is an older method, so you have to change your Gmail settings to allow access to the feed. Go to https://www.google.com/settings/security/lesssecureapps to enable access for "less secure apps":

Google

Review blocked sign-in attempt

Hi ESP8266,
Google just blocked someone from signing into your Google Account esp8266.test.mail@gmail.com from an app that may put your account at risk.

Less secure app
Thursday, February 9, 2017 7:06 PM (Central European Standard Time)
Belgium*

Don't recognize this activity?
If you didn't recently receive an error while trying to access a Google service, like Gmail, from a non-Google application, someone may have your password.

SECURE YOUR ACCOUNT

Are you the one who tried signing in?
Google will continue to block sign-in attempts from the app you're using because it has known security problems or is out of date. You can continue to use this app by allowing access to less secure apps, but this may leave your account vulnerable.

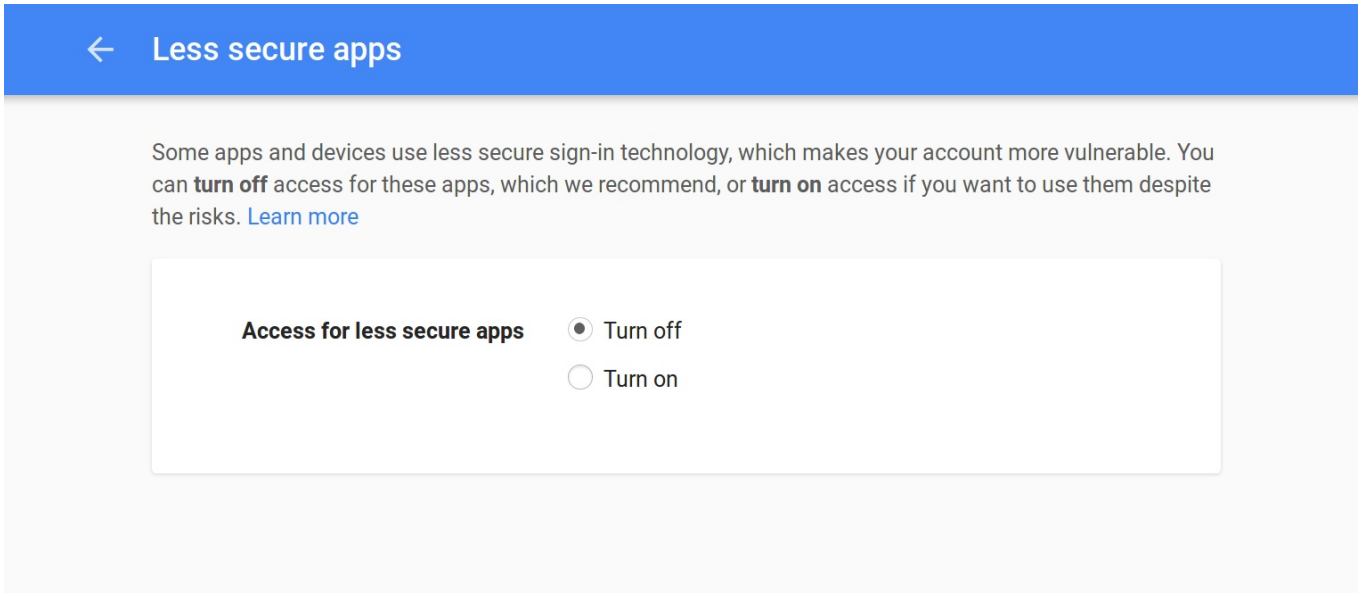
Best,
The Google Accounts team

*The location is approximate and determined by the IP address it was coming from.

This email can't receive replies. For more information, visit the Google Accounts Help Center.

You received this mandatory email service announcement to update you about important changes to your Google product or account.

© 2017 Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA



Until there's support for the new OAuth2 protocol on the ESP, we'll have to use the old, less secure method.

Hardware

Connect an LED (+ resistor) to pin 13, as an *unread email indicator*.

