

# ESP8266WiFi library

ESP8266 is all about Wi-Fi. If you are eager to connect your new ESP8266 module to Wi-Fi network to start sending and receiving data, this is a good place to start. If you are looking for more in depth details of how to program specific Wi-Fi networking functionality, you are also in the right place.

## Introduction

The [Wi-Fi library for ESP8266](#) has been developed based on [ESP8266 SDK](#), using naming convention and overall functionality philosophy of [Arduino WiFi library](#). Over time the wealth Wi-Fi features ported from ESP9266 SDK to [esp8266 / Arduino](#) outgrew [Arduino WiFi library](#) and it became apparent that we need to provide separate documentation on what is new and extra.

This documentation will walk you through several classes, methods and properties of [ESP8266WiFi](#) library. If you are new to C++ and Arduino, don't worry. We will start from general concepts and then move to detailed description of members of each particular class including usage examples.

The scope of functionality offered by [ESP8266WiFi](#) library is quite extensive and therefore this description has been broken up into separate documents marked with :arrow\_right:.

## Quick Start

Hopefully you are already familiar how to load [Blink.ino](#) sketch to ESP8266 module and get the LED blinking. If not, please check [this tutorial](#) by Adafruit or [another great tutorial](#) developed by Sparkfun.

To hook up ESP module to Wi-Fi (like hooking up a mobile phone to a hot spot), you need just couple of lines of code:

```
#include <ESP8266WiFi.h>

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

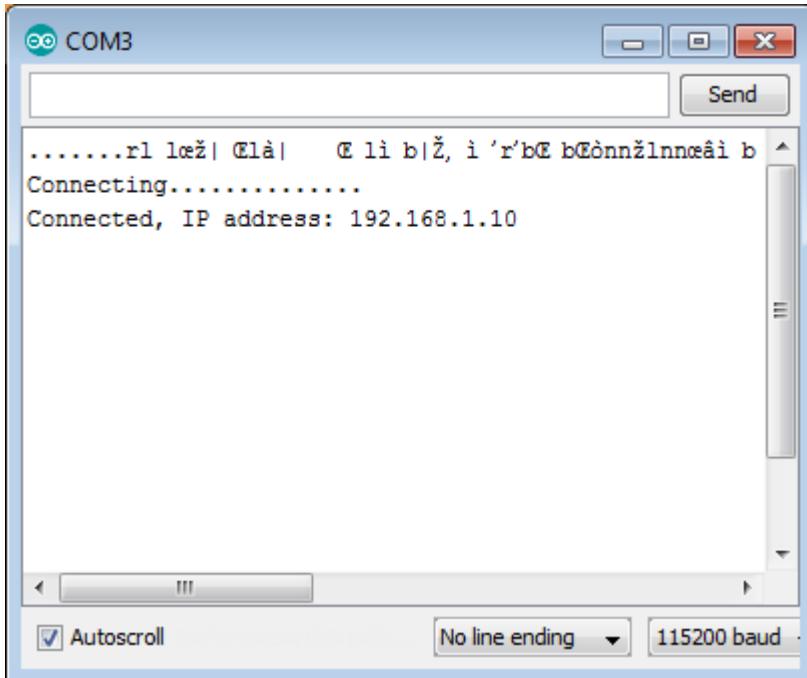
  WiFi.begin("network-name", "pass-to-network");

  Serial.print("Connecting");
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println();

  Serial.print("Connected, IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {}
```

In the line `WiFi.begin("network-name", "pass-to-network")` replace `network-name` and `pass-to-network` with name and password to the Wi-Fi network you like to connect. Then upload this sketch to ESP module and open serial monitor. You should see something like:



How does it work? In the first line of sketch `#include <ESP8266WiFi.h>` we are including [ESP8266WiFi](#) library. This library provides ESP8266 specific Wi-Fi routines we are calling to connect to network.

Actual connection to Wi-Fi is initialized by calling:

```
WiFi.begin("network-name", "pass-to-network");
```

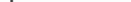
Connection process can take couple of seconds and we are checking for this to complete in the following loop:

```
while (WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
}
```

The `while()` loop will keep looping while `WiFi.status()` is other than `WL_CONNECTED`. The loop will exit only if the status changes to `WL_CONNECTED`.

The last line will then print out IP address assigned to ESP module by [DHCP](#):

```
Serial.println(WiFi.localIP());
```

If you don't see the last line but just more and more dots  , then likely name or password to the Wi-Fi network in sketch is entered incorrectly. Verify name and password by connecting from scratch to this Wi-Fi a PC or a mobile phone.

