

Sheffield Hallam University
Department of Engineering
 BEng (Hons) Computer Systems Engineering



Activity ID		Activity Title			Laboratory Room No.	Level
Lab 102		RESTful Arduino web services			4302	6
Semester	Duration [hrs]	Group Size	Max Total Students	Date of approval/ review	Lead Academic	
1	6	1	25	09-21	Alex Shenfield	

Equipment (per student/group)

Number	Item
1	Arduino kit
1	SparkFun ESP8266 shield

Learning Outcomes

	Learning Outcome
1	Understand the benefits, challenges and pitfalls of developing internet enabled embedded systems.
3	Implement multi-component distributed embedded systems using a flexible network architecture.

Developing RESTful Arduino web services

Introduction

Computing is about more than the PC on your desktop! Embedded devices are everywhere – from wireless telecommunications infrastructure points to electronic point of sale terminals. One definition of an embedded system is:

“An embedded system is a computer system designed to perform one or a few dedicated functions often with real-time computing constraints.”

(http://en.wikipedia.org/wiki/Embedded_system)

In this series of labs you are going to be introduced to the open source Arduino platform – a cheap, simple to program, well documented prototyping platform for designing electronic systems. The heart of this prototyping system is the Arduino microcontroller board itself which is based on the commonly used ATmega328 chip.

The purpose of this lab session is to demonstrate some of the network functionality we can add to the Arduino platform using the SparkFun ESP8266 WiFi shield, before showing how we can run RESTful web services on the Arduino.

Bibliography

There are no essential bibliographic resources for this laboratory session aside from this tutorial sheet. However the following websites and tutorials may be of help (especially if you haven't done much electronics previously or your digital logic and/or programming is a bit rusty):

- <http://www.arduino.cc/>
- <http://www.ladyada.net/learn/arduino/index.html>
- <http://tronixstuff.wordpress.com/tutorials/>

Equipment

Check that you have all the necessary equipment (see Figure 1)!

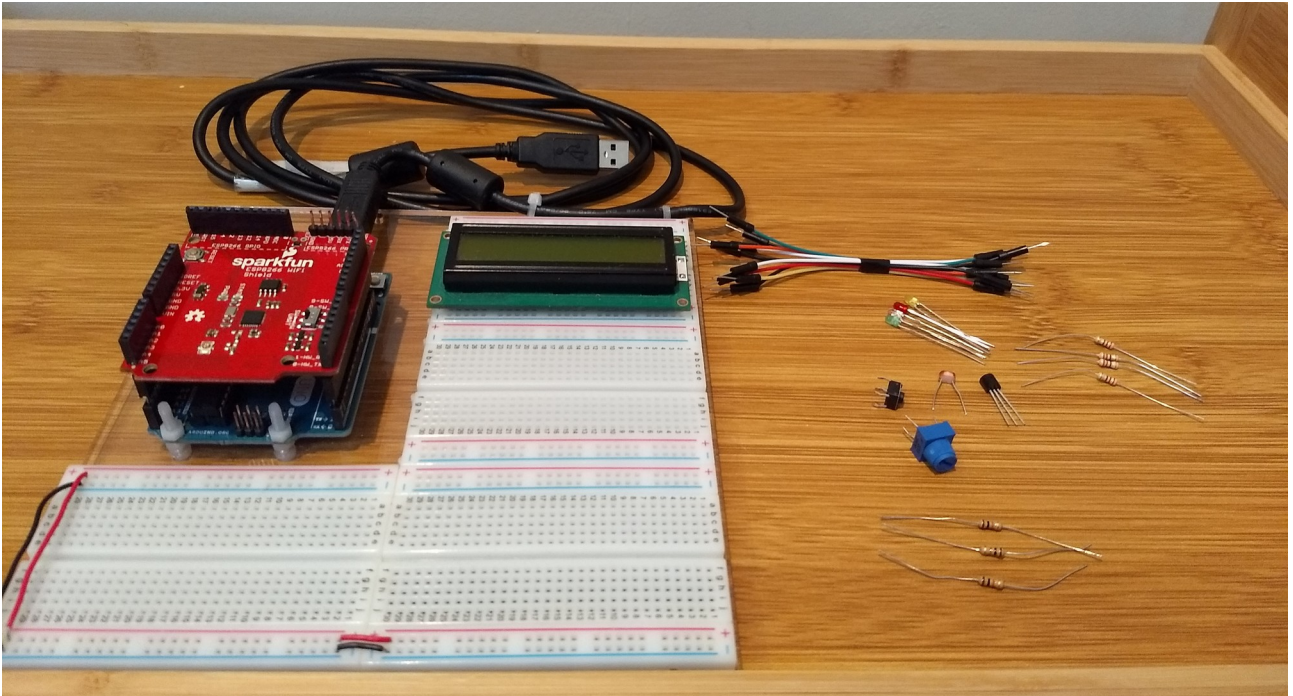


Figure 1 – The necessary equipment for this lab

Task 1

The Arduino SparkFun ESP8266 WiFi shield is capable of turning the Arduino into both a web client (like a browser) or a web server using AT commands sent to the ESP8266 chip embedded on the shield.

Unfortunately, the SparkFun ESP8266 shield ships with a (very) old version of the Espressif AT firmware that is lacking in many features and the default SparkFun library is also very limited!

To overcome these limitations I have reflashed the default firmware on the ESP8266 chips with AT firmware version 1.7¹ and we will use Juraj Andrassy's WiFiEspAT library² which (unlike the SparkFun library) uses the standard Arduino WiFi networking API. Although this is not installed by default, you can install it from the Arduino Library Manager – press CTRL + Shift + I, and then search for “WiFiEspAT” (see Figure 2).

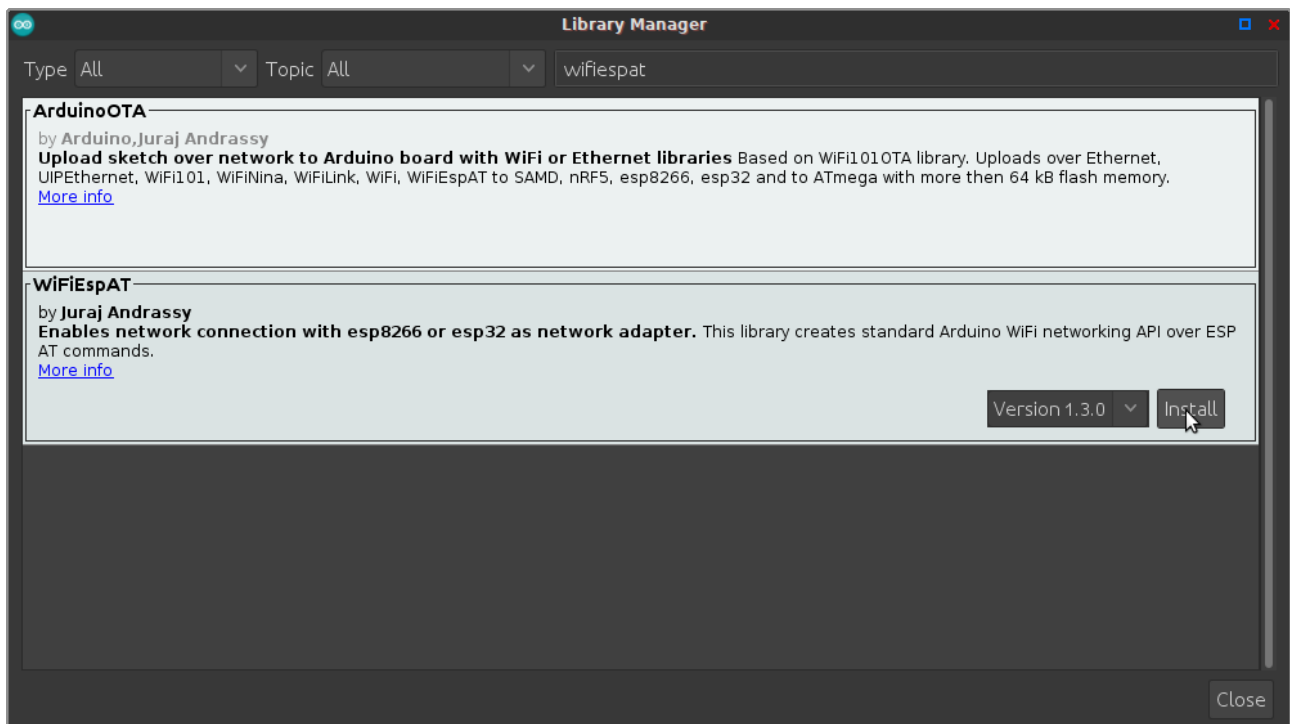


Figure 2 – Installing the WiFiEspAT library

- 1 I am actually using the firmware built by Boris Lovosevic (see https://github.com/loboris/ESP8266_AT_LoBo) because the official firmware can only be flashed to ESP8266 devices with 1MB flash – and unfortunately the model on the SparkFun shield has only 512KB.
- 2 See <https://github.com/jandrassy/WiFiEspAT> for more details.

