LESSON 11

BUILDING A WEB SERVER WITH BOTTLE

OBJECTIVE

In this lesson you will learn how to create a web server on your Raspberry Pi using a software package named Bottle. Your projects can then be controlled by a web page, enhancing the functionality and overall user-interface capabilities.

MATERIALS

- Raspberry Pi connected to a monitor, keyboard, and mouse
- Circuit from Lesson 9, Activity #1 containing the RGB LED, IR Receiver, the Amplifier/Speaker, the Servo, and the MPU6050 Accelerometer/Gyro
  Note: You will only be using the RGB LED portion of this circuit. If you must rebuild the circuit, refer to Lesson 2, Activity #1, Steps 1 through 4 for building instructions.
- Wired or wireless network for connecting multiple devices.
- Web browser equipped device to display content from the Raspberry Pi web server.

REVIEW CONCEPTS

If you do not feel comfortable with the following concepts, please review them before proceeding.

- Connecting your Raspberry Pi to a Network (Lesson A-9)
- Using the Command Line (Lesson B-1)
- Determining the IP Address of your Raspberry Pi (Lesson C-1)

LESSON

In this lesson you will learn about different components required for establishing communication over a network. You will also learn how to use a Python package named Bottle to create and host a web page on your Raspberry Pi. This web page will later be used to control the color of your RGB LED from another device on your network.

> NOTE – While you will be using the term web server throughout this lesson, the server you create during this lesson will **not** be available outside of your home network. Making this server available outside your home network by modifying your home router or firewall settings would be far outside the scope of this lesson. Making any server on your home network available to the internet is a *major security risk* and should not be undertaken by anyone without extensive network security knowledge.

WEB PAGE CONTROL

The ability to control a project with a web page interface can make a project much more useful. Instead of controlling a program or hardware with a keyboard or mouse connected directly to the Pi, you can make the program control available to any device on your home network that can launch a web browser. These types of devices might include phones, tablets, smart TVs, gaming consoles, or any other device with a web browser.

The complexity of your web page might cause issues on certain devices. For example, a web page that can be properly displayed on a computer, might not display or function well on a phone due to the software support in the browser of the phone. Keeping web page controls simple is the best way to ensure the highest level of compatibility across all devices.

COMPONENTS REQUIRED FOR WEB PAGE CONTROL

The components required for web page control of a device are a server and a client.

The server refers to the hardware that is serving, or hosting, the web page. This hardware could be something as small like a Raspberry Pi, or a much larger device like a rack full of powerful computers. The server hardware required will depend on the amount of information that you want to make available. A large website like Google or Facebook requires entire buildings called data centers that are full of thousands of extremely fast servers that are working together to serve the website to everyone. Luckily, you only need a Raspberry Pi for hosting a couple of simple web pages.

A client refers to a device that connects to a server. The client might be a PC, tablet, phone, or any other device that can load a web page.

Depending on how it's used, a Raspberry Pi can be a server, a client, or both at the same time. When hosting a web page, it's considered a server. When accessing a web page using its web browser, then it would be considered a client. Since both of these things can happen at the same time, the Pi can be a server and a client, simultaneously.

IP ADDRESSES AND PORTS

During the lesson on remotely accessing your Pi, you learned how devices are identified on a network with a unique IP address. When a device is running a server, the content being hosted by the server can be accessed by other devices on the network by using the server's IP address. A device can host multiple servers at the same time, and port numbers are used to identify them.

Port numbers specify which server you want to communicate with, at a certain IP address. Port numbers will be between 0 and 65536, and will follow the IP address, separated by a colon:

Full Address	**IP Address**	**Port Number**
10.0.0.20:80	10.0.0.20	80
192.168.1.200:22	192.168.1.200	22
10.0.1.8:5900	10.0.1.8	5900

The VNC and SSH protocols that we used to access your Pi were running as servers on the Pi. Here are some of the more popular ports:

Protocol	**Port Number**
HTTP (Web traffic – insecure)	80
HTTPS (Web traffic – secure)	443
SSH	22
VNC	5900

These port numbers are the default port for each protocol listed. This means that you don't always have to specify a port when accessing these services. When trying to make a VNC connection to a VNC server running on 10.0.0.83, the VNC client software will automatically attempt to communicate using port 5900. If the server is up and responds to a request on port 5900, then a connection will be established with that server.

Web browsers will automatically attempt to communicate with a server on ports 80 and 443, and if a connection is established on one of those ports, the website content will be loaded in the web browser.

These default ports can easily be overridden in a client by supplying a different port number when connecting to the server. If a server at 10.0.0.100 has been configured to run SSH on port 6022 then you can connect to SSH by pointing your SSH client to 10.0.0.100:6022.

There can only be one server type assigned to a port number. This means that if you're running a web server on port 80, you cannot also run an SSH server on port 80. Attempting to assign multiple services to the same port will result in errors.