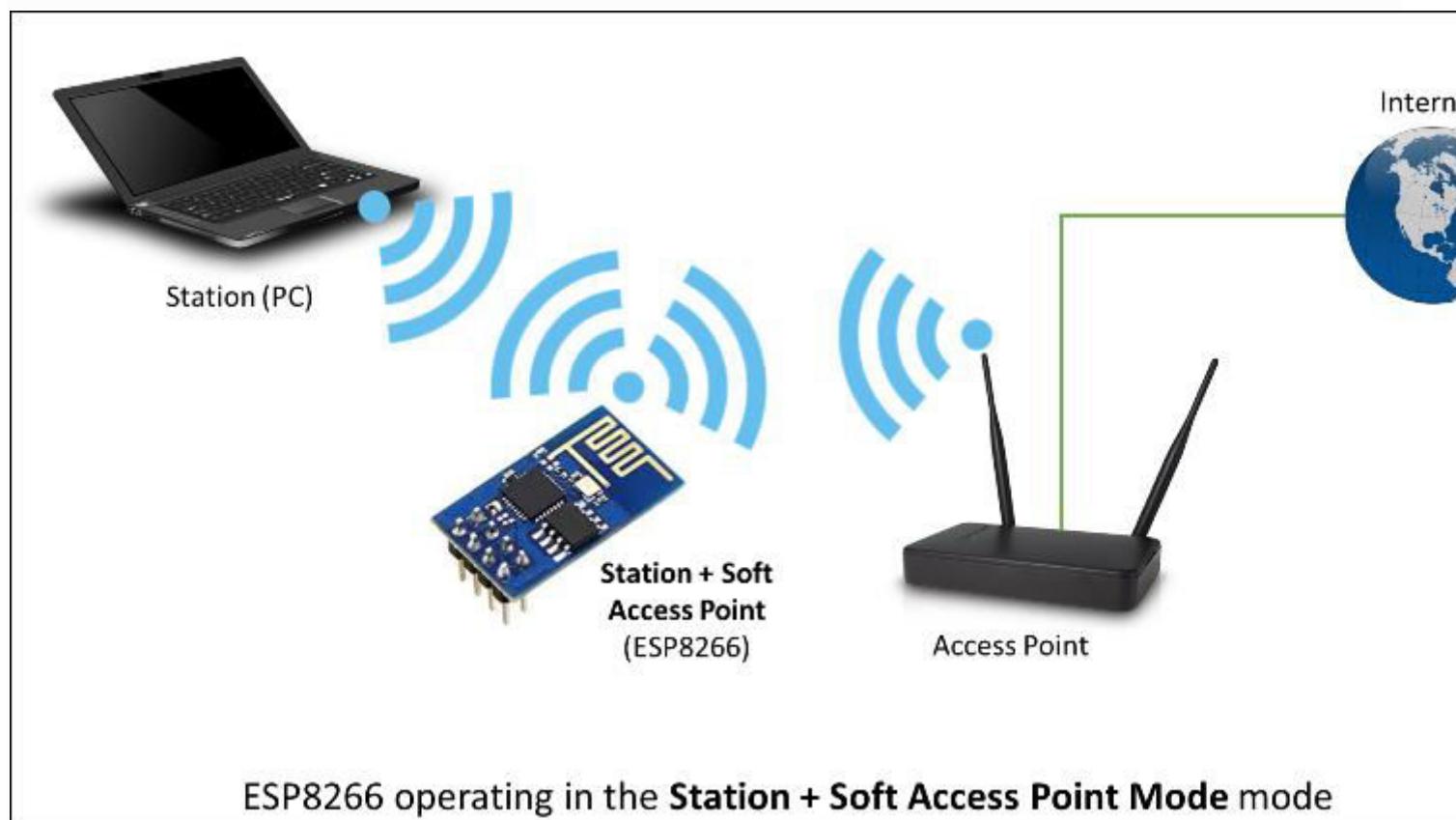
*Note:* if connection is established, and then lost for some reason, ESP will automatically reconnect to last used access point once it is again back on-line. This will be done automatically by Wi-Fi library, without any user intervention.

That's all you need to connect ESP8266 to Wi-Fi. In the following chapters we will explain what cool things can be done by ESP once connected.