The first task in this lab is to set the Arduino up as a simple web server. Connect the Arduino and SparkFun ESP8266 shield up as in Figure 3 below.

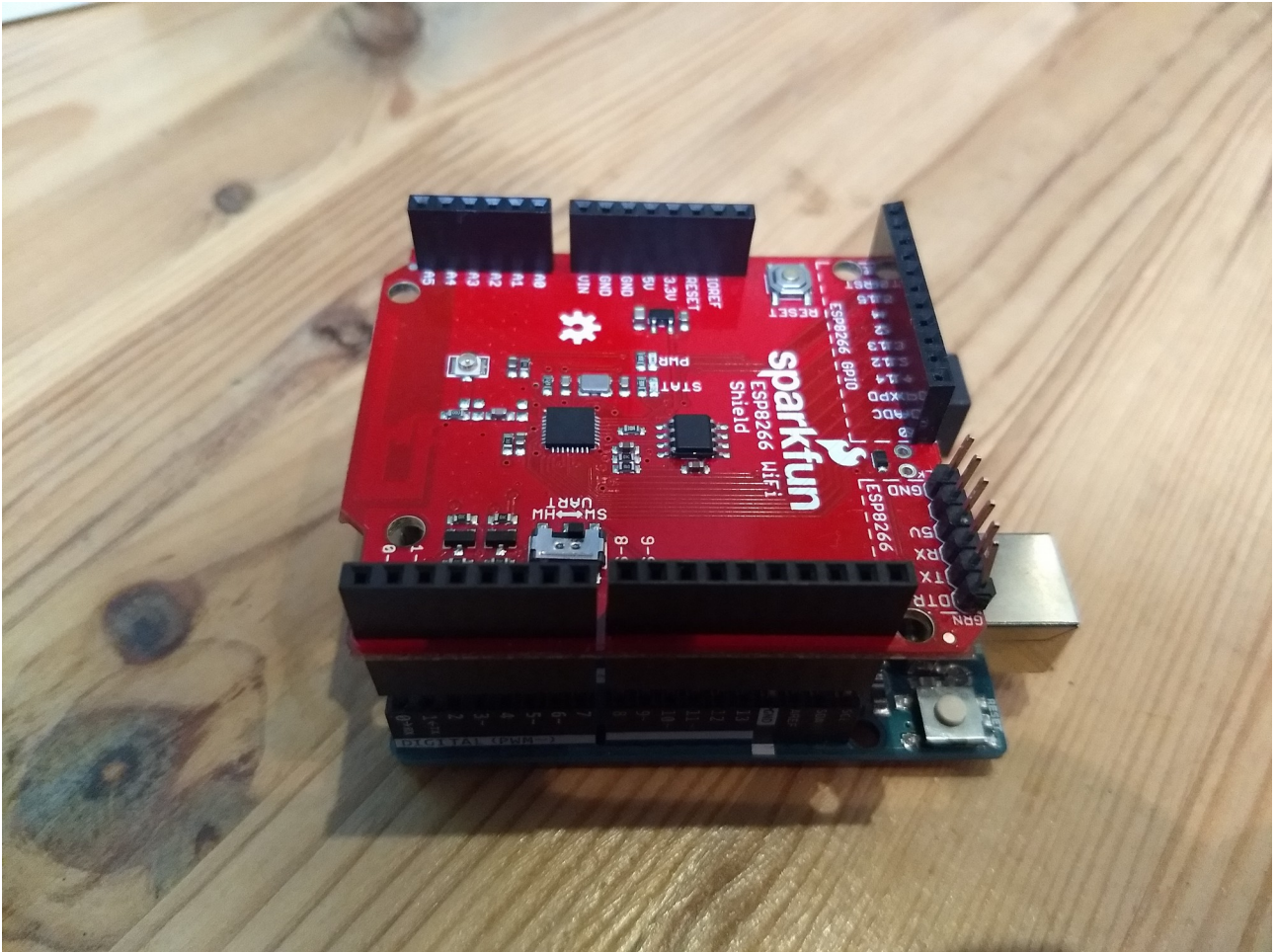


Figure 3 – Arduino and ESP8266 shield

The ESP8266 shield communicates with the Arduino via a serial interface (UART) to send commands and receive data. The Arduino Uno development platform that we are using in these labs only has one hardware serial port (on pins 0 and 1) and that is used for programming the board and communicating with the serial monitor via USB; however, we can simulate a serial port in software using the SoftwareSerial library. This does have a few limitations – of these, the most important limitation is that the SoftwareSerial library is not reliable at high speeds.

When flashing the latest firmware on to the shields, I have set the default baud rate of the ESP8266 chips on the SparkFun shield to 9600bps (which is a speed that the SoftwareSerial library comfortably supports). This is important to be aware of because the original default baud rate is 115200bps which will cause communication problems with the Arduino!

The UART switch on the SparkFun shields should also be set to “SW” (see Figure 4). This redirects the serial communication between the Arduino and the ESP8266 chip to use pins 8 and 9 (rather than the hardware serial port on pins 0 and 1).

IF YOU TRY AND PROGRAM THE ARDUINO WHEN THIS SWITCH IS SET TO “HW”, YOU MIGHT CORRUPT THE FIRMWARE ON THE ESP8266 CHIP – SO TRY NOT TO DO THIS!

FOR THESE LABS YOU CAN LEAVE THE SWITCH SET TO “SW” PERMANENTLY!

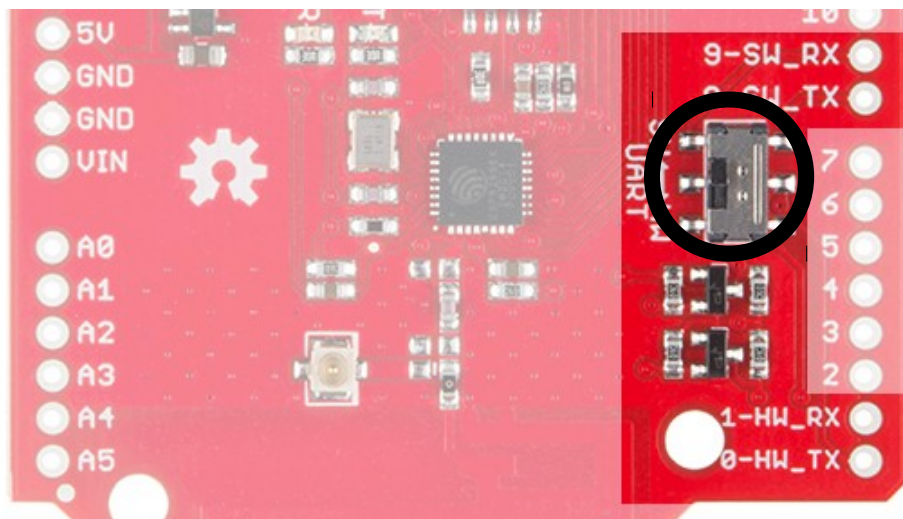


Figure 4 – The mode selector switch on the ESP8266 shield

We are now going to write a simple program to make the Arduino act like a web server. The code is provided in code listings 1 & 2 (below) and goes through the following steps:

1. Sets up a WiFi network connection to the Arduino
2. Listens for data being sent from a client
3. Displays this data on the serial monitor
4. Sends the standard HTTP response header and closes the connection when there is no more data being sent

Firstly we need to provide our wireless access point credentials. I like to do this in a separate header file to keep these apart from the main code. To add a separate header file you can click the down arrow on the right hand side of the Arduino IDE as shown in Figure 5a (or alternatively you can press CTRL + SHIFT + N). This will allow you to choose a name for the new file (as shown in Figure 5b) – I have chosen “MyCredentials.h”.