Some ports are also not available for use by all users. For instance, processes created by Thonny will run as the user named **`pi`**, which has limited permissions. The user named **`pi`** is not allowed to access certain ports like 80 and 443 out of concern for security. The options in this case are to run the process as root, which has other serious security considerations, or to select a different port to use for the server. Since there are around 65,000 ports to choose from, it's not too difficult to find an unassigned port in the upper area of this range. Port 8080 is a pretty common alternate port to use for web servers.

HTML CODE

HTML or Hypertext Markup Language is used for creating web pages. Just like Python, HTML code can be as simple of complicated as you would like. Here is an example of a simple HTML file:

```
<!DOCTYPE html>
<html>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

This file will display the message Hello World! in a web browser. The DOCTYPE code lets web browsers know that the code in the file is HTML so it can be properly displayed by the browser.

HTML TAG TYPES

HTML relies mostly on items called tags that look like this:

```
<tag>content</tag>
```

Both tags are enclosed with **<** and **>**, while the closing tag also includes a forward slash, or **/**. These tags are wrapped around some type of content. Below are descriptions of some of the more common tags you will find in HTML files:

<html> – This tag encloses all of the HTML code on the web page.

<head> - The head tag is used to apply a heading to the page. If present, this tag should be located before the body content.

<body> - The body tag contains the content for the main body of the web page.

<h1> - The heading tag is used to format text into six pre-defined heading sizes. Heading 1 (h1) is the largest heading font while Heading 6 (h6) is the smallest.

`<p>` - The paragraph tag is used to hold items that should be treated as paragraphs. Web browsers will apply extra spacing between paragraph elements to help with readability.

`<br>` - The break tag is used to insert a carriage return into a web page. This can help to add space when arranging items in a user-friendly layout.

`<a>` - The hyperlink tag is used to create a link to another web page or file. Unlike the tags above, the hyperlink tag requires more than just an opening and closing tag. The hyperlink tag also needs the text or image that will be used to display the link on the page, as well as the target location of the link. Here is an example of creating a link with the text **`Visit 42 Electronics`** that goes to **`www.42electronics.com`**:

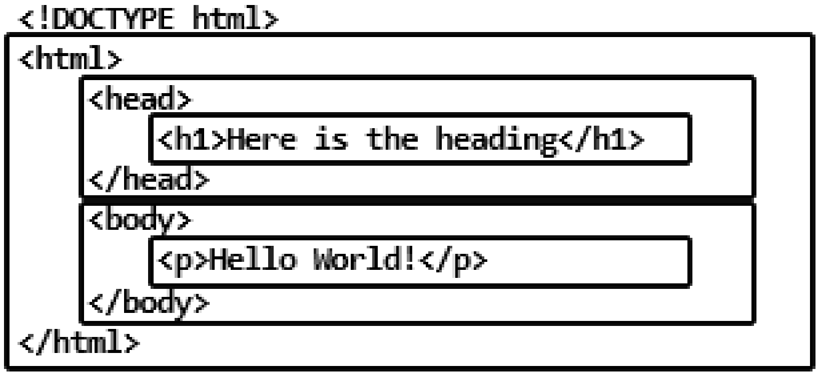
```
<a href="https://www.42electronics.com">Visit 42 Electronics</a>
```

The target link location is included in the opening **`<a>`** tag, followed by the text that will be displayed, and then the closing **`</a>`** tag.

Tags are nested inside of each other to form the content on the page:

```
<!DOCTYPE html>
<html>
    <head>
        <h1>Here is the heading</h1>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

Here is the same code with boxes drawn around each tag element:



```
<!DOCTYPE html>
<html>
    <head>
        <h1>Here is the heading</h1>
    </head>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

You can see that the **html** element holds both the **head** and **body** elements. The **h1** heading is located inside the **head** element, and the paragraph element is located inside the **body**.

Keeping opening and closing tags spaced at the same level will help keep the file organized. The file below is the exact same code as above, but the spaces have been removed:

```
<!DOCTYPE html>
<html><head><h1>Here is the heading</h1></head><body><p>Hello
World!</p></body></html>
```

The web page built by these two files will look identical in a web browser, but the file with proper spacing will be much easier to modify later, as opening and closing tag locations are much easier to identify. A missing opening or closing tag will completely break an HTML file, and this type of problem is much easier to fix when the code is formatted like the first example.

THE BOTTLE WEB SERVER

To serve HTML web pages, your device will need to run some type of server software. In upcoming lessons, you will learn about a web server called Apache that is very popular, but it also relies on an extensive software installation and configuration process. This may be much more than you need for hosting a simple web page, so a simple, lightweight alternative called Bottle was created.

Bottle requires the installation of a small Python module and will allow you to host web pages using a single Python program. The Bottle Python module will make web pages available whenever this Python program is running and will shut down when you end the program.

Bottle uses an element called a route to determine what will happen when a URL is requested. Here is an example of a route:

```
@route('/red_on')
def red_on():
    GPIO.output(13, GPIO.HIGH)
    return "Red On"
```

The path following **`@route`** determines what URL will trigger this block of code. In this case visiting **`ip_address:portnumber/red_on`** will cause this function to run. The function will push **`GPIO13`** high and **`return`** the message **`Red On`** to the web browser window.

It's always helpful to have a home page with links to your other pages. That way you can add new, clickable links to the home page and not have to remember the direct URL for every page that you have programmed into your web server. The URL for a home page is generally just **`ip_address:portnumber`** without any trailing value. The route for this page can be specified by a single forward slash:

```
@route('/')
```

This route will need to return all of the HTML for the home page, so some special formatting will be required. Triple, single quotation marks are used to indicate the beginning and end of your HTML code:

```
@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
        <p>Click the links below to change the LED color</p>
        <br>
        <a href="/red_on">Red On</a>
        <br>
        <a href="/red_off">Red Off</a>
        <br>
        <br>
        <a href="/cleanup">GPIO Cleanup</a>
    </body>
</html>
'''
```

Since HTML code will often contain spaces, carriage returns, and other special characters, the triple, single-quotation marks let Python know to ignore all of these formatting issues and return the entire block of triple-quoted code to the browser, where it can be rendered as an HTML web page.

The only other thing that Bottle needs to create a web server is a **run** command at the end of the program that specifies the IP address of the **host** server, and the **port** that the server will take requests from:

```
run(host='10.0.0.100', port=8080)
```

In the example above, the Bottle server will run on IP address 10.0.0.100 and port number 8080. The IP address must match the current network IP address of your Pi. If you're unsure of the current network IP address of your Pi, you can refer back to Lesson 1 where we determined the IP address for remote access purposes.

In summary, a bottle server just needs the following three elements to operate:

1. Import the bottle module
2. Define the routes that will be triggered when URLs are visited
3. Run command with the IP address and port of the server

THE PIP INSTALLER

Until this point, all Python packages used have been installed by cloning a GitHub repository and running some additional commands based on the package. There is another installer method available named PIP that stands for Package Installer for Python.

You may remember that when working with previous software packages you've installed using GitHub, your program must be located in the correct folder in order to find the files needed for import. This isn't a problem with the PIP installer as packages are installed directly into your Python environment, so there's no need to worry about folder locations. A Python program in any folder of your Raspberry Pi will have access to all of the PIP installed packages.

You're probably asking why you haven't always install using PIP as it sounds like a more user-friendly system. Unfortunately, not all software packages are not available on PIP, so often the GitHub cloning / installation method is the only option.

Just like **`python`** vs **`python3`** on the CLI, there are two versions of PIP that can be used, and they are:

`pip`	Python 2 package installer
`pip3`	Python 3 package installer

These commands operate like previous installers we've used in the past. If you use **`pip3`** to install a package, then it won't be available in Python 2 programs. The same is true for the inverse. If you use **`pip`** to install a package, then it won't be available to Python 3 programs. In these lessons, you will primarily use **`pip3`** so the packages will be available to the Python 3 programs that you create in Thonny.

Here are some of the more common commands that are available for PIP:

```
pip3 list
```

The **list** command will list every pip3 package that's installed on your system. This can be helpful if a tutorial has packages that must be installed, and you're not sure if they're already on your system.

```
pip3 show package_name
```

The **show** command will display all the version information for the **package_name** specified, as well as contact information for the author, and websites to reference for more information about the package.

```
pip3 install package_name
```

The **install** command will install the **package_name** that you specify, as long as the package you're trying is found. If the package name is incorrect or does not exist, this command will result in an error.