## Who is Who

Devices that connect to Wi-Fi network are called stations (STA). Connection to Wi-Fi is provided by an access point (AP), that acts as a hub for one or more stations. The access point on the other end is connected to a wired network. An access point is usually integrated with a router to provide access from Wi-Fi network to the internet. Each access point is recognized by a SSID (**S**ervice **S**et **I**Dentifier), that essentially is the name of network you select when connecting a device (station) to the Wi-Fi.

ESP8266 module can operate as a station, so we can connect it to the Wi-Fi network. It can also operate as a soft access point (soft-AP), to establish its own Wi-Fi network. Therefore we can connect other stations to such ESP module. ESP8266 is also able to operate both in station and soft access point mode. This provides possibility of building e.g. [mesh networks](#).



The [ESP8266WiFi](#) library provides wide collection of C++ [methods](#) (functions) and [properties](#) to configure and operate an ESP8266 module in station and / or soft access point mode. They are described in the following chapters.

## Class Description

The [ESP8266WiFi](#) library is broken up into several classes. In most of cases, when writing the code, user is not concerned with this classification. We are using it to break up description of this library into more manageable pieces.

B	E	S	WiFiClient WiFiClientSecure WiFiEventHandlerOpaque WiFiEventModeChange WiFiEventSoftAPModeProbeRequestReceived WiFiEventSoftAPModeStationConnected WiFiEventSoftAPModeStationDisconnected WiFiEventStationModeAuthModeChanged WiFiEventStationModeConnected
BufferDataSource BufferedStreamDataSource	ESP8266WiFiAPClass ESP8266WiFiClass ESP8266WiFiGenericClass ESP8266WiFiMulti ESP8266WiFiScanClass ESP8266WiFiSTAClass	SList SSLContext	
C		U	
ClientContext		UdpContext	
D		W	
DataSource			
	P		WifiAPlist_t
	ProgmemStream		

Chapters below describe all function calls ([methods](#) and [properties](#) in C++ terms) listed in particular classes of [ESP8266WiFi](#). Description is illustrated with application examples and code snippets to show how to use functions in practice. Most of this information is broken up into separate documents. Please follow to access them.

## Station

Station (STA) mode is used to get ESP module connected to a Wi-Fi network established by an access point.



Station class has several features to facilitate management of Wi-Fi connection. In case the connection is lost, ESP8266 will automatically reconnect to the last used access point, once it is again available. The same happens on module reboot. This is possible since ESP is saving credentials to last used access point in flash (non-volatile) memory. Using the saved data ESP will also reconnect if sketch has been changed but code does not alter the Wi-Fi mode or credentials.

Check out separate section with [examples](#).

## Soft Access Point

An [access point \(AP\)](#) is a device that provides access to Wi-Fi network to other devices (stations) and connects them further to a wired network. ESP8266 can provide similar functionality except it does not have interface to a wired network. Such mode of operation is called soft access point (soft-AP). The maximum number of stations that can simultaneously be connected to the soft-AP can be set [from 0 to 8](#), but defaults to 4.



ESP8266 operating in the **Soft Access Point** mode

The soft-AP mode is often used and an intermediate step before connecting ESP to a Wi-Fi in a station mode. This is when SSID and password to such network is not known upfront. ESP first boots in soft-AP mode, so we can connect to it using a laptop or a mobile phone. Then we are able to provide credentials to the target network. Once done ESP is switched to the station mode and can connect to the target Wi-Fi.

Another handy application of soft-AP mode is to set up [mesh networks](#). ESP can operate in both soft-AP and Station mode so it can act as a node of a mesh network.

[Soft Access Point Class documentation](#)

Check out separate section with [examples](#).