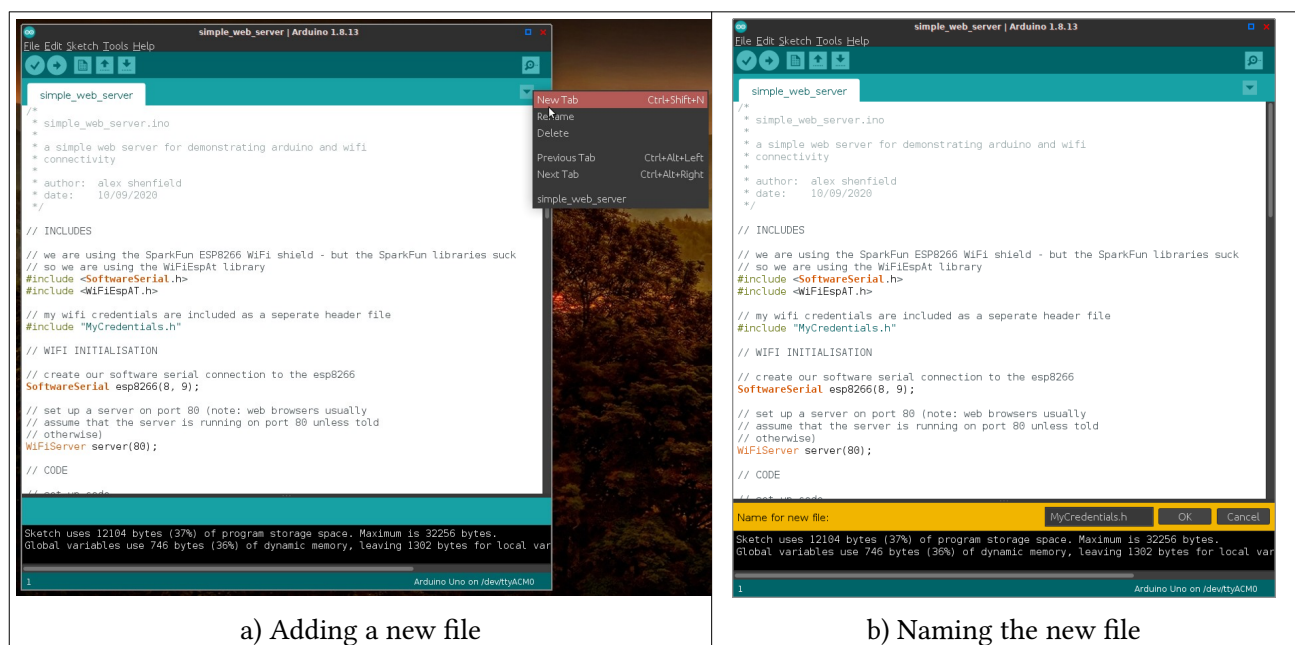


Figure 5 – Adding a new header file for WiFi credentials

Code listing 1: MyCredentials.h

```
// my wifi credentials
const char mySSID[] = "<your SSID>";
const char myPSK[] = "<your password>";
```

Code listing 2: A simple web server

```

/*
 * simple_web_server.ino
 *
 * a simple web server for demonstrating arduino and wifi
 * connectivity
 *
 * author:  alex shenfield
 * date:    10/09/2020
 */

// INCLUDES

// we are using the SparkFun ESP8266 WiFi shield - but the SparkFun libraries suck
// so we are using the WiFiEspAt library
#include <SoftwareSerial.h>
#include <WiFiEspAT.h>

// my wifi credentials are included as a separate header file
#include "MyCredentials.h"

// WIFI INITIALISATION

// create our software serial connection to the esp8266
SoftwareSerial esp8266(8, 9);

// set up a server on port 80 (note: web browsers usually
// assume that the server is running on port 80 unless told
// otherwise)
WiFiServer server(80);

```



```
// CODE

// set up code
void setup()
{
    // set up serial comms for debugging and display of DHCP allocated IP address
    Serial.begin(9600);

    // set up the esp8266 module
    esp8266.begin(9600);
    if (!WiFi.init(esp8266))
    {
        Serial.println("error talking to ESP8266 module");
        while(true)
        {
            {
            }
        }
    }
    Serial.println("ESP8266 connected");

    // connect to wifi
    WiFi.begin(mySSID, myPSK);

    // waiting for connection to Wifi network
    Serial.println("waiting for connection to WiFi");
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.print('.');
    }
    Serial.println();
    Serial.println("connected to WiFi network");

    // print the IP address to the serial monitor
    IPAddress myIP = WiFi.localIP();
    Serial.print("My IP: ");
    Serial.println(myIP);

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

```
// main code
void loop()
{
    // check for a client connection
    WiFiClient client = server.available();

    // if a remote client is connected
    if (client)
    {
        // get the ip address of the remote client
        IPAddress ip = client.remoteIP();
        Serial.print("new client at ");
        Serial.println(ip);

        // while the client is still connected
        while (client)
        {
            // and has more data to send
            if (client.available() > 0)
            {
                // read bytes from the incoming client and write them to
                // the serial monitor
                Serial.print((char)client.read());
            }
            // when the client is done sending data
            else
            {
                // send standard http response header (to acknowledge the
                // data)
                client.println("HTTP/1.1 200 OK");
                client.println("Content-Type: text/html");
                client.println();

