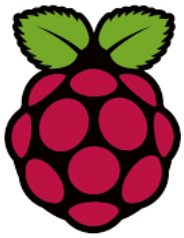

Tutorial:
How to create a web-based GUI to control your Raspberry Pi remotely over the internet.

Skill level: Basic beginner-level Python
Techniques used: Python3, Flask, GPIO
Hardware:

- Raspberry Pi 4
- LEDs (2 each red, green, blue, orange)
- 8 x 220 Ohm resistors
- Breadboard
- 8 jumper wires
- (optional) GPIO breakout board

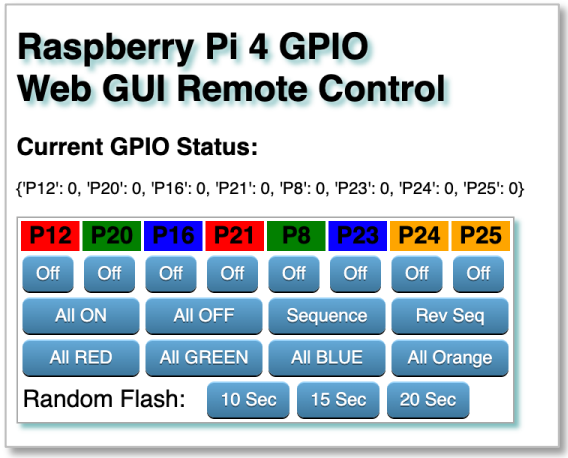

Fig 1: Screenshot of finished web interface

Contents

The purpose of this project was just to see if I could find a way to remotely control the GPIO pins on my Raspberry Pi 4. I discovered that it was relatively easy to control individual pins from the command line, but wanted a way to do so graphically via a Graphical User Interface (GUI).

This was a follow-up project after I created a GUI to control the board locally, based on Python and TKinter. However, that project, while successful, required me to either be physically logged into the Pi, or connected via VNCViewer.

So I decided to see if I could replicate that GUI, but make it accessible on the web, either on my own network, or globally via the web. It is possible to do this via PHP and Apache, but Python

comes with a very nifty native web server called Flask, and I discovered, after some trial and effort, that Flask is pretty easy to use.

While this current project just flashes some LEDs, it would be easy to modify the code and circuit to control other devices – lights, appliances, garage door openers, cat feeders, etc. For higher-voltage devices, you would just need to use transistors to deal with the higher current; this is well documented elsewhere, so we'll be sticking to just LEDs for demonstration purposes.

1. Create Physical Circuit

First, set up your physical circuit. While you don't absolutely need a breadboard, it will make circuit building vastly easier. I also used a pin breakout board to connect the pins to my breadboard, but this is optional.

One thing to note in all future references to pin numbers, we are using the GPIO numbering scheme, rather than the physical pin locations on the board. This website has a good reference diagram:
https://www.raspberrypi.org/documentation/usage/gpio/

You can also get a quick reference for your specific board by running this command at the terminal:

```
pinout
```

There are about 23 pins available for GPIO (general-purpose input/output) on the Pi 4. Your board may be slightly different. For this project, we just need 8 pins, so it should be possible to adapt this for older Raspberry Pi boards. I will show where to change this in code if necessary – it only requires editing one line of code.