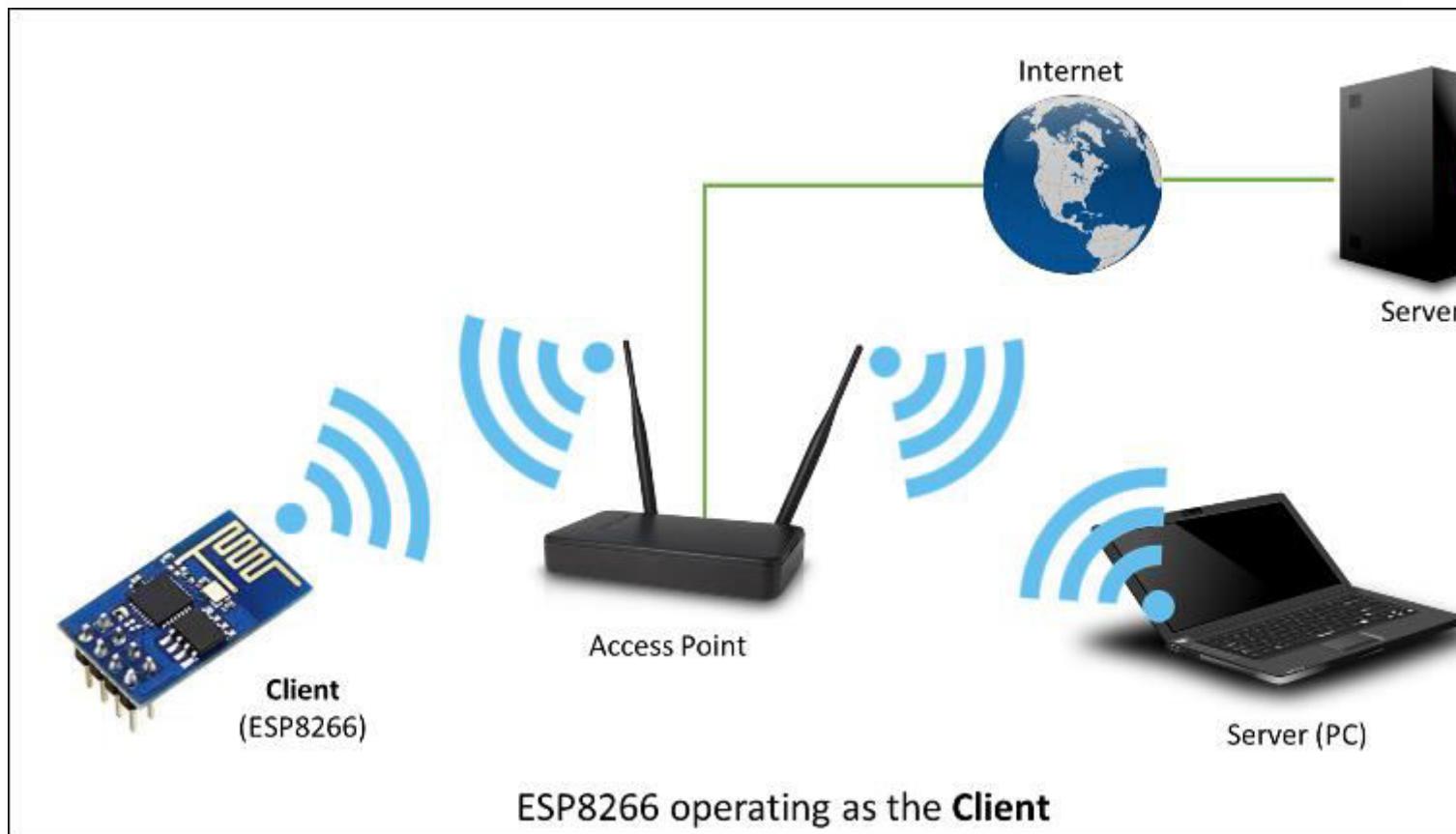
## Scan

To connect a mobile phone to a hot spot, you typically open Wi-Fi settings app, list available networks and pick the hot spot you need. Then enter a password (or not) and you are in. You can do the same with ESP. Functionality of scanning for, and listing of available networks in range is implemented by the Scan Class.

Check out separate section with [examples](#).

## Client

The Client class creates [clients](#) that can access services provided by [servers](#) in order to send, receive and process data.



Check out separate section with [examples](#) / [list of functions](#)

## axTLS Client Secure - DEPRECATED

The following section details axTLS, the older TLS library used by the project. It is still supported, but additional fixes and documentation will generally not be undertaken. See the following section for the updated TLS client object.