                // disconnect from client
                client.stop();
            }
        }
    }
}
```

We can then upload our code and open the Arduino IDE serial monitor window. This should show us the IP address of our networked Arduino as in Figure 6, below.

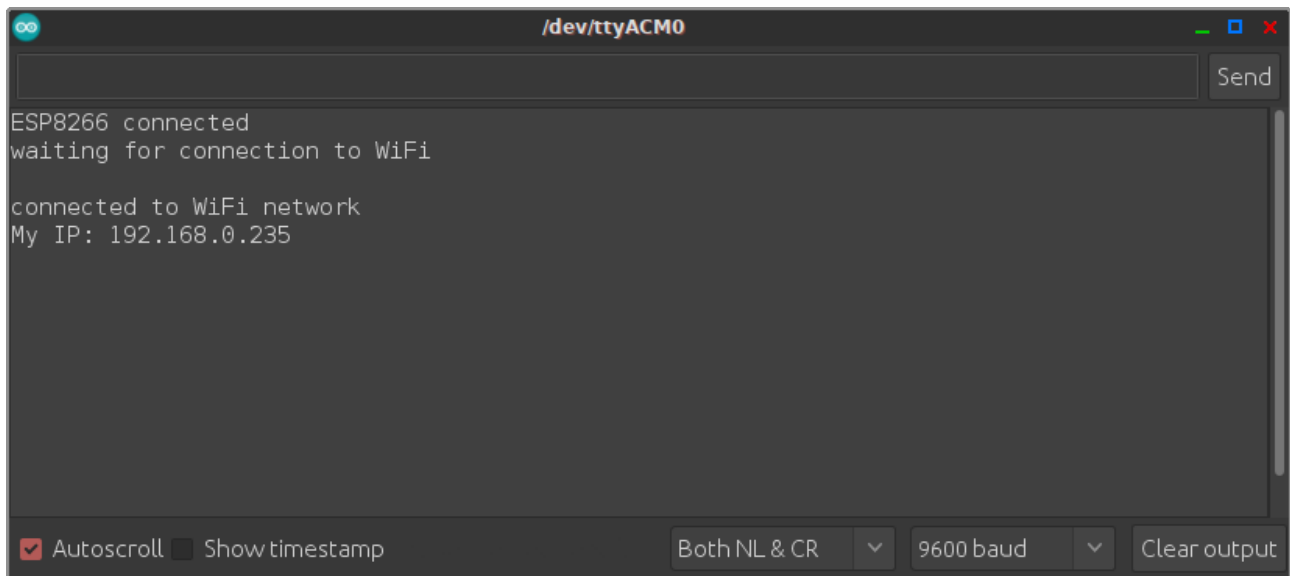


Figure 6 – The serial monitor display the Arduino web server's IP address

Now, using your favourite web browser, you can navigate to this address (see Figure 7). You should see the serial monitor display all the connection information (see Figure 8).



Figure 7 – Navigating to the Arduino web server

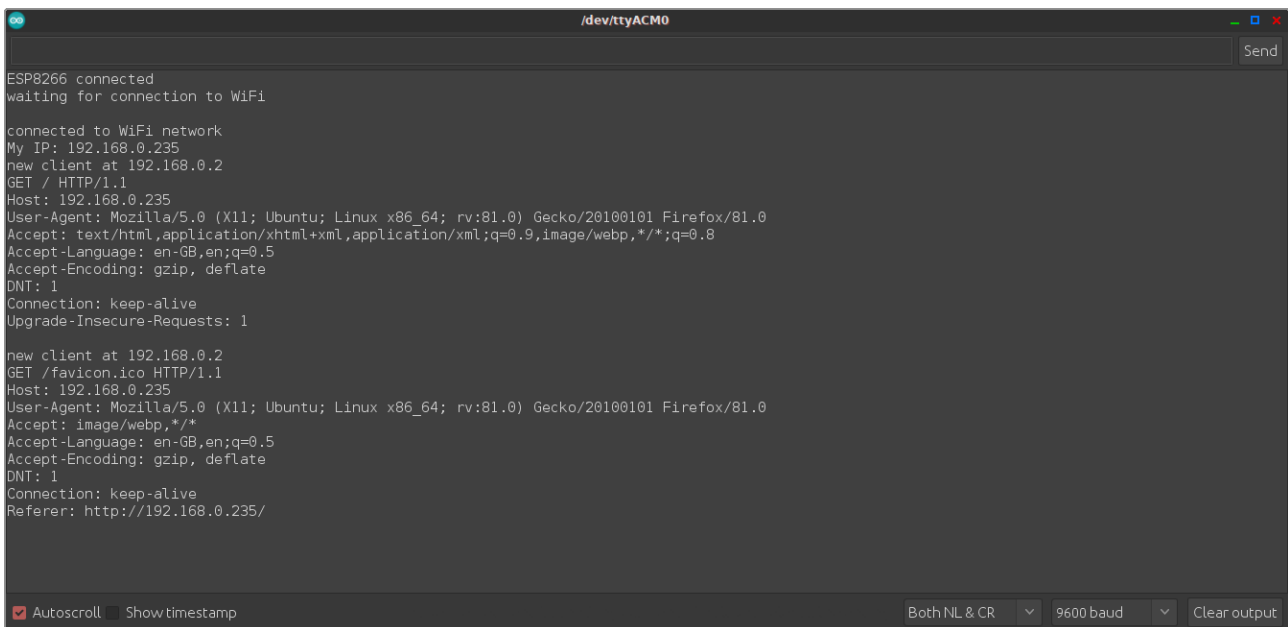


Figure 8 – Serial monitor output

Try entering a different URL and seeing what output you get on the serial monitor – e.g.

192.168.0.235/testing

Q1 What is the key difference?

Exercises

Now alter your program to actually display some properly formatted HTML in the web page rather than just sending an empty HTTP response header. A good resource for some basic HTML commands is:

<http://www.w3schools.com/html/default.asp>

Some suggestions are:

1. displaying a simple text string
2. displaying a link to another web page
3. experimenting with the various HTML formatting options

Figure 9 shows a (admittedly hideous) example web page served up by the Arduino. Hopefully you can do better!

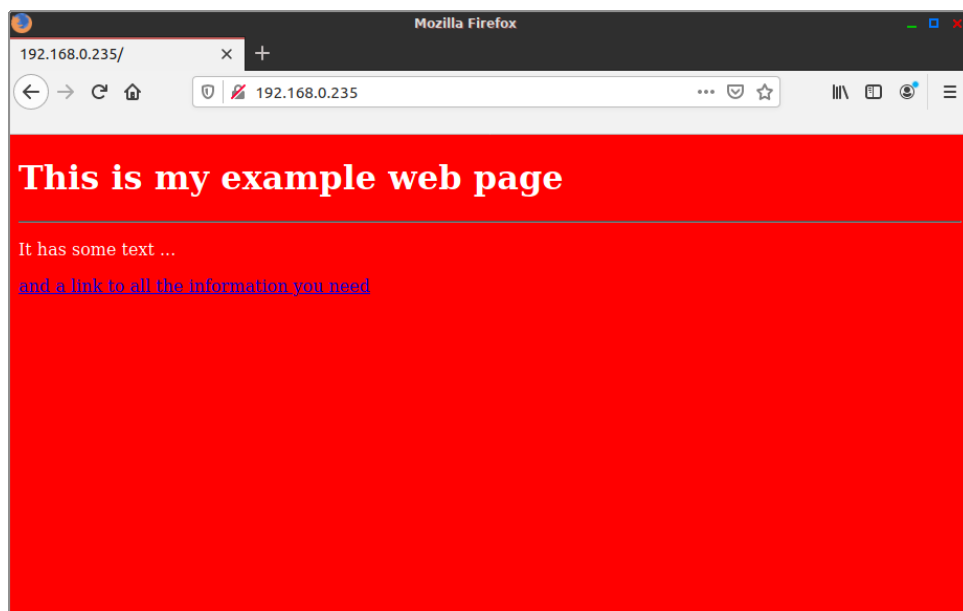


Figure 9 – “My eyes are bleeding!”

Task 2

Now we have mastered the basics and learnt a bit about the structure of HTTP requests and responses, we can use our Arduino as more than just a web server. In fact, by network enabling our Arduino, we are well on our way to creating elements of the “Internet of Things”.

In this task we are going to use the Arduino and WiFi shield as a web service to control 4 LEDs attached to pins 2, 3, 4, and 5.

First build the circuit shown in Figure 10³.

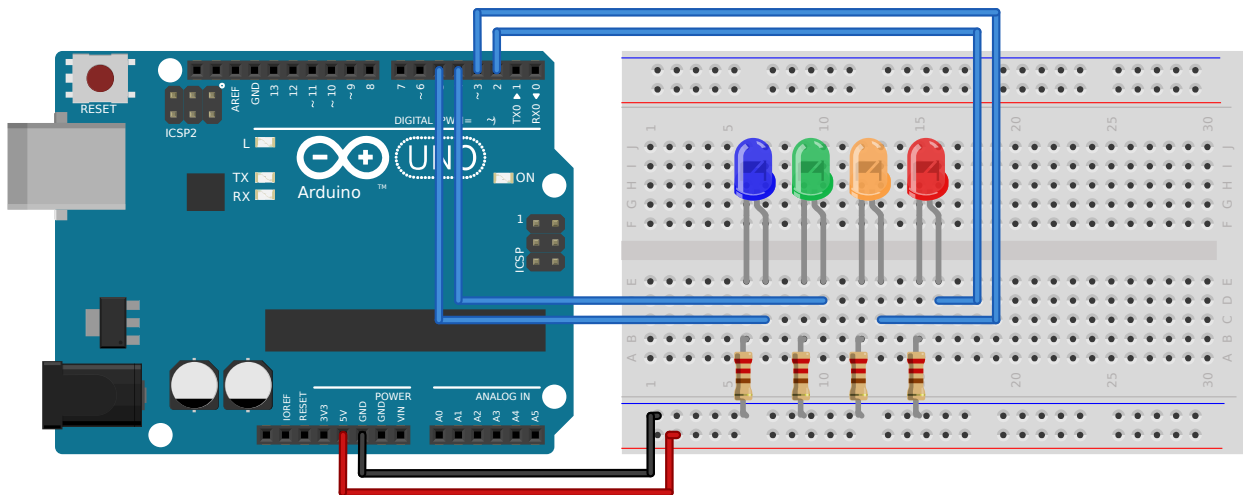


Figure 10 – WiFi controlled LEDs

We now need to write the code to control these LEDs over the network. We can break this down into 3 parts:

1. the initialisation part (where we set up the Arduino and the WiFi network connection) shown in code listing 3. Don't forget to add a header file with your WiFi credentials!
2. the main loop of the program (that simply checks for a connection to the server) and the functions that handle the web client's HTTP requests to the Arduino (shown in code listing 4)
3. the functions that are called to actually control the LEDs (shown in code listing 5)

You should recognise most of the code in code listing 3 from the simple web server we created in Task 1. The HTTP handler code (in code listing 4) basically just parses the information returned from the HTTP client (i.e. the web browser) looking for key markers (in this case a '?' character). Once it sees this marker, it knows the the following data is either the Arduino pins to turn on or a command to turn the pins off.

3 Note, Fritzing doesn't have a part drawing for the SparkFun ESP8266 shield so I am using the standard Arduino part to indicate connections (obviously you will have the SparkFun ESP8266 shield on top of the Arduino!).

Code listing 3: Setting up the web server

