



## **Developer Study Guide**

### **Bluetooth® Internet Gateways**

#### **LE Peripherals**

Release : 2.0.1

Document Version: 2.0.0

Last updated : 24th June 2021

# Contents

<b>1. REVISION HISTORY .....</b>	<b>3</b>
<b>2. INTRODUCTION.....</b>	<b>4</b>
<b>3. PREREQUISITES .....</b>	<b>4</b>
3.1 Equipment .....	4
3.2 Knowledge and Experience .....	4
<b>4. PROJECT - BUILDING A BLUETOOTH® INTERNET GATEWAY FOR LE PERIPHERALS .....</b>	<b>5</b>
4.1 A Generic Physical Architecture .....	5
4.2 Refining the Physical Architecture for Bluetooth LE Peripheral Devices .....	5
4.3 Selected Physical Architecture .....	6
4.4 Gateway Development .....	9
4.4.1 Platform Set-up .....	9
4.4.2 Apache Web Server.....	11
4.4.3 websocketd .....	15
4.4.4 Bluetooth API .....	18
4.4.4 Adapter Code Development .....	19
<b>5. PROJECT - BUILDING GATEWAY APPLICATIONS .....</b>	<b>58</b>
5.1 Introduction .....	58
5.2 Explore Blue .....	58
5.2.1 Task - Set up Explore Blue .....	58
5.2.2 Task - Experiment with Explore Blue .....	59
5.3 Task - Develop the Micro:bit Controller Application .....	63
5.3.1 Overview .....	63
5.3.2 Preparation .....	63
5.3.3 Task - Completing the micro:bit controller application .....	67
<b>6. SECURITY .....</b>	<b>70</b>
6.1 Warning! .....	70
6.2 Further Security Measures for the LE Peripherals Gateway .....	71
6.3 Encrypted web sockets communication .....	72
6.4 Authentication .....	72
6.5 Reverse Proxying and the LE Peripherals Gateway .....	73

6.5.1 Web server configuration changes .....	73
6.5.2 Adjusting the gateway applications .....	73
6.6 Task - Securing the Bluetooth Devices .....	74
6.5.1 Pairing with the gateway .....	74
6.5.2 The Bluetooth firewall .....	75
<b>7. SCALABILITY .....</b>	<b>81</b>
<b>8. CLOSE .....</b>	<b>81</b>

## 1. Revision History

Version	Date	Author	Changes
1.0.0	18th November 2020	Martin Woolley Bluetooth SIG	Initial version
1.0.1	6 <sup>th</sup> January 2021	Martin Woolley Bluetooth SIG	Added <i>enabled</i> configuration property to the Bluetooth firewall.
2.0.0	24 <sup>th</sup> June 2021	Martin Woolley Bluetooth SIG	<p><b>Release</b></p> <p>Added new module covering gateways for Bluetooth mesh networks.</p> <p>Modularised the study guide to accommodate the two main cases of LE Peripherals and mesh networks.</p> <p>Updated to be based on Python 3 rather than Python 2.</p> <p><b>Document</b></p> <p>Removed content which is now located in module 02 - First Steps.</p> <p>Updated physical architecture to more accurately reflect the component parts used and their interactions.</p> <p>Updated code fragments to be based on Python 3.</p> <p>Updated instructions for setting up and running the <i>motion</i> daemon so it runs as a non-root user.</p>

## 2. Introduction

This module continues on from module 02 *First Steps* and covers the completion of the hands-on project to develop a Bluetooth internet gateway for LE Peripheral devices.

## 3. Prerequisites

### 3.1 Equipment

To complete the practical work in this module you will need some equipment, and the following table lists the hardware and platform software that was used by the author in creating this resource. You are free to use compatible alternatives, but the information in the module relates to the items listed here.