```
pip3 uninstall package_name
```

The **uninstall** command will uninstall the **package_name** that you specify, as long as the package is currently installed on your system. If the package is not installed, this command will result in an error.

ACTIVITIES

In the following activities, you will install the Bottle web server package, verify the installation, and then create a test web page to test the server functionality. You will then modify that program to first control just the red element of the RGB LED, and then add the ability to control the green and blue elements as well.

ACTIVITY #1 – PREPARING THE CIRCUIT AND INSTALLING BOTTLE

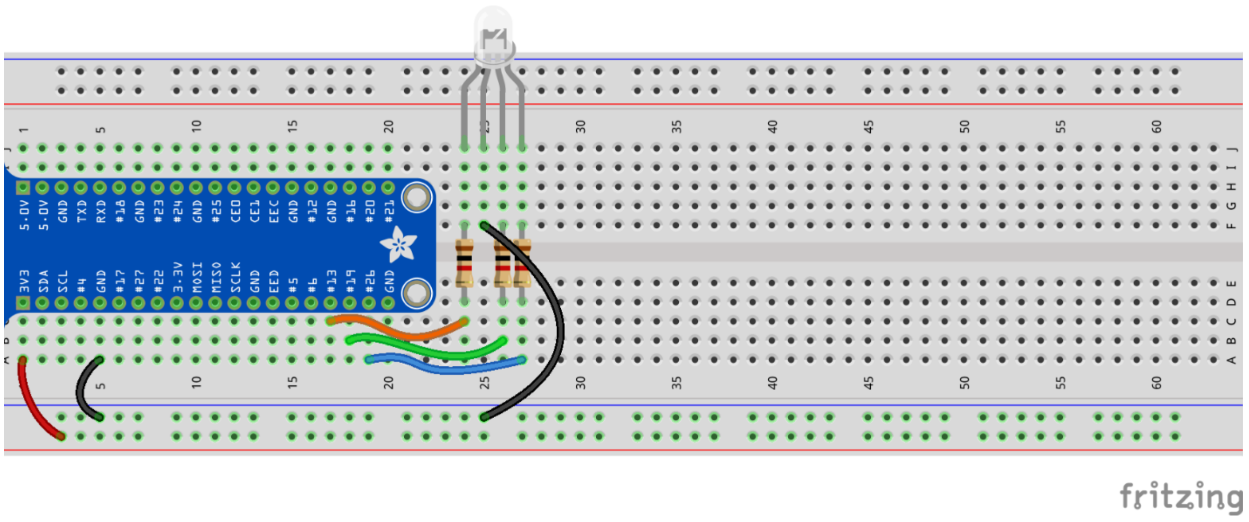
In this activity, you will remove some of the components from your breadboard, as they will not be needed for upcoming projects. The only components required for these activities are the RGB LED and its associated resistors and wiring.

This project will use the circuit from Lesson 9, Activity #1 as a starting point:

- If you still have this circuit built, then proceed with Step #1
- If you need to rebuild the circuit, build it using the instructions from Lesson 2, Activity #1, Steps 1 through 4 before proceeding with Step #2

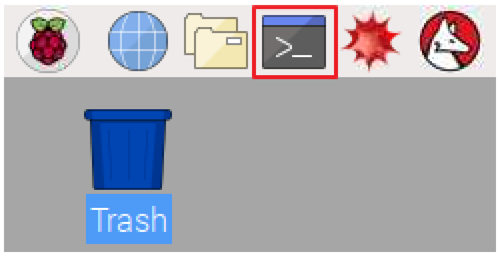
STEP #1

The first step will be to remove some of the components from your breadboard. With the Raspberry Pi powered off, remove the IR receiver, the amplifier and speaker, the servo, the MPU6050, and any wiring associated with these components. This will leave only the components required to run the RGB LED. When completed, your breadboard will look like this:



STEP #2

Now that the RGB LED is ready to run, you need to install the **`bottle`** web server package using the PIP installer. Power the Raspberry Pi back on and open a Terminal window using the icon in the upper menu bar:



STEP #3

You will be installing the bottle package using **`pip3`** so the package will be available using Python 3 in Thonny. Make sure your Pi is connected to the internet, and run the following command in the Terminal window:

```
pip3 install bottle
```

The bottle package will be installed without any additional prompts.

STEP #4

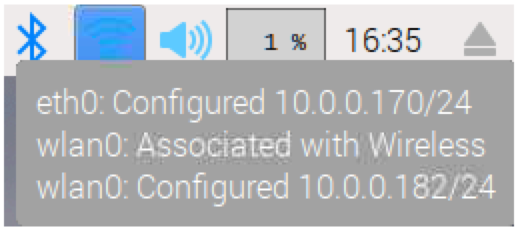
Verify that the bottle package was installed properly by using the show command to view the package information. Run the following command in your Terminal window:

```
pip3 show bottle
```

The output from this command should show all the information about the installed version of **`bottle`**. If there is no information printed to the console, that means the **`bottle`** was not found on your system. Go back to Step #3 to reinstall **`bottle`** and run the **`show`** command again to confirm that the installed version information is reported.

STEP #5

Before you can build a web page to display using the bottle web server, you will need to determine the current IP address of your Raspberry Pi on your home network. The easiest way to do this is to hover your mouse over the network icon in the upper-right menu bar of the desktop:



```
eth0: Configured 10.0.0.170/24
wlan0: Associated with Wireless
wlan0: Configured 10.0.0.182/24
```

This Pi is reporting an IP address of **`10.0.0.170`** for the wired network on **`eth0`**, and **`10.0.0.182`** for the wireless network on **`wlan0`**. You will likely only have only wired or wireless, but not both. It doesn't matter which interface you use, just make note of the IP address listed for one of the interfaces you are using in this menu, as it will be used to configure the Bottle server, and later to connect to your Pi with a web browser.

STEP #6

The next step will be to create a web server program in Python using bottle. Open up Thonny and save a new program under the name **`web_test.py`**. Once this file is created, add the following lines of code to the file:

```
from bottle import route, run

@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
    <p>Hello World!</p>
    </body>
</html>
'''

run(host='your_ip_address', port=8080)
```

This code will import the methods named **`route`** and **`run`** from the **`bottle`** module. A **`route`** will be configured for the home page located at **`/`**. The page consists of the HTML code required to print **`Hello World!`** on a web page. The **`run`** command will start the server using the specified IP address and port number **`8080`**. Be sure to replace **`your_ip_address`** with the IP address that you located in Step # 5.

STEP #7

The last step will be to start the bottle web server. Start the server by using the Run button in Thonny. The console will display information about the server that you're starting up:

```
Bottle v0.12.16 server starting up (using WSGIRefServer())...
Listening on http://10.0.0.169:8080/
Hit Ctrl-C to quit.
```

Now that the server is up and running, use another device on your home network to access the web server on your Raspberry Pi. Open a browser on another device connected to your home network and type the IP address and port number of the server into your browser separated by a colon. For the example above, the address would be:

```
10.0.0.169:8080
```

Going to this address from another device on the network will result in a very simple web page being displayed:

```
Hello World!
```

If your web page does not look like this, double-check your code with Step #6 and confirm that you are using the correct IP address on both the **`run`** line of the program, as well as that same IP address when accessing from another device.

Here is table with some examples of IP addresses, the run code, and the address to use when accessing your Pi from another device:

Raspberry Pi IP Address	**Run Code**	**URL for Accessing Server from Another Device**
10.0.0.169	run(host='10.0.0.169', port=8080)	10.0.0.169:8080
192.168.1.200	run(host='192.168.1.200', port=8080)	192.168.1.200:8080
10.1.1.16	run(host='10.1.1.16', port=8080)	10.1.1.16:8080

Ensure that you can connect to your web page before proceeding, as this program will be used as a base for the next activity.

ACTIVITY #2 – MODIFYING THE PROGRAM TO CONTROL THE RGB LED

In this activity, you will modify the program that you built in Activity #1 to include code that will allow for control of the red element of the RGB LED.

STEP #1

The first step will be to create a new copy of **`web_test.py`** to use as a base for our new program. This will leave **`web_test.py`** unmodified so it can be used as a base for new programs in the future. With **`web_test.py`** open in Thonny, use **`Save As`** to save a new copy of the program and name it **`web_rgb_red.py`**.

STEP #2

The first modification to the program will be to import the RPi.GPIO module so you can use it to control the red element of the RGB LED. Import the RPi.GPIO module just below the import of bottle. The new line has been highlighted below:

```
from bottle import route, run
import RPi.GPIO as GPIO

@route('/')
```

STEP #3

The next change will be to set the GPIO pin mode to BCM and configure GPIO13 as an output. Add the highlighted lines below to your program, just below the import section:

```
from bottle import route, run
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.OUT)

@route('/')
```

STEP #4

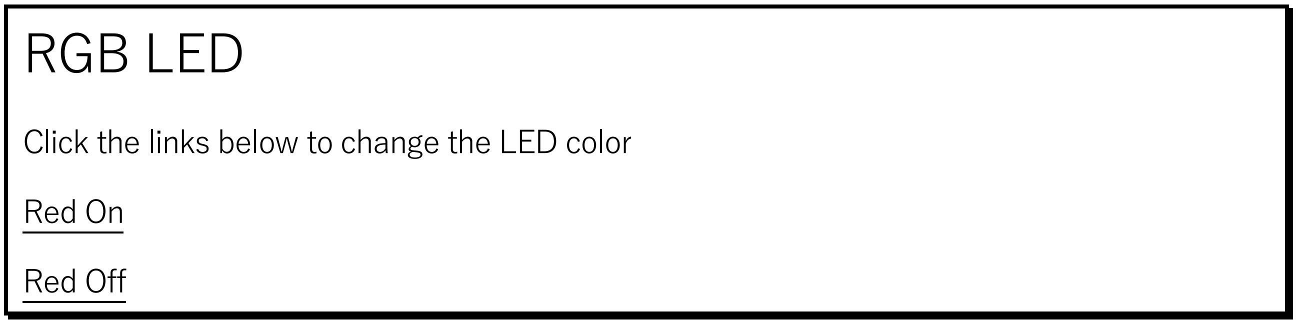
The only route currently defined in our web server file is the **home** route located at **/**. You will need to define some new routes for turning GPIO13 on and off. Add the highlighted routes below to your program between the existing **home** route and the **run** command:

```
</html>
'''

@route('/red_on')
def red_on():
    GPIO.output(13, GPIO.HIGH)
    return "Red On"