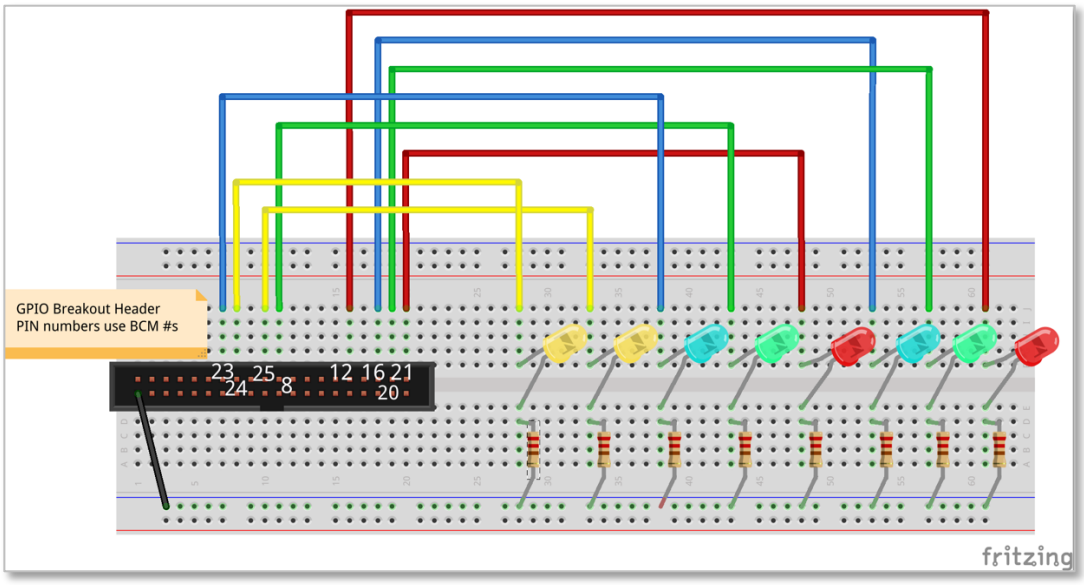

Fig 2: Physical layout of the circuit.

For simplicity, I chose to use all of the pins on the same side of the breakout board, but it really doesn't matter, as long as you document which LED is connected to which GPIO pin.
The +ve (longer) wire of the LEDs is connected to the jumper wires. These will receive 3 volts when switched on. The -ve wires connect to ground via the resistors.

2. Set up your software environment

From now on, we will assume your Raspberry Pi is connected and running, and connected to the same network as your desktop or laptop. For instance, you should both be on the same WiFi network. For running commands on the Pi, you can access via ssh, or if you have a keyboard and mouse connected you can enter code directly from your keyboard.

Open a terminal on your Pi and find its IP address:

```
ifconfig
```

If you are on WiFi, this will likely be labeled "wlan0:" It will likely be either 10.0.0.xxx or 192.168.xxx.xxx (where xxx is your specific numbers). Make a note of this IP number – we will need it later.

You will need to install a few additional packages that may not come standard with your Pi. The first is the GPIO library that allows you to control the pins on the board. Log into your Raspberry Pi, and from your command line type:

```
sudo apt-get install rpi.gpio
```

Next, install the Flask framework:

```
sudo apt-get install python3-flask
```

I should also note this is written for Python3 (or latest version). I have not tested it under V2.7.

Once you've installed the necessary software, it might be a good idea to just check everything works before writing any code. If you've followed my diagram and connected a red LED to pin 12, type python3 at your command line to enter the interactive Python shell. Then type only the commands after the >>> symbols below:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license"
for more information.
>>> import RPi.GPIO as GPIO
>>> GPIO.setmode(GPIO.BCM)
>>> GPIO.setup(12, GPIO.OUT)
__main__:1: RuntimeWarning: This channel is
already in use, continuing anyway.  Use
GPIO.setwarnings(False) to disable warnings.
>>> GPIO.output(12, GPIO.HIGH)
>>>
```

If all goes well, your red LED should light up. If not, verify you've connected the wires to the correct pins. Once it's working, you can exit the interactive Python shell by typing `exit()`

3. Start Flask web server

Now it's time to write some code. If you have a keyboard and monitor attached directly to your Pi, you can use one of the installed code editors, such as Geany or Thonny. You can also use your favorite command-line editor, such as Pico or even vi. Personally, since I'm remoting into my Pi, I write code in Atom on my desktop, then copy and paste into Pico in my terminal. But I'll leave that decision up to you.

Let's first set up a folder structure to contain and organize our code. Flask requires and expects this structure by default. In your home folder, create a folder called WebServer:

```
cd ~
mkdir WebServer
```

Within your new WebServer folder, create two new folders 'static' and 'templates':

```
cd WebServer
mkdir static
mkdir templates
```

We are going to have three files:
pi_webapp.py
This will contain all of our Python code.
templates/index.html
This is a normal html file, but we will use specially formatted placeholders to swap in dynamically generated code.
static/style.css

This is just an ordinary css file – no special codes allowed or needed.

Your finished structure, after you create your files, will look like this:

```
.
├── pi_webapp.py
├── static
│   └── style.css
└── templates
    └── index.html
```

To test that everything is setup and working, let's first just create a simple Hello World website. Using your preferred code editor, create a file called helloWorld.py in your WebServer folder. Then write this code and save your file:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
return 'Hello World!'
if __name__ == '__main__':
app.run(debug=True, port=80, host='0.0.0.0')
```

Now you need to start the application. At the terminal, type:

```
sudo python3 helloWorld.py
```

Next, open your laptop/desktop browser (remember you need to be on the same network as your Pi) and browse to the IP address of your Pi.