Item	Version(s)	Comments
Raspberry Pi	4 Model B	To complete the practical work relating to scalability, at least two Raspberry Pi devices are required. If you do not wish to carry out these exercises, a single Raspberry Pi will suffice.
Linux, Raspbian distribution	Kernel version 4.19.97	Created using the Raspberry Pi Imager tool. The current latest version should be OK.
BlueZ Bluetooth stack	5.50	This is the version that was included in Raspbian, installed by the Raspberry Pi Imager when this study guide was created.
Python 3	3.7.3	Later releases of Python version 3 should be OK.
BBC micro:bit	Any	A micro:bit is used as an example Bluetooth device which the gateway provides access to. You may use any Bluetooth Low Energy peripheral, provided you are familiar with its GATT services and characteristics and are willing to adapt the exercises accordingly.

Other software will be installed or created in the practical exercises themselves and where appropriate, version information will be provided at that point.

### 3.2 Knowledge and Experience

To get the best out of this resource, you should already understand the fundamental concepts relating to Bluetooth Low Energy (LE) when used with GAP, GATT and ATT. If any of these terms are

new to you then you are advised to read the *Bluetooth Low Energy - Basic Theory* module in the Bluetooth LE Developer Study Guide, which you can download from <https://www.bluetooth.com/bluetooth-resources/bluetooth-le-developer-starter-kit/>.

You should also be comfortable issuing basic commands from a Linux shell and understand Linux fundamentals such as users and permissions. There are numerous Linux tutorials for beginners available on the internet if you need to acquire this knowledge first.

## 4. Project - Building a Bluetooth® Internet Gateway for LE Peripherals

In this section, we'll continue with the work initiated in section 7 of module 02 *First Steps*. In module 02 we'd got as far as considering requirements, had defined a logical architecture and a tentative physical architecture. We'd noted though that when examined at a more detailed level, we'd need the physical architecture to vary depending on whether we were building a gateway for LE Peripheral devices or Bluetooth mesh nodes.

### 4.1 A Generic Physical Architecture

The physical architecture defined in module 02 is repeated here in Figure 1.