The Code

```
#include <WiFiClientSecure.h>   // Include the HTTPS library
#include <ESP8266WiFi.h>        // Include the Wi-Fi library
#include <ESP8266WiFiMulti.h>   // Include the Wi-Fi-Multi library

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'

const char* host = "mail.google.com"; // the Gmail server
const char* url = "/mail/feed/atom";  // the Gmail feed url
const int httpsPort = 443;            // the port to connect to the email server

                                      // The SHA-1 fingerprint of the SSL certificate for the Gmail
server (see below)
const char* fingerprint = "D3 90 FC 82 07 E6 0D C2 CE F9 9D 79 7F EC F6 E6 3E CB 8B B3";

                                      // The Base64 encoded version of your Gmail login credentials (see
below)
const char* credentials = "ZW1haWwuYWRkcmVzc0BnbWFpbC5jb206cGFzc3dvcmQ=";

const byte led = 13;

void setup() {
  Serial.begin(115200);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  pinMode(led, OUTPUT);

  wifiMulti.addAP("ssid_from_AP_1", "your_password_for_AP_1");   // add Wi-Fi networks you want to
connect to
  wifiMulti.addAP("ssid_from_AP_2", "your_password_for_AP_2");
  wifiMulti.addAP("ssid_from_AP_3", "your_password_for_AP_3");

  Serial.println("Connecting ...");
  int i = 0;
  while (wifiMulti.run() != WL_CONNECTED) { // Wait for the Wi-Fi to connect: scan for Wi-Fi networks,
and connect to the strongest of the networks above
    delay(250);
    Serial.print('.');
  }
  Serial.println('\n');
  Serial.print("Connected to ");
  Serial.println(WiFi.SSID());              // Tell us what network we're connected to
  Serial.print("IP address:\t");
  Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer
  Serial.println('\n');
}

void loop() {
  int unread = getUnread();
  if (unread == 0) {
    Serial.println("\r\nYou've got no unread emails");
    digitalWrite(led, LOW);
```

```
  } else if (unread > 0) {
    Serial.printf("\r\nYou've got %d new messages\r\n", unread);
    digitalWrite(led, HIGH);
  } else {
    Serial.println("Could not get unread mails");
  }
  Serial.println('\n');
  delay(5000);
}

int getUnread() {    // a function to get the number of unread emails in your Gmail inbox
  WiFiClientSecure client; // Use WiFiClientSecure class to create TLS (HTTPS) connection
  Serial.printf("Connecting to %s:%d ... \r\n", host, httpsPort);
  if (!client.connect(host, httpsPort)) {   // Connect to the Gmail server, on port 443
    Serial.println("Connection failed");    // If the connection fails, stop and return
    return -1;
  }

  if (client.verify(fingerprint, host)) {   // Check the SHA-1 fingerprint of the SSL certificate
    Serial.println("Certificate matches");
  } else {                                  // if it doesn't match, it's not safe to continue
    Serial.println("Certificate doesn't match");
    return -1;
  }

  Serial.print("Requesting URL: ");
  Serial.println(url);

  client.print(String("GET ") + url + " HTTP/1.1\r\n" +
               "Host: " + host + "\r\n" +
               "Authorization: Basic " + credentials + "\r\n" +
               "User-Agent: ESP8266\r\n" +
               "Connection: close\r\n\r\n"); // Send the HTTP request headers

  Serial.println("Request sent");

  int unread = -1;

  while (client.connected()) {                          // Wait for the response. The response is in XML
format
    client.readStringUntil('<');                        // read until the first XML tag
    String tagname = client.readStringUntil('>');       // read until the end of this tag to get the tag
name
    if (tagname == "fullcount") {                       // if the tag is <fullcount>, the next string
will be the number of unread emails
      String unreadStr = client.readStringUntil('<');   // read until the closing tag (</fullcount>)
      unread = unreadStr.toInt();                       // convert from String to int
      break;                                            // stop reading
    }                                                   // if the tag is not <fullcount>, repeat and
read the next tag
  }
  Serial.println("Connection closed");

  return unread;                                        // Return the number of unread emails
}
```

How it works

The setup should be pretty familiar by now.
The only new thing is the getUnread() function:
First, it starts an HTTPS connection to the Gmail server on port 443. Then it checks if the fingerprint of the certificate matches, so it knows that it's the real Google server, and not some hacker. If the certificate doesn't match, it's not safe to send the credentials to the server.

If it matches, we send a HTTP GET request to the server:

```
GET /mail/feed/atom HTTP/1.1\r\n
Host: mail.google.com\r\n
Authorization: Basic aVeryLongStringOfBase64EncodedCharacters=\r\n
User-Agent: ESP8266\r\n
Connection: close\r\n\r\n
```

The request contains the URI we want to access (in this case this is the Atom feed URL), the host (which is mail.google.com), and the base64-encoded version of your login credentials.
As you can see, the different lines of the header are separated by a CRLF (Carriage Return + Line Feed, \r\n). Two CRLF's mark the end of the header.
The Gmail server will process our request, and send the feed as a response over the same HTTPS connection. This response is an XML document, that consists of tags with angled brackets, just like HTML. If you need a lot of data, it's recommended to use a proper XML parser library, but we only need one tag, so we can just skim through the response text until we find the <fullcount>x</fullcount> tag. The number inside this tag is the number of unread emails in the inbox.
We can just convert it to an integer, and stop reading.