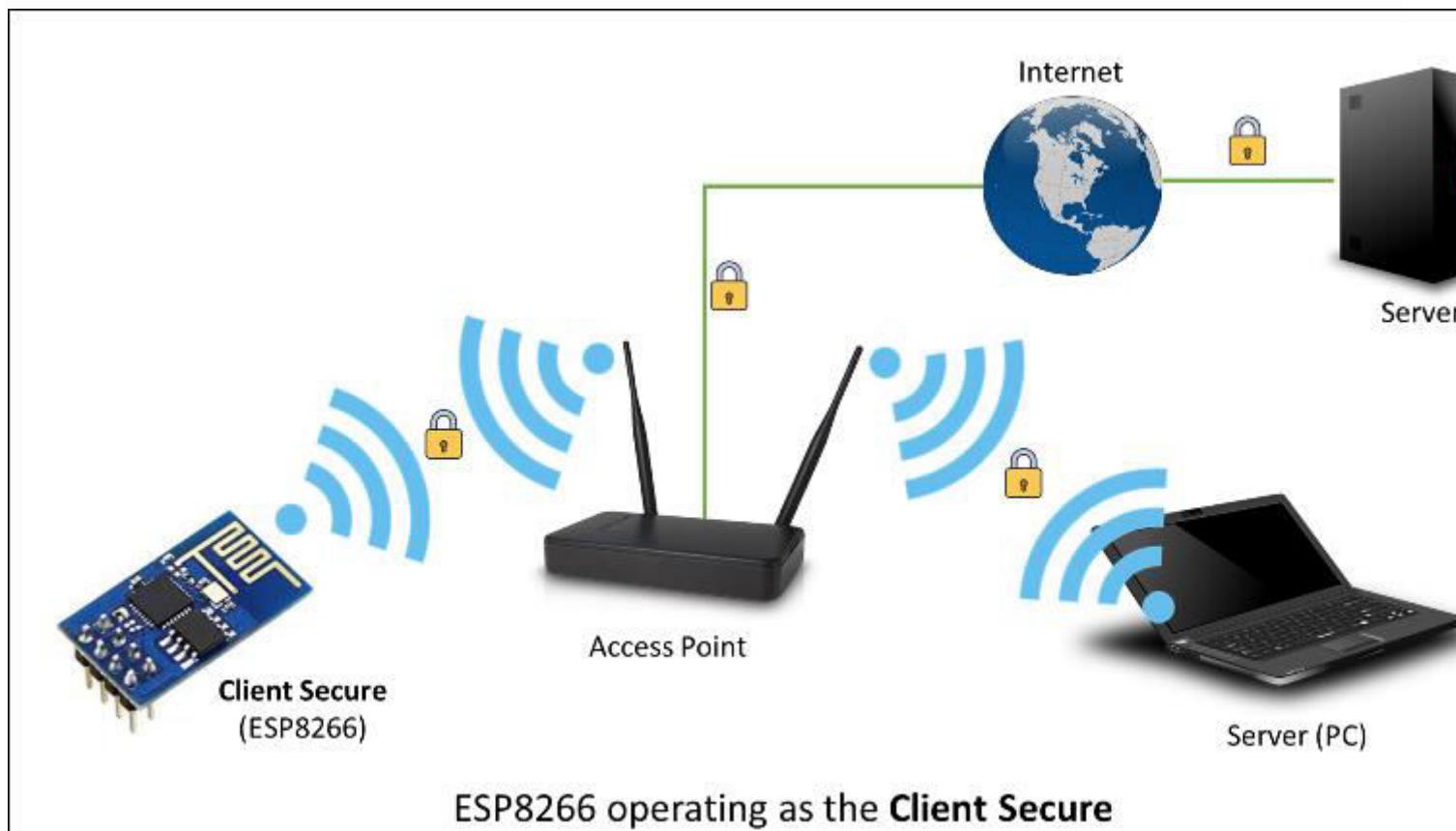
The axTLS Client Secure is an extension of [Client Class](#) where connection and data exchange with servers is done using a [secure protocol](#). It supports [TLS 1.1](#). The [TLS 1.2](#) is not supported.

Secure applications have additional memory (and processing) overhead due to the need to run cryptography algorithms. The stronger the certificate's key, the more overhead is needed. In practice it is not possible to run more than a single secure client at a time. The problem concerns RAM memory we can not add, the flash memory size is usually not the issue. If you like to learn how [client secure library](#) has been developed, access to what servers have been tested, and how memory limitations have been overcome, read fascinating issue report [#43](#).

Check out separate section with [examples / list of functions](#)

## BearSSL Client Secure and Server Secure

`BearSSL::WiFiClientSecure` and `BearSSL::WiFiServerSecure` are extensions of the standard [Client](#) and [Server](#) classes where connection and data exchange with servers and clients using [secure protocol](#). It supports [TLS 1.2](#) using a wide variety of modern ciphers, hashes, and key types.