You should see a very basic web page, with just the text "Hello World!" If you look at the terminal, you should see the output from your webserver. Go ahead and refresh the page in your browser and watch the output change.

This confirms you have Flask installed and can successfully start the web server. Go ahead and shut it down for now by typing the standard script shutdown command Ctrl-C.

By the way, notice the line debug=True above. This is very useful during development, as it will attempt to show you any code errors in your browser window. You would of course did=sable this in a production application.

4. Build web GUI

You can delete helloWorld.py – we won't be needing it anymore. If you want a more step-by-step explanation of how Flask works, I suggest you look at this tutorial, particularly the section on Adding Dynamic Content:

https://projects.raspberrypi.org/en/projects/python-web-server-with-flask

We're now ready to build our web application. The complete code for the three files is linked at the bottom of this tutorial. In the following section, I'm just going to look at discrete parts, but we will assume you have loaded and verified all code in the correct files.

The first thing to notice in your html file **/templates/index.html** is that wherever we want to insert dynamic content, we surround it with curly braces:

```
<a href="/{{ x }}/off" class="btn btnOn">On</a>
```

In the case above, x is a variable that was passed to our template dynamically. We'll see how shortly.

If we want to run some Python code to dynamically generate content within our html, we surround these code snippets with <% %>

```
{% for x, y in pinStat.items(): %}
<th class="{{ x }}">{{ x }}</th>
{% endfor %}
```

Something to notice here is this is mostly Python code, but with a few changes. For instance, the **for** loop needs a closing **endfor**. Likewise, any **if** commands require a closing **endif**. This is obviously different from standard Python, but it should be easy enough to adapt to.

Let's take a look at the first few lines of pi_webapp.py

```
# ######
#»  Web Interface for Raspberry Pi 4 GPIO Control
# ######
import RPi.GPIO as GPIO
from flask import Flask, render_template, request
import time, random
app = Flask(__name__)
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
#define GPIO pins
pins = [12,20,16,21,8,23,24,25]
colors = ['red','green','blue','red','green','blue','orange','orange']
validColors = ['red', 'green', 'blue', 'orange']

# configure all pins for output
for i in pins:
»   GPIO.setup(i, GPIO.OUT)
```

Lines 4 – 7 just set up our environment. Line 8 tells our code that we are using the GPIO numbering scheme, rather than the physical position of the pins.

Lines 11 and 12 specify which LEDs are connected to which pins on your board:

```
pins = [12,20,16,21,8,23,24,25]
colors = ['red','green','blue','red','green','blue','orange','orange']
```

You should be able to see from this that my two red LEDs are connected to pins 12 and 21, and my two orange LEDs are connected to pins 24 and 25. These are the only lines you will need to change if you connect to different GPIO pins, or have different LED colors. These two lists are used in numerous places throughout the code.

Next we configure all of these pins as output:

```
for i in pins:
     GPIO.setup(i, GPIO.OUT)
```

This iterates through our '**pins'** list and sets each pin number to output. We'll use this system again and again throughout the code, so you can see why it's important to make sure lines 11 and 12 reflect the correct pins and LED colors of your physical setup.

We're not going to delve into the technical details of Flask here, so if you want to fully understand what "routes" are, check out the tutorial here:
https://projects.raspberrypi.org/en/projects/python-web-server-with-flask

However, let's look at a couple of lines where we configure the web server. On lines 23 and 24, we create a Python dictionary that checks and reports the current status of each GPIO pin. You can see how this looks in Fig 1 above. Here is what it initially contains:

```
{'P12': 1, 'P20': 0, 'P16': 0, 'P21': 0, 'P8': 0, 'P23': 0, 'P24': 0, 'P25': 0}
```

As you can see, P12 is on (remember we switched it on earlier during testing), so it is returning 1. All other pins are currently off, so return 0. Also notice on line 24 I prepend a letter 'P' in front of the pin number. This is because **variables cannot begin with a number**. Later in the code, when we just need the pin number again, we strip off that 'P' and convert the remaining number to int().

Let's look at lines 19-20 and then line 26:

```
19. @app.route("/")
20. def index():
```

… snip …

```
26.   return render_template('index.html', pinStatus=p, pins=pins,
colors=colors)
```

Line 19 tells the app that this should run at the web root – "/". But line 26 is where the magic happens. This instructs Flask to look in our /templates folder for a file named index.html, and to "render" it by swapping out any dynamic tags with the variables we pass to it. In this case, we pass three things:

pinStatus=p
this assigns the dictionary containing the status of all pins to the variable **pinStatus**
pins=pins, colors=colors
These pass both of our lists to the template, where we'll use them shortly.