@route('/red_off')
def red_off():
    GPIO.output(13, GPIO.LOW)
    return "Red"

run(host='your_ip_address', port=8080)
```

The route named **/red_on** will push **GPIO13** high and print the words **Red On** in the web browser. The route named **/red_off** will pull **GPIO13** low and print the words **Red Off** in the web browser.

STEP #5

The routes to turn the LED on and off are set, and they can be accessed by going to **`/red_on`** and **`/red_off`** when the server is running, but you can modify the home page to include clickable links to these pages. Modify the HTML code for the home page to give you the web page layout below:



RGB LED

Click the links below to change the LED color

Red On

Red Off

The underlines indicate that this text is a link that can be clicked on in a web browser. The **`Red On`** text will link to **`/red_on`** and the **`Red Off`** text will link to **`/red_off`**.

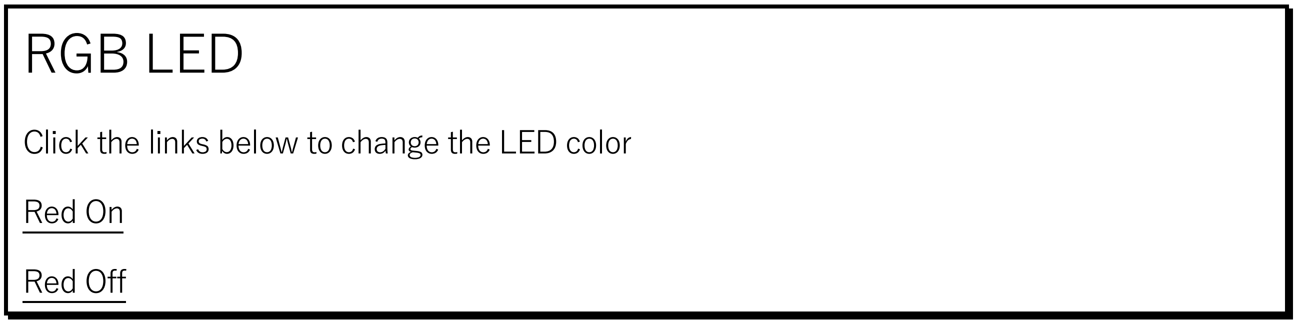
Change the code inside of the body tags of the home route to the highlighted block of code below:

```
@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
        <h1>RGB LED</h1>
        <p>Click the links below to change the LED color</p>
        <a href="/red_on">Red On</a>
        <br>
        <br>
        <a href="/red_off">Red Off</a>
    </body>
</html>
'''
```

The **`<h1>`** tag will make the heading larger than the rest of the text on the page, and the **`<br>`** tags will add carriage returns between the links.

STEP #6

Press the **`Run`** button in Thonny and access **`your_ip_address:8080`** using the web browser on another device connected to your home network. Make sure to use the IP address of your Raspberry Pi in place of **`your_ip_adddress`**. If you still have this page open from Activity #1, a quick refresh or reload in the browser will load the updated HTML content. Your web page should look like this:



RGB LED

Click the links below to change the LED color

Red On

Red Off

Clicking on the **`Red On`** or **`Red Off`** links will turn the red element of the RGB LED on and off. After clicking one of the links, use the back button in your browser to return to the home page. Use the **`Stop`** button in Thonny to stop the program when you're done testing.

You will notice that you don't yet have any GPIO cleanup code in this program, so you will receive GPIO errors after running this program since it's not cleaning up the GPIO pins. You will add the ability to run a GPIO cleanup in Activity #3.

If you're program does not behave as expected, double-check that it matches the full program below:

```
from bottle import route, run
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(13, GPIO.OUT)

@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
        <h1>RGB LED</h1>
        <p>Click the links below to change the LED color</p>
        <a href="/red_on">Red On</a>
        <br>
        <br>
        <a href="/red_off">Red Off</a>
    </body>
</html>
'''

@route('/red_on')
def red_on():
    GPIO.output(13, GPIO.HIGH)
    return "Red On"

@route('/red_off')
def red_off():
    GPIO.output(13, GPIO.LOW)
    return "Red Off"

run(host='your_ip_address', port=8080)
```

This program will be used as a base for the next activity, so do not procced to the next activity until your program is working as expected.

ACTIVITY #3 – ADDING THE GREEN AND BLUE ELEMENTS

In this activity, you will modify the program that you built in Activity #2 to add the green and blue elements of the RGB LED. The HTML code will be modified to include links for red, green, blue and GPIO cleanup:

Red – turns the red element on and the other elements off

Green – turns the green element on and the other elements off

Blue – turns the blue element on and the other elements off

GPIO Cleanup – resets all GPIO pins back to default state (server will need to be reset after clicking this link)

STEP #1

The first step will be to create a new copy of **`web_rgb_red.py`** to use as a base for your new program. This will leave **`web_rgb_red.py`** unmodified so it can be used as a base for new programs in the future. With **`web_rgb_red.py`** open in Thonny, use **`Save As`** to save a new copy of the program and name it **`web_rgb_all.py`**.

STEP #2

The first modification will be to add some variables that will hold the pin numbers for each color. This will help to keep the program more organized. You will use the same variable name and pin assignments as previous programs that have interacted with the RGB LED:

- red = 13
- green = 19
- blue = 26

Add the highlighted block below to your program, just below the import block:

```
from bottle import route, run
import RPi.GPIO as GPIO

red = 13
green = 19
blue = 26

GPIO.setmode(GPIO.BCM)
```

STEP #3

The first modification will be to configure all of the GPIO lines connected to the RGB LED as outputs. You can use the color names that were configured in the last step. Change the first setup line to use **`red`** instead of **`13`** and add lines for **`green`** and **`blue`** below that:

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)

@route('/')
```

STEP #4

Since you will be updating all three elements of the RGB LED, it will be useful to have a function that can update all three elements from one line of code in the main program. This is the same function you've used in earlier programs. Add the highlighted lines of code below to you program, just after the GPIO setup block:

```
GPIO.setup(blue, GPIO.OUT)

def led_update(red_value,green_value,blue_value):
    GPIO.output(red, red_value)
    GPIO.output(green, green_value)
    GPIO.output(blue, blue_value)

@route('/')
```

This function will be called with the high or low values for the red, green, and blue elements, in that order. Here is the output table for the RGB LED:

Function Call	Red Element	Green Element	Blue Element
led_update(1,0,0)	On	Off	Off
led_update(0,1,0)	Off	On	Off
led_update(0,0,1)	Off	Off	On
led_update(0,0,0)	Off	Off	Off

This will greatly simplify the code needed to change the state of LED elements later in the routes.

STEP #5

You will now be modifying the HTML code on the home page to include links for green, blue, and GPIO cleanup. The GPIO cleanup link will help you to avoid a lot of GPIO errors from being displayed whenever you restart the server or run another program that needs to use GPIO pins.

You will be removing the **`/red_off`** link and adding the new links mentioned above. Make the following, highlighted changes to your HTML code in the **`/`** route:

```
@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
    <h1>RGB LED</h1>
    <p>Click the links below to change the LED color</p>
    <a href="/red_on">Red</a>
    <br>
    <a href="/green_on">Green</a>
    <br>
    <a href="/blue_on">Blue</a>
    <br>
    <br>
    <a href="/cleanup">GPIO Cleanup</a>
    </body>
</html>
'''
```

STEP #6

The **/red_on** route is currently only turning on the red element of the RGB LED. This route will need to be modified to include the ability to turn the green and blue elements off using the **led_update()** function that you created earlier. The **/red_off** route will no longer be needed so it will also be removed in this step.

Remove the existing GPIO.output command from the **/red_on** route and replace it with a call to the **update_led()** function that will pass values of **1**, **0**, and **0** for the status of the **red**, **green**, and blue RGB LED elements. Also, remove the route and function for **/red_off** as they will no longer be needed:

```
@route('/red_on')
def red_on():
    led_update(1,0,0)
    return "Red On"

run(host='your_ip_address', port=8080)
```

STEP #7

Next, you will duplicate the **`/red_on`** route and function, but for the green and blue elements. The **`/green_on`** route will turn on the green RGB LED element and the **`/blue_on`** route will turn on the blue RGB LED element. Add the following highlighted routes to your program just after the **`/red_on`** route:

```
@route('/red_on')
def red_on():
    led_update(1,0,0)
    return "Red On"

@route('/green_on')
def green_on():
    led_update(0,1,0)
    return "Green On"

@route('/blue_on')
def blue_on():
    led_update(0,0,1)
    return "Blue On"

run(host='your_ip_address', port=8080)
```

STEP #8

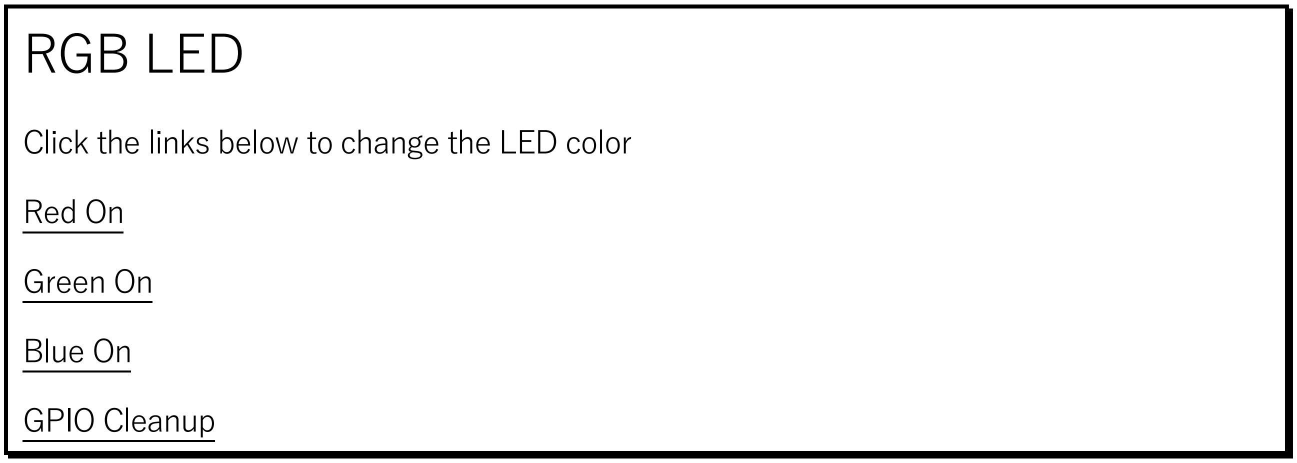
The last program modification will be to add a /cleanup route that will turn all RGB LED elements off, as well as run a **`GPIO.cleanup()`** function. The message GPIO Cleanup Completed will also be printed to the console, letting the user know that the cleanup has been completed.