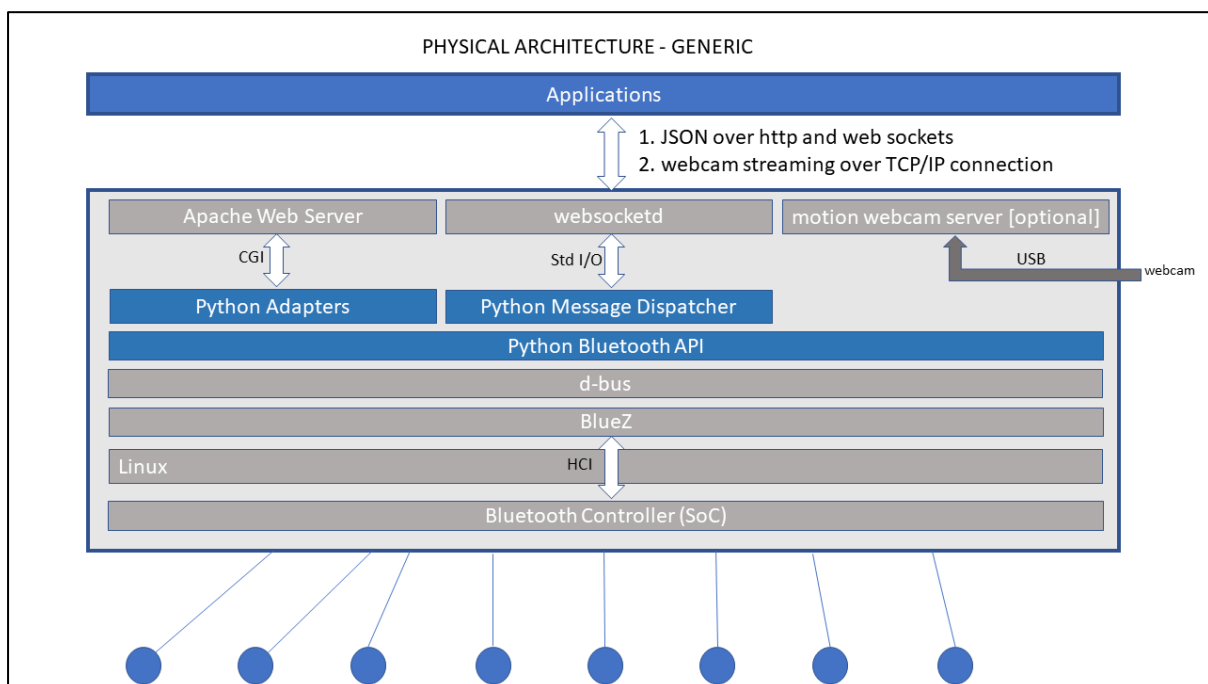


Figure 1 - Generic Physical Architecture for a Bluetooth Internet Gateway

### 4.2 Refining the Physical Architecture for Bluetooth LE Peripheral Devices

BlueZ provides an API for use with GAP/GATT Peripheral and Central devices which involves a Linux system service called *bluetoothd*. For Bluetooth mesh, a different service called *bluetooth-meshd* is used. The *bluetoothd* daemon and the *bluetooth-meshd* daemon may not be run at the same time on the same Linux computer. One of the reasons for this is that they each serve to queue and serialise requests from applications to and from the (usually) single Bluetooth controller that the

device has. If both daemons were running at the same time, it would not be possible in the current implementations of each for that serialised use of the Bluetooth controller to be achieved.

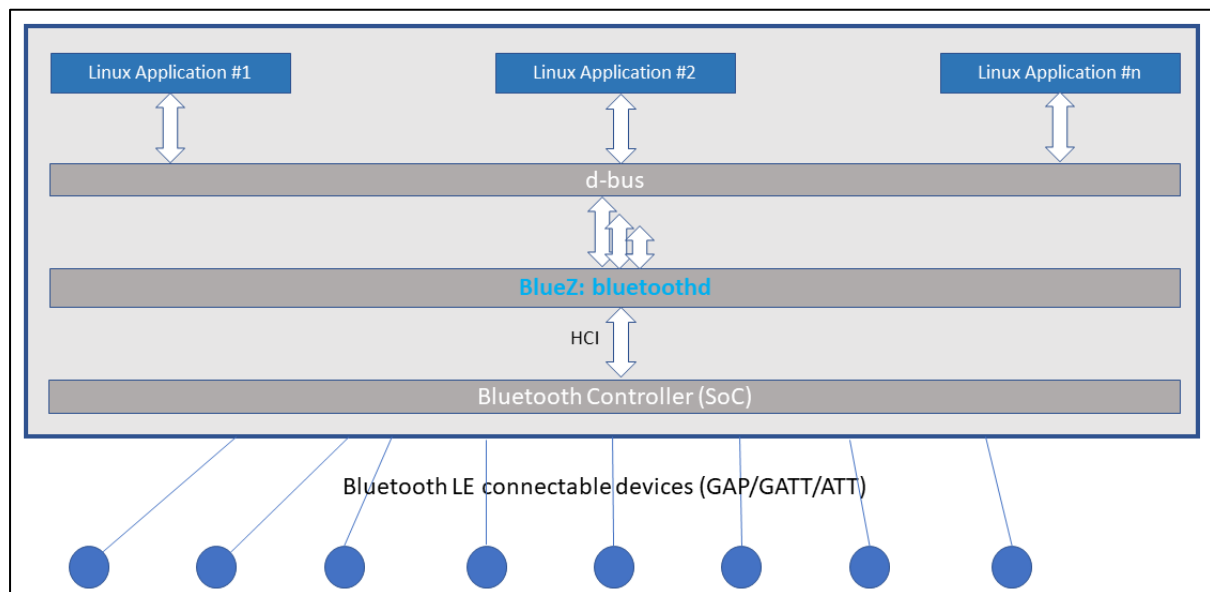