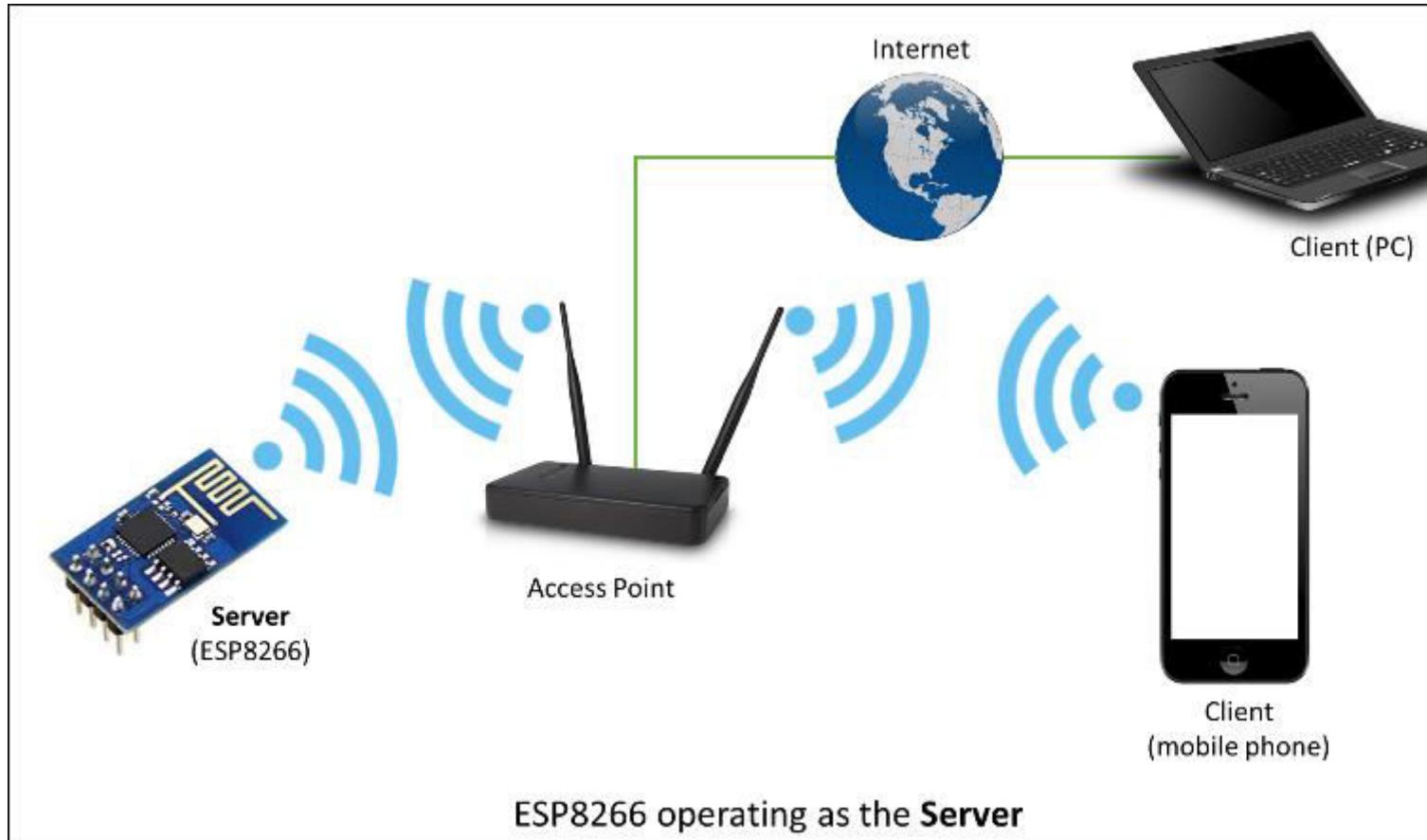
Secure clients and servers require significant amounts of additional memory and processing to enable their cryptographic algorithms. In general only a single secure client or server connection at a time can be processed given the little RAM present on the ESP8266, but there are methods of reducing this RAM requirement detailed in the relevant sections.

`BearSSL::WiFiClientSecure` contains more information on using and configuring TLS connections.

`BearSSL::WiFiServerSecure` discusses the TLS server mode available. Please read and understand the `BearSSL::WiFiClientSecure` first as the server uses most of the same concepts.

## Server

The Server Class creates [servers](#) that provide functionality to other programs or devices, called [clients](#).



Clients connect to sever to send and receive data and access provided functionality.

Check out separate section with [examples / list of functions](#).

## UDP

The UDP Class enables the [User Datagram Protocol \(UDP\)](#) messages to be sent and received. The UDP uses a simple “fire and forget” transmission model with no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

Check out separate section with [examples / list of functions](#).

## Generic

There are several functions offered by ESP8266’s [SDK](#) and not present in [Arduino WiFi library](#). If such function does not fit into one of classes discussed above, it will likely be in Generic Class. Among them is handler to manage Wi-Fi events like connection, disconnection or obtaining an IP, Wi-Fi mode changes, functions to manage module sleep mode, hostname to an IP address resolution, etc.