```

/*
 * web_addressable_leds.ino
 *
 * simple web service to control LEDs
 *
 * author:  alex shenfield
 * date:    10/09/2020
 */

// we are using the SparkFun ESP8266 WiFi shield - but the SparkFun libraries suck
// so we are using the WiFiEspAt library
#include <SoftwareSerial.h>
#include <WiFiEspAT.h>

// my wifi credentials are included as a separate header file
#include "MyCredentials.h"

// create our software serial connection to the esp8266
SoftwareSerial esp8266(8, 9);

// set up a server on port 80 (note: web browsers usually
// assume that the server is running on port 80 unless told
// otherwise)
WiFiServer server(80);

// variable indicating when to start paying attention to data
boolean pay_attention = true;

// CODE

// set up code
void setup()
{
    // set up serial comms for debugging and display of DHCP allocated IP address
    Serial.begin(9600);

    // set up the esp8266 module
    esp8266.begin(9600);
    if (!WiFi.init(esp8266))
    {
        Serial.println("error talking to ESP8266 module");
        while(true)
        {
        }
    }
    Serial.println("ESP8266 connected");

    // connect to wifi
    WiFi.begin(mySSID, myPSK);

    // waiting for connection to Wifi network
    Serial.println("waiting for connection to WiFi");
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.print('.');
    }
    Serial.println();
    Serial.println("connected to WiFi network");
}

```

```
// print the IP address to the serial monitor
IPAddress myIP = WiFi.localIP();
Serial.print("My IP: ");
Serial.println(myIP);

// start the server
server.begin();

// set up the pins as outputs
pinMode(2, OUTPUT);
pinMode(3, OUTPUT);
pinMode(4, OUTPUT);
pinMode(5, OUTPUT);
}
```

Code listing 4: HTTP request handler functions

```
// main code
void loop()
{
    // constantly check for connections
    check_for_connections();
}

// method to check for incoming connections and process them
void check_for_connections()
{
    // get any client that is connected to the server and
    // trying to send data
    WiFiClient client = server.available();

    // record whether we have already sent the standard http
    // response header
    boolean header_sent = false;

    // if a remote client is connected
    if (client)
    {
        // get the ip address of the remote client
        IPAddress ip = client.remoteIP();
        Serial.print("new client at ");
        Serial.println(ip);

        // while the client is connected ...
        while (client)
        {
            // if we haven't already sent the http header
            if (!header_sent)
            {
                // send standard http response header (to acknowledge the
                // data)
                client.println("HTTP/1.1 200 OK");
                client.println("Content-Type: text/html");
                client.println();
                header_sent = true;
            }
        }
    }
}
```

```
// ... and has more data to send
if (client.available() > 0)
{
    // read the next byte
    char c = client.read();

    // debugging
    Serial.print(c);

    // pay attention to all the data between the '?' character
    // and a space
    if (c == '?')
    {
        pay_attention = true;
    }
    else if (c == ' ')
    {
        pay_attention = false;
    }

    // if we are paying attention ...
    if (pay_attention)
    {
        // use a switch statement to decide what to do
        switch (c)
        {
            case '2':
                trigger_pin(2, client);
                break;

            case '3':
                trigger_pin(3, client);
                break;

            case '4':
                trigger_pin(4, client);
                break;

            case '5':
                trigger_pin(5, client);
                break;

            case 'x':
                clear_pins(client);
                break;

        }
    }
}
// when the client is done sending data
else
{
    // disconnect from client
    client.stop();
}
}
```

Code listing 5: The LED control functions

```
// separate function for triggering pins - note we only need
// the wifi client so we can send data to it
void trigger_pin(int pin_number, WiFiClient client)
{
    // print a status message
    client.print("Turning on pin <b>");
    client.println(pin_number);
    client.println("</b><br>");

    // turn on appropriate pin
    digitalWrite(pin_number, HIGH);
}

// another function to clear all pins - again the wifi client
// is only needed to return data to the web page
void clear_pins(WiFiClient client)
{
    // print a status message
    client.println("Clearing all pins!<br>");

    // turn off pins
    digitalWrite(2, LOW);
    digitalWrite(3, LOW);
    digitalWrite(4, LOW);
    digitalWrite(5, LOW);
}
```

Enter this code, make sure you understand it, and then upload it to the Arduino. You should then be able to control the LEDs by navigating to the IP address shown in the serial monitor.

You can turn on the LEDs on pins 3 and 5 by going to the URL:

192.168.0.235/?35

and turn off all the LEDs by going to the URL:

192.168.0.235/?x

This is the first step in exposing some simple functionality of our Arduino as a RESTful web service! We can now turn on some of our digital pins (i.e. the LEDs) by sending simple HTTP GET commands to our Arduino web service.

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Connect up a potentiometer to one of the analog pins and adapt the code above to read (and return) the potentiometer value if we call:
192.168.0.235/?p
2. Add a temperature sensor and light sensor to the system and hook into the web service to return those values when a given URL is called.
3. Add a push button to the system and write a method that returns the last latched button state (i.e. every time the button is pressed we toggle the state and when the method is called we return the last state). Hook that into the web service.

Task 3

So in the last task we exposed some basic functionality of our system as a RESTful web service, providing the ability to control some of the digital outputs of our Arduino via HTTP calls. However, in this case we have had to implement all the HTTP connection parsing code ourselves. For the simple example in the previous task, this is not too difficult – however, if we want to implement a more complex system (for example, monitoring various analog inputs and providing the ability to call more complex functions via HTTP), the code gets substantially more complex!

Fortunately there is no need to reinvent the wheel! One of the key advantages of the Arduino platform is that there is a vast ecosystem of existing libraries and projects to help us. In this case, Marco Schwartz has written the excellent aREST library for Arduino that enables us to write programs that register variables and methods with a REST API – without having to write all the low level HTTP handling code! See:

- <https://github.com/marcoschwartz/aREST>
- <https://arest.io/>

This allows us to focus on the application design rather than the REST API and web service details. As with the installation of the WiFiEspAT library in task 1, we can install this via the library manager in the Arduino IDE.

In this task we are going to create a system that uses the aREST framework to allow us to control the brightness of an LED and monitor the ambient light levels within a room – all via HTTP calls to a RESTful web service running on our Arduino.

Code listing 6 shows a pseudo-code outline of this system.

Code listing 6: RESTful light controller pseudo code