Add the following highlighted route to your program, between the **`/blue_on`** route and the **`run`** command:

```
@route('/blue_on')
def blue_on():
    led_update(0,0,1)
    return "Blue On"

@route('/cleanup')
def cleanup():
    led_update(0,0,0)
    GPIO.cleanup()
    return "GPIO Cleanup Completed"

run(host='your_ip_address', port=8080)
```

STEP #9

Press the **`Run`** button in Thonny and access **`your_ip_address:8080`** using the web browser on another device connected to your home network. Be sure to use the IP address of your Raspberry Pi in place of **`your_ip_address`**. If you still have this page open from Activity #2, a quick refresh or reload in the browser will load the updated HTML content. Your web page should look like this:



RGB LED

Click the links below to change the LED color

Red On

Green On

Blue On

GPIO Cleanup

Clicking on the **`Red On`**, **`Green On`**, or **`Blue On`** links will turn on that RGB LED element. After clicking one of the links, use the back button in your browser to return to the home page. When you're done changing the color of the LED, use the **`GPIO Cleanup`** link to set the GPIO pins back to their default state before using the **`Stop`** button in Thonny to stop the server.

If you're program does not behave as expected, double-check that it matches the full program below:

```
from bottle import route, run
import RPi.GPIO as GPIO

red = 13
green = 19
blue = 26
GPIO.setmode(GPIO.BCM)
GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)

def led_update(red_value,green_value,blue_value):
    GPIO.output(red, red_value)
    GPIO.output(green, green_value)
    GPIO.output(blue, blue_value)

@route('/')
def home():
    return '''
<!DOCTYPE html>
<html>
    <body>
    <h1>RGB LED</h1>
    <p>Click the links below to change the LED color</p>
    <a href="/red_on">Red</a>
    <br>
    <a href="/green_on">Green</a>
    <br>
    <a href="/blue_on">Blue</a>
    <br>
    <br>
    <a href="/cleanup">GPIO Cleanup</a>
    </body>
</html>
'''

@route('/red_on')
def red_on():
    led_update(1,0,0)
    return "Red On"

@route('/green_on')
```

```
def green_on():
    led_update(0,1,0)
    return "Green On"

@route('/blue_on')
def blue_on():
    led_update(0,0,1)
    return "Blue On"

@route('/cleanup')
def cleanup():
    led_update(0,0,0)
    GPIO.cleanup()
    return "GPIO Cleanup Completed"

run(host='your_ip_address', port=8080)
```

QUESTIONS FOR UNDERSTANDING

1. Can multiple servers on a device be assigned to the same port?

2. What web address would be used to access a server running on port 8042 of a device located at IP address 192.168.1.42?

3. If you used the command **`pip install bottle`** to install the bottle web server package, will this package be available to Python 3 programs that you create in Thonny?

Answers can be found on the next page.

ANSWERS TO THE QUESTIONS FOR UNDERSTANDING

1. *Can multiple servers on a device be assigned to the same port?*

   ANSWER: No. The port number is used to identify the server that will respond to a request on a specific port. Attempting to assign multiple servers to the same port will result in errors because both servers will be trying to respond to requests using that port.

2. *What address would be used to access a server running on port 8042 of a device located at IP address 192.168.1.42?*

   ANSWER: The address 192.168.1.42:8042 would be used to access a server running on port 8042 of IP address 192.168.1.42.

3. *If you used the command* **`pip install bottle`** *to install the bottle web server package, will this package be available to Python 3 programs that you create in Thonny?*

   ANSWER: No. Packages installed using **`pip`** will only be available in the Python 2 environment. If a package is to be used in Python 3, the **`pip3`** command must be used for installation.

CONCLUSION

In this lesson, you learned how to use the Bottle package in Python to create a web server with the ability to control an LED from another device on your home network. This project could be modified to control other types of electronic components that you have connected to your Raspberry Pi in previous lessons.

In the next lesson, you will learn to use a camera connected to the Pi to and take pictures and display those using a GUI interface.