Check out separate section with [examples / list of functions](#).

## Diagnostics

There are several techniques available to diagnose and troubleshoot issues with getting connected to Wi-Fi and keeping connection alive.

## Check Return Codes

Almost each function described in chapters above returns some diagnostic information.

Such diagnostic may be provided as a simple `boolean` type `true` or `false` to indicate operation result. You may check this result as described in examples, for instance:

```
Serial.printf("Wi-Fi mode set to WIFI_STA %s\n", WiFi.mode(WIFI_STA) ? "" : "Failed!");
```

Some functions provide more than just a binary status information. A good example is `WiFi.status()`.

```
Serial.printf("Connection status: %d\n", WiFi.status());
```

This function returns following codes to describe what is going on with Wi-Fi connection:

- 0 : `WL_IDLE_STATUS` when Wi-Fi is in process of changing between statuses
- 1 : `WL_NO_SSID_AVAIL` in case configured SSID cannot be reached
- 3 : `WL_CONNECTED` after successful connection is established
- 4 : `WL_CONNECT_FAILED` if password is incorrect
- 6 : `WL_DISCONNECTED` if module is not configured in station mode

It is a good practice to display and check information returned by functions. Application development and troubleshooting will be easier with that.

## Use `printDiag`

There is a specific function available to print out key Wi-Fi diagnostic information:

```
WiFi.printDiag(Serial);
```

A sample output of this function looks as follows:

```
Mode: STA+AP
PHY mode: N
Channel: 11
AP id: 0
Status: 5
Auto connect: 1
SSID (10): sensor-net
Passphrase (12): 123!$#0&*esP
BSSID set: 0
```

Use this function to provide snapshot of Wi-Fi status in these parts of application code, that you suspect may be failing.

## Enable Wi-Fi Diagnostic

By default the diagnostic output from Wi-Fi libraries is disabled when you call `Serial.begin`. To enable debug output again, call `Serial.setDebugOutput(true)`. To redirect debug output to `Serial1` instead,

call `Serial1.setDebugOutput(true)`. For additional details regarding diagnostics using serial ports please refer to [the documentation](#).

Below is an example of output for sample sketch discussed in [Quick Start](#) above with `Serial.setDebugOutput(true)`:

```
Connectingscandone
state: 0 -> 2 (b0)
state: 2 -> 3 (0)
state: 3 -> 5 (10)
add 0
aid 1
cnt

connected with sensor-net, channel 6
dhcp client start...
chg_B1:-40
...ip:192.168.1.10,mask:255.255.255.0,gw:192.168.1.9
.
Connected, IP address: 192.168.1.10
```

The same sketch without `Serial.setDebugOutput(true)` will print out only the following:

```
Connecting....
Connected, IP address: 192.168.1.10
```