```
Procedure LIGHTS:

  initialise_system()

  // register variables and functions with
  // the REST api
  register_variable_with_api(light_level)
  register_function_with_api(led_control)

LOOP:
  light_level = read_sensor()

  // if we have a client connection,
  // parse the HTTP commands, extract
  // the parameters, and service the
  // callbacks
  if client.connected()
    handle_connection()
  end if

end LOOP

end Procedure LIGHTS
```

Now build the circuit shown in Figure 11⁴.

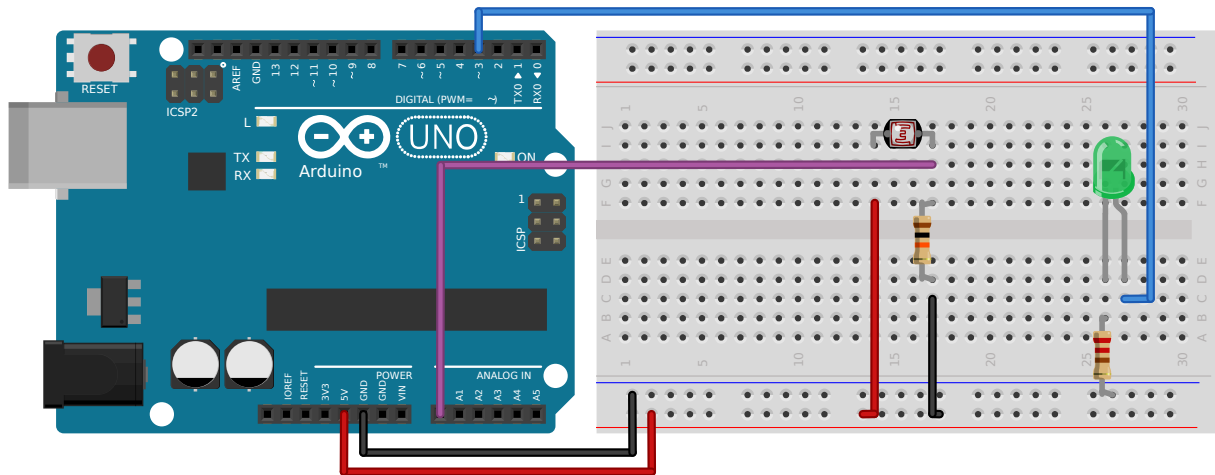


Figure 11 – The light controller circuit

Code listing 7 provides the full code for this RESTful light controller application.

Code listing 7: A RESTful lighting controller

```
/*
 * restful_arduino.ino
 *
 * arduino restful api example using the aREST library - in this example we are
 * setting the brightness of an led on pin 3 and reading light values
 *
 * author:  alex shenfield
 * date:    10/09/2020
 */

// INCLUDES

// we are using the SparkFun ESP8266 WiFi shield - but the SparkFun libraries suck
// so we are using the WiFiEspAt library
#include <SoftwareSerial.h>
#include <WiFiEspAT.h>

// my wifi credentials are included as a seperate header file
#include "MyCredentials.h"

// include the aREST library (making sure that it knows we are using a WiFi client
#define WiFi_h 1
#include <aREST.h>

// include the avr watchdog timer library
#include <avr/wdt.h>
```

4 Again, you will have the SparkFun shield on top of the Arduino here.

```
// WIFI INITIALISATION

// create our software serial connection to the esp8266
SoftwareSerial esp8266(8, 9);

// set up a server on port 80 (note: web browsers usually
// assume that the server is running on port 80 unless told
// otherwise)
WiFiServer server(80);

// AREST DECLARATIONS

// create an aREST instance
aREST rest = aREST();

// variables to monitor with our webservice
int light_level = 0;
```

```
// CODE

// set up code
void setup()
{
    // set up serial comms for debugging and display of DHCP allocated ip
    // address
    Serial.begin(9600);
    Serial.println("starting rest web service on arduino ...");

    // set up the esp8266 module
    esp8266.begin(9600);
    if (!WiFi.init(esp8266))
    {
        Serial.println("error talking to ESP8266 module");
        while(true)
        {
            {
            }
        }
    }
    Serial.println("ESP8266 connected");

    // connect to wifi
    WiFi.begin(mySSID, myPSK);

    // waiting for connection to Wifi network
    Serial.println("waiting for connection to WiFi");
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.print('.');
    }
    Serial.println();
    Serial.println("connected to WiFi network");

    // print the IP address to the serial monitor
    IPAddress myIP = WiFi.localIP();
    Serial.print("My IP: ");
    Serial.println(myIP);

    // start the server
    server.begin();

    // set pin modes
    pinMode(3, OUTPUT);
    pinMode(A0, INPUT);

    // expose the light level variable to the REST api
    rest.variable("light_level", &light_level);

    // expose the led trigger function to the REST api
    rest.function("led", led_control);

    // set the name and id for this webservice node
    rest.set_id("001");
    rest.set_name("alexs_arduino");

    // start watchdog timer
    wdt_enable(WDTO_4S);
}
```

```
// main loop
void loop()
{
  // update our light level each time round the loop
  light_level = analogRead(A0);

  // listen for incoming clients
  WiFiClient client = server.available();
  rest.handle(client);
  wdt_reset();
}

// FUNCTIONS EXPOSED TO THE REST API

// led control function accessible by the API
int led_control(String command)
{
  // debugging information about the actual command string
  Serial.print("command is ");
  Serial.println(command);

  // get pwm signal from command
  int pwm = command.toInt();
  pwm = constrain(pwm, 0, 255);

  // send pwm signal to the led
  analogWrite(3, pwm);