This is the format of the XML feed, you can see the fullcount tag on line 5:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://purl.org/atom/ns#" version="0.3">
    <title>Gmail - Inbox for esp8266.test.mail@gmail.com</title>
    <tagline>New messages in your Gmail Inbox</tagline>
    <fullcount>5</fullcount>
    <link rel="alternate" href="https://mail.google.com/mail" type="text/html" />
    <modified>2017-03-05T15:54:06Z</modified>
    <entry>
        <title>New sign-in from Firefox on Linux</title>
        <summary>New sign-in from Firefox on Linux Hi ESP8266, Your Google Account
esp8266.test.mail@gmail.com was just used to sign in from Firefox on Linux. ESP8266 Test
esp8266.test.mail@gmail.com Linux Sunday,</summary>
        <link rel="alternate" href="https://mail.google.com/mail?
account_id=esp8266.test.mail@gmail.com&amp;message_id=123456789&amp;view=conv&amp;extsrc=atom"
type="text/html" />
        <modified>2017-03-05T15:52:45Z</modified>
        <issued>2017-03-05T15:52:45Z</issued>
        <id>tag:gmail.google.com,2004:1234567891234567890</id>
        <author>
            <name>Google</name>
            <email>no-reply@accounts.google.com</email>
        </author>
    </entry>

    ...

</feed>
```

The loop just prints the number of unread emails, and turns on an LED if you have unread messages.

Getting the fingerprint of the Gmail server

Like I mentioned before, we need a fingerprint to check the identity of the server. To get this fingerprint, execute the following command in a terminal (Linux & Mac):

```
openssl s_client -connect mail.google.com:443 < /dev/null 2>/dev/null | openssl x509 -fingerprint -noout
-in /dev/stdin | sed 's/:/ /g'
```

Copy the hexadecimal fingerprint string and paste it into the sketch on line 12. For example:

```
const char* fingerprint = "D3 90 FC 82 07 E6 0D C2 CE F9 9D 79 7F EC F6 E6 3E CB 8B B3";
```

Encoding your login credentials

To get access to the feed, you have to enter your email address and password. You can't send them as plain text, you have to encode them to base64 first. Use the following command in a terminal (Linux & Mac):

```
echo -n "email.address@gmail.com:password" | base64
```

Then add it to line 15 of the sketch. For example:

```
const char* credentials = "ZW1haWwuYWRkcmVzc0BnbWFpbC5jb206cGFzc3dvcmQ=";
```

Other APIs

Many services send their data in JSON format. If you just need one piece of information, you may be able to use the same approach of scanning the entire JSON text for a certain word, but it's much easier to use a JSON parser, like the ArduinoJson library. It will deserialize the JSON text, and create a JSON object, you could compare it to an associative array. You can browse the entire tree structure, and easily find the data you're looking for.
The downside is that it uses more memory.

Advanced

DNS Captive Portal

When using the ESP8266 in access point mode, you probably want to redirect users to the right page. You can do this by creating a captive portal, using DNS. It's basically just a DNS server that will convert all host names to the ESP's own IP address.
This technique is also used by open Wi-Fi networks that redirect you to a login page before you can start browsing the internet.

Wi-Fi configuration

If you want to be able to change the Wi-Fi connection settings without re-uploading the code, you could take a look at the WiFiManager library by *tzapu*. This will try to connect to known networks, but if it fails, it will start a Wi-Fi access point. You can then connect to this access point, open the browser, and pick a network to connect to. The new configuration is saved.
The WiFiManager library uses a captive portal to present you with the right Wi-Fi settings page.
You could also implement a Wi-Fi manager yourself, or you can just check out the example that comes with the ESP8266 Arduino Core (Examples > DNSServer > CaptivePortalAdvanced).

I²S

The ESP8266 has an I²S bus on the RXD pin. It can run at 80MHz, and has DMA (direct memory access), so it's really fast. Its main purpose is to connect an I²S DAC (Digital to Analog Converter) to have an audio output, but you can use it for other things as well.
For example, CNLohr managed to transmit analog television, by connecting an antenna wire to the I²S pin. You can also use it to control WS2812Bs LEDs. You can even use it to communicate over Ethernet (not really useful, and definitely not recommended, but it works).
Another great use for the I²S bus is outputting data to shift registers. This gives you extra outputs that are reasonably fast, for things like LEDs or stepper motors.

Other examples

You can find lots of other examples in the Arduino IDE, I'd recommend to check those out as well.

YouTube

There's some great channels on YouTube that do amazing things with the ESP8266. Here's a short list of the ones I'm currently following. If you've got more recommendation, just leave a comment!

- Andreas Spiess
- CNLohr
- Acrobotic
- Miika Kurkela

In conclusion ...

Congratulations, you've reached the end of this rather long article on the basics of the ESP8266. I hope this was interesting to you, and that you'll use this knowledge for your own DIY projects.

If you have any remarks or if you want to help improve this guide, don't hesitate to leave a comment or to send me a message.

Thank you for reading!
Pieter, 8-3-2017