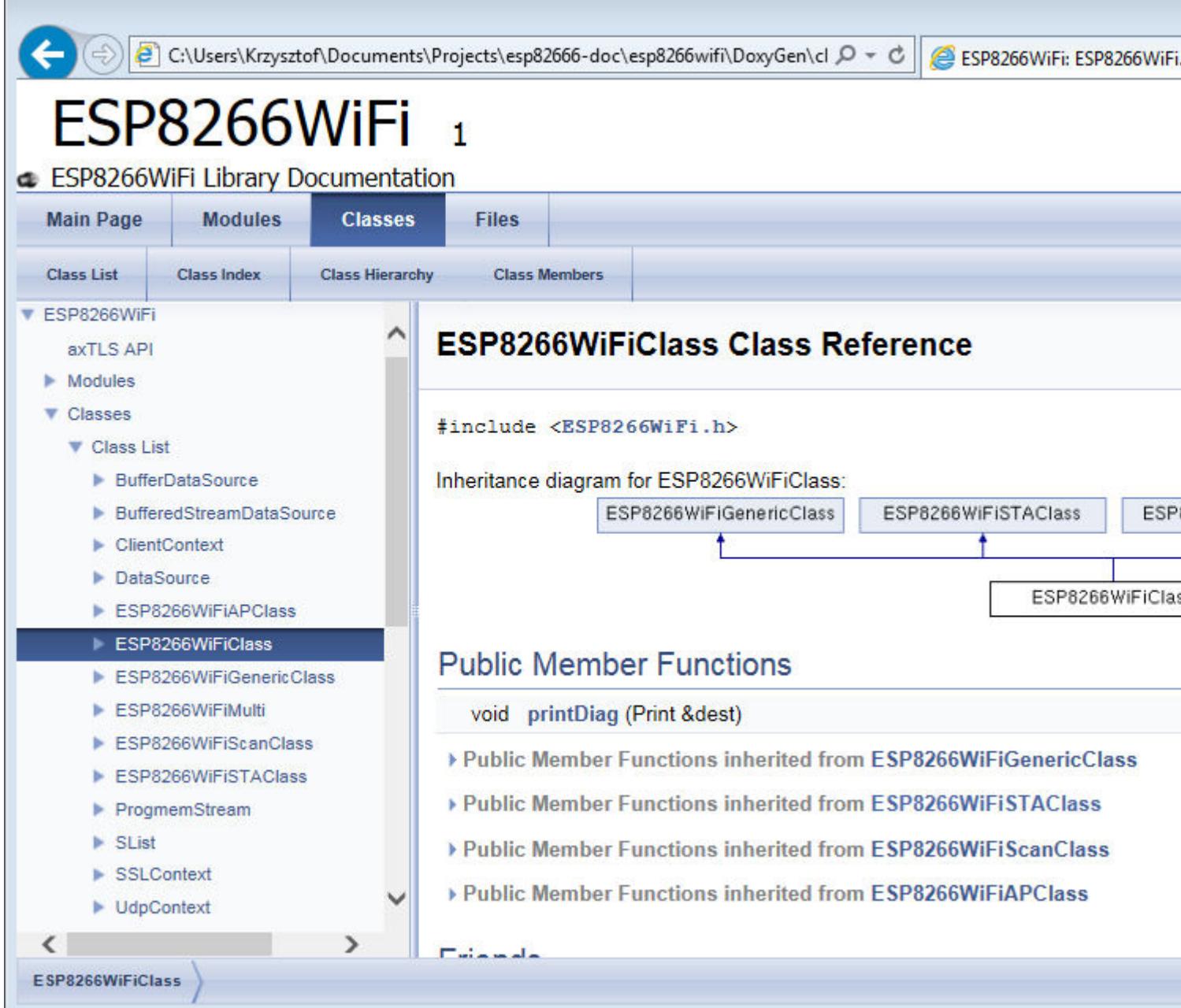
## Enable Debugging in IDE

Arduino IDE provides convenient method to [enable debugging](#) for specific libraries.

## What's Inside?

If you like to analyze in detail what is inside of the ESP8266WiFi library, go directly to the [ESP8266WiFi](#) folder of esp8266 / Arduino repository on the GitHub.

To make the analysis easier, rather than looking into individual header or source files, use one of free tools to automatically generate documentation. The class index in chapter [Class Description](#) above has been prepared in no time using great [Doxygen](#), that is the de facto standard tool for generating documentation from annotated C++ sources.



The tool crawls through all header and source files collecting information from formatted comment blocks. If developer of particular class annotated the code, you will see it like in examples below.

```
wl_status_t ESP8266WiFiSTAClass::begin ( const char * ssid,
                                         const char * passphrase = NULL,
                                         int32_t channel = 0,
                                         const uint8_t * bssid = NULL,
                                         bool connect = true
)
```

Start Wifi connection if passphrase is set the most secure supported mode will be automatically selected

#### Parameters

**ssid** const char\* Pointer to the SSID string.

**passphrase** const char \* Optional. Passphrase. Valid characters in a passphrase must be between ASCII 32-126 (de

**bssid** uint8\_t[6] Optional. BSSID / MAC of AP

**channel** Optional. Channel of AP

**connect** Optional. call connect

#### Returns

Definition at line 97 of file [ESP8266WiFiSTA.cpp](#).

```
bool ESP8266WiFiSTAClass::hostname ( char * aHostname )
```

Set ESP8266 station DHCP hostname

#### Parameters

**aHostname** max length:32

#### Returns

ok

Definition at line 422 of file [ESP8266WiFiSTA.cpp](#).

If code is not annotated, you will still see the function prototype including types of arguments, and can use provided links to jump straight to the source code to check it out on your own. Doxygen provides really excellent navigation between members of library.

```
uint8_t WiFiUDP::begin ( uint16_t port )
```

Definition at line 77 of file [WiFiUdp.cpp](#).

Several classes of [ESP8266WiFi](#) are not annotated. When preparing this document, [Doxygen](#) has been tremendous help to quickly navigate through almost 30 files that make this library.