  // return 1 (indicating success)
  return 1;
}
```

Enter this code, make sure you understand it, and then upload it to the Arduino. Now open the serial monitor - you should see something like that shown in Figure 12.

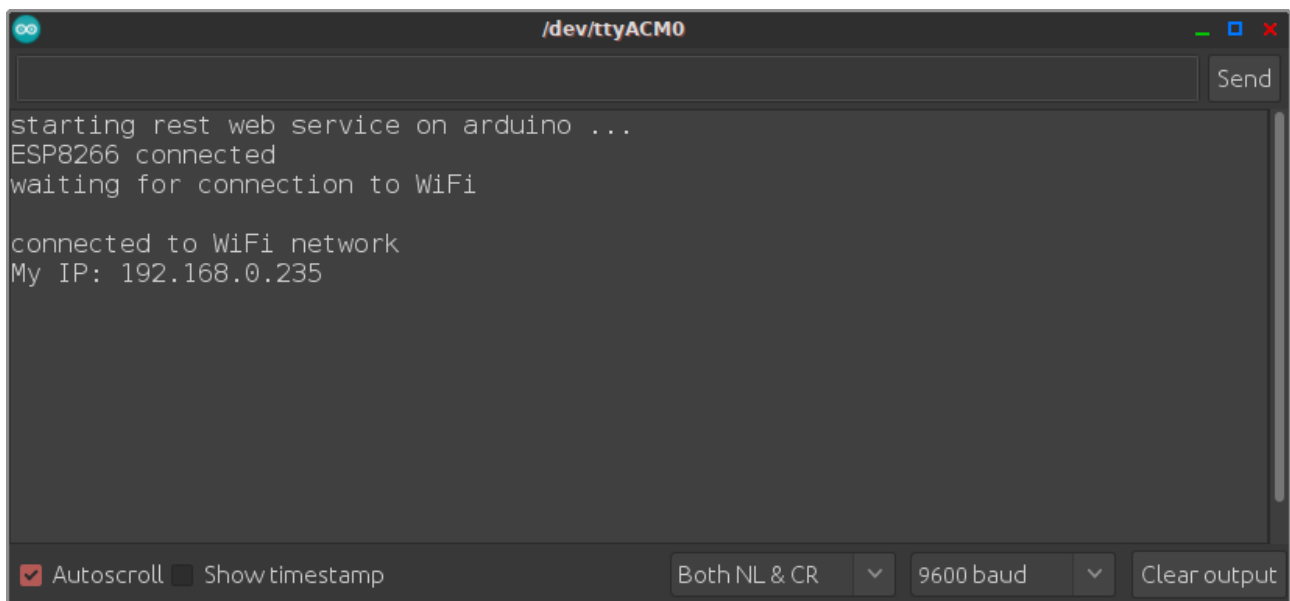


Figure 12 – Serial monitor window

To test the aREST api is working you can open your favourite web browser and enter the URL⁵:

192.168.0.235/mode/3/o

This sets digital pin 3 on the Arduino to be an output and should return the JSON output shown in Figure 13.



Figure 13 – JSON output for setting D3 as an output

Now if we go to the URL:

192.168.0.235/digital/3/1

We should see the LED connected to pin 3 light up and the following JSON output returned (see Figure 14).

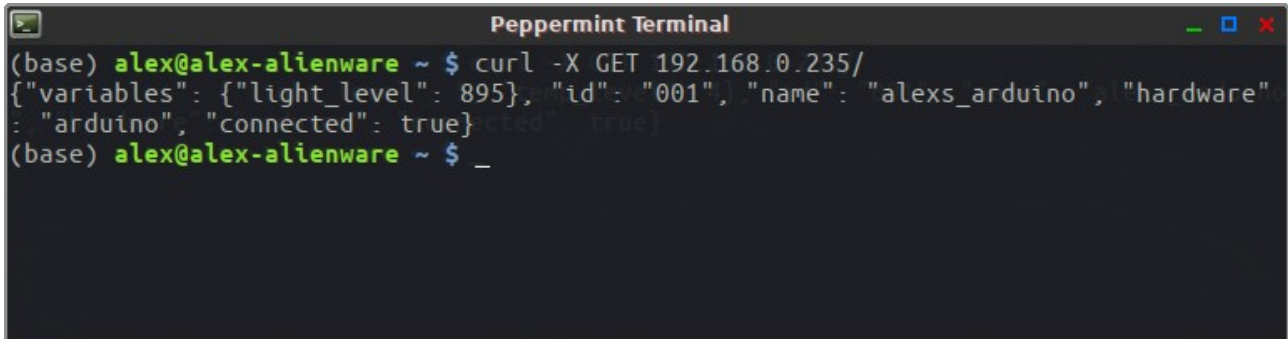


Figure 14 – JSON output for turning pin D3 on

⁵ Don't forget to change this to the IP address of your IoT device!

Ultimately, when we enter a URL using our web browser, what we are really doing is sending HTTP GET commands to the web service. Whilst we can carry on doing this using a web browser, I am going to use **curl**⁶ to send commands to the web service in the rest of these examples.

If we send an HTTP GET request for the web service root the aREST api provides us with some information about what variables are being monitored (see Figure 15).

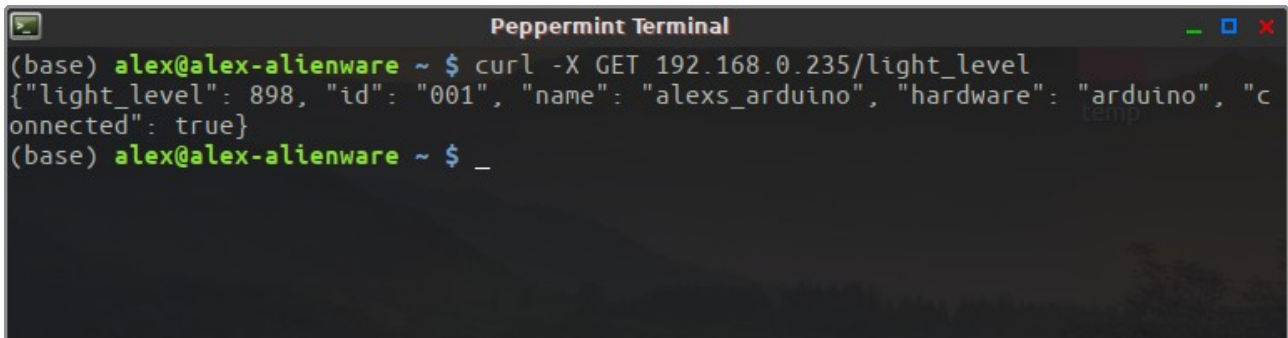


```

Peppermint Terminal
(base) alex@alex-alienware ~ $ curl -X GET 192.168.0.235/
{"variables": {"light_level": 895}, "id": "001", "name": "alexs_arduino", "hardware":
: "arduino", "connected": true}
(base) alex@alex-alienware ~ $ _
  
```

Figure 15 – HTTP GET of the web service root

We can see in Figure 15 that we are monitoring a variable called “light_level” (if we were monitoring more than one variable, they would all show up here). We can then send an HTTP GET request for just that variable to find out what it's current value is (see Figure 16).



```

Peppermint Terminal
(base) alex@alex-alienware ~ $ curl -X GET 192.168.0.235/light_level
{"light_level": 898, "id": "001", "name": "alexs_arduino", "hardware": "arduino", "c
onected": true}
(base) alex@alex-alienware ~ $ _
  
```

Figure 16 – HTTP GET of the “light_level” variable

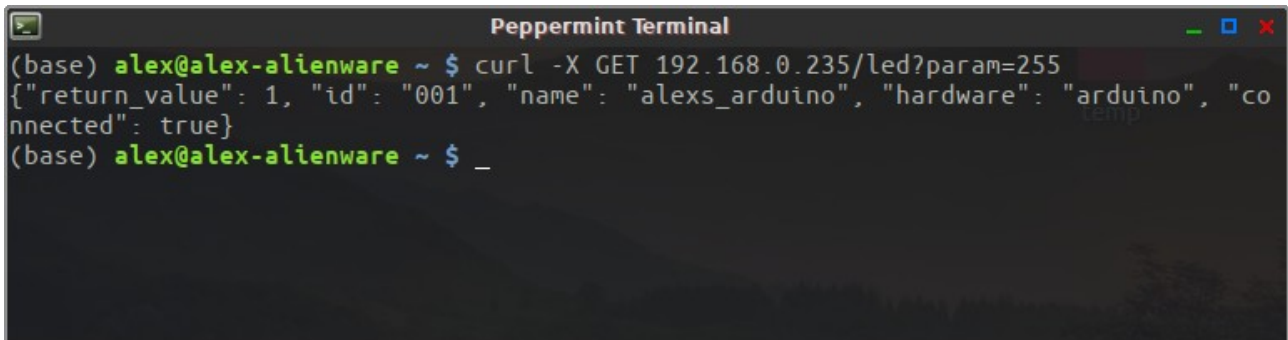
6 See <https://curl.haxx.se/> for more information - a copy of the windows version of curl is available on blackboard

Now we are going to remotely call the **led_control** function via the aREST api. To do this, we need to provide some parameters (i.e. the brightness of the LED as a PWM signal) to the method call.

Send an HTTP GET command with the following format (either via curl or your web browser):

192.168.0.235/led?param=255

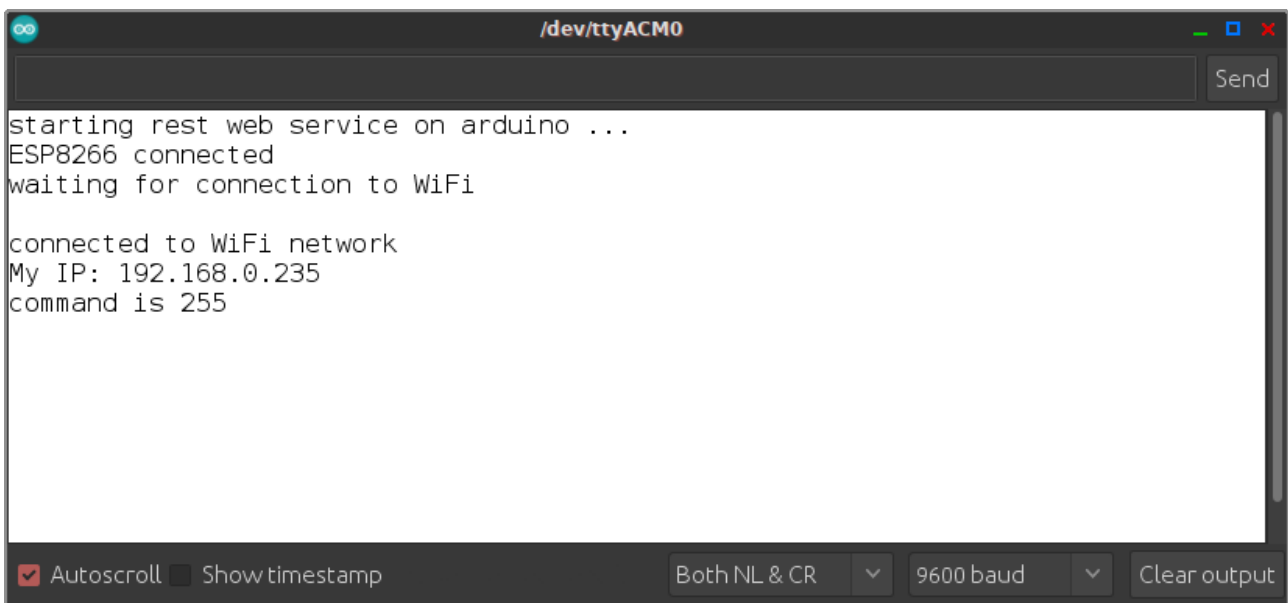
This will turn on the LED and set it to maximum brightness. Figure 17 shows the curl HTTP GET call and response, Figure 18 shows the debugging output in the serial monitor window, and Figure 19 shows the actual system.