Now, let's start our web server and look at the result:

```
sudo python3 pi_webapp.py
```

Once you run this command, go to your laptop/desktop browser and enter the IP address of your Raspberry Pi. You should see your GUI, and the dictionary containing pinStatus (highlighted in blue in Fig 3):

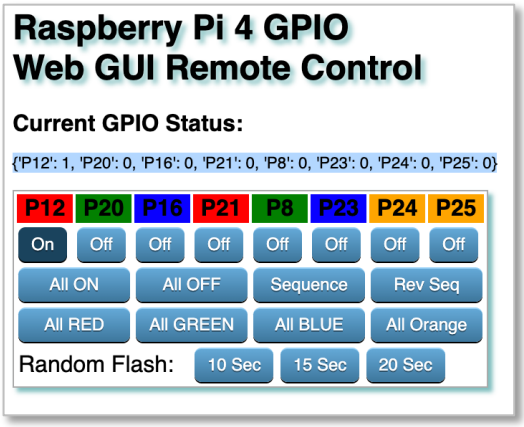

Fig 3: Finished web GUI.

Let's take a look at what happened our index.html template. On line 18 we had this code:

```
<p>{% print(pinStatus) %}</p>
```

When Flask renders this template, it sees the {% … %} block and processes the line of Python contained in this block. In this case, it prints out the contents of the pinStatus variable, which was passed in on line 26 of our script.

It's worth looking at the loop in the <style> block, lines 7 through 11.

```
{% for i in range(pins|length): %}
th.P{{ pins[i] }} {
  background-color: {{ colors[i] }} ;
}
{% endfor %}
```

The **for** loop is a little different. If this were regular Python, this would have been written:
`for i in range(len(pins))`

However, the engine that renders our template does not recognize that method, and instead uses its own slightly modified way to get the length of the **pins** list. When this loop is rendered, it iterates through the **pins** list, and creates a series of style rules. If you do "view source" on the web

page, you can see the result. Notice also that we have to explicitly tell when to end the **for** loop with **endfor** – in regular Python this would be implied by just un-indenting the next line, but we can't do that in the template file.

This is our basic template, before you configure any of the pins. The URL will be something like http://192.168.86.125/ (or whatever is the IP address of your Raspberry Pi). If you hover your mouse over any of the buttons, you will see what URL they call. For instance, if you hover over the button below the red P21, you'll see the new URL is http://192.168.86.125/P21/on

Clicking on that link causes Flask to run our second "route", defined on Line 29 of pi_webapp.py

```
@app.route("/<pinNum>/<action>")
```

This one, instead of "/", expects two parameters – pinNum and action – which it then passes into the function '**action(pinNum,action)**'. In this case, we pass it 'P21' and 'on'. Since we just want the number, we strip off the 'P', ad do some checking to verify this is a valid pin number.

The rest of the function looks at the second parameter, and decides which function to call. In this case, it calls function `on_off(12,'on')`

This function, on line 71, first verifies that the pin number is actually one of our pre-defined pins. This is necessary, since it's technically easy to enter a non-existent pin number in the URL. If it is valid, and the second parameter is 'on' then we set Pin 12 to HIGH, which turns it on, lighting up our red LED. The same function is used to turn the pin off.

The rest of the functions are pretty straightforward – most cycle through our '**pins**' list and either set the relevant pins to HIGH or LOW.

The remaining file in our application is /static/style.css. This is just a standard css file, and no dynamic code is allowed. That is why I created the dynamic style block in the head of the index.html template.

That's the end of this tutorial, and I hope you enjoyed building your web GUI for the Raspberry Pi. I encourage you to experiment and see what you can come up with, whether it's just playing with the css to style the remote control differently, or modifying the code to create different light effects. Or you could attach some sensors to the GPIO pins and read those inputs, printing the results to your web page.

Even though this tutorial only had you remote control a Raspberry Pi on your local network, there's no reason this wouldn't work from anywhere in the world if you know how to make your Pi's IP address accessible from outside your firewall. And if you replace the LEDs with transistors to control higher-current devices, this could be the basis of a DIY home automation system.

Download code for the three files from GitHub:
https://github.com/rorylysaght/RaspberryPi_Web_Remote_Control_GUI