```

Peppermint Terminal
(base) alex@alex-alienware ~ $ curl -X GET 192.168.0.235/led?param=255
{"return_value": 1, "id": "001", "name": "alexs_arduino", "hardware": "arduino", "connected": true}
(base) alex@alex-alienware ~ $ _
  
```

Figure 17 – HTTP GET command setting the brightness of the LED



```

/dev/ttyACM0
starting rest web service on arduino ...
ESP8266 connected
waiting for connection to WiFi

connected to WiFi network
My IP: 192.168.0.235
command is 255
  
```

Figure 18 – Serial monitor window showing the debugging output for the application (i.e. exactly what parameter has been passed to the **led_control** method)

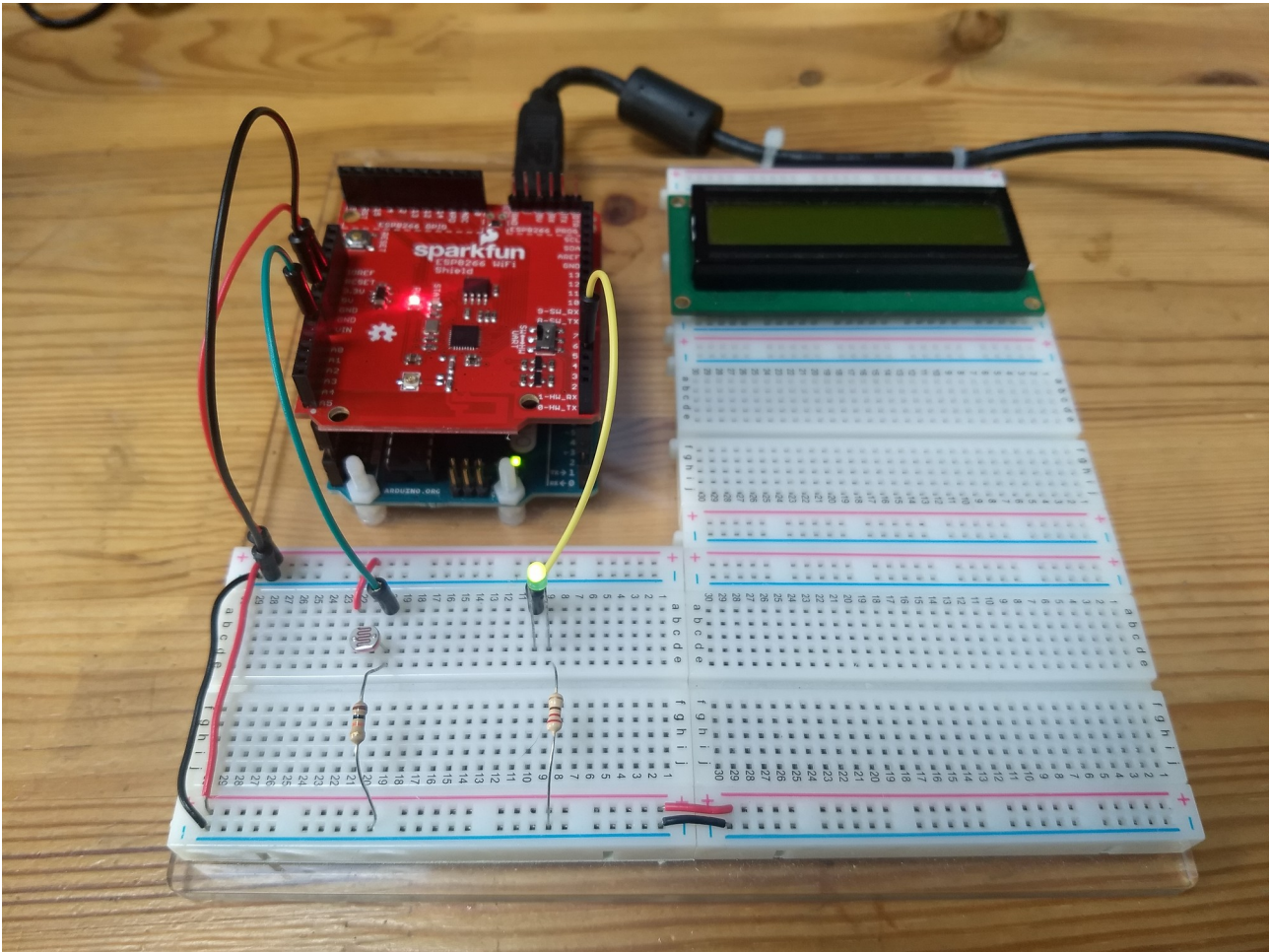


Figure 19 – The embedded web service with the LED turned on at maximum brightness

Now try modifying the HTTP GET request to set the brightness at a different level (e.g. try a value of 50 or 100).

Exercises

Now you are going to make some modifications to the program to add functionality:

1. Provide a calibration routine for the light sensor during the `setup()` function that means that the light value is remapped as a percentage.
2. Connect up a temperature sensor and potentiometer to some of the other analog pins and adapt the code above (in code listing 7) to monitor those variables too.
3. Add more LEDs to your system and write an additional method that cycles through them every time it is called. Make sure this is hooked in to the web service api.
4. Add an additional method to generate random dice rolls on the Arduino and return the random number. Again, hook this in to the web service api.

Check list

Task 1 – Program	
Task 1 – Web development exercises	
Task 2 – Program	
Task 2 – Exercise 1 (Potentiometer)	
Task 2 – Exercise 2 (Environment sensors)	
Task 2 – Exercise 3 (Push button)	
Task 3 – Program	
Task 3 – Exercise 1 (Calibration)	
Task 3 – Exercise 2 (Other sensors)	
Task 3 – Exercise 3 (More LEDs)	
Task 3 – Exercise 4 (Random numbers)	

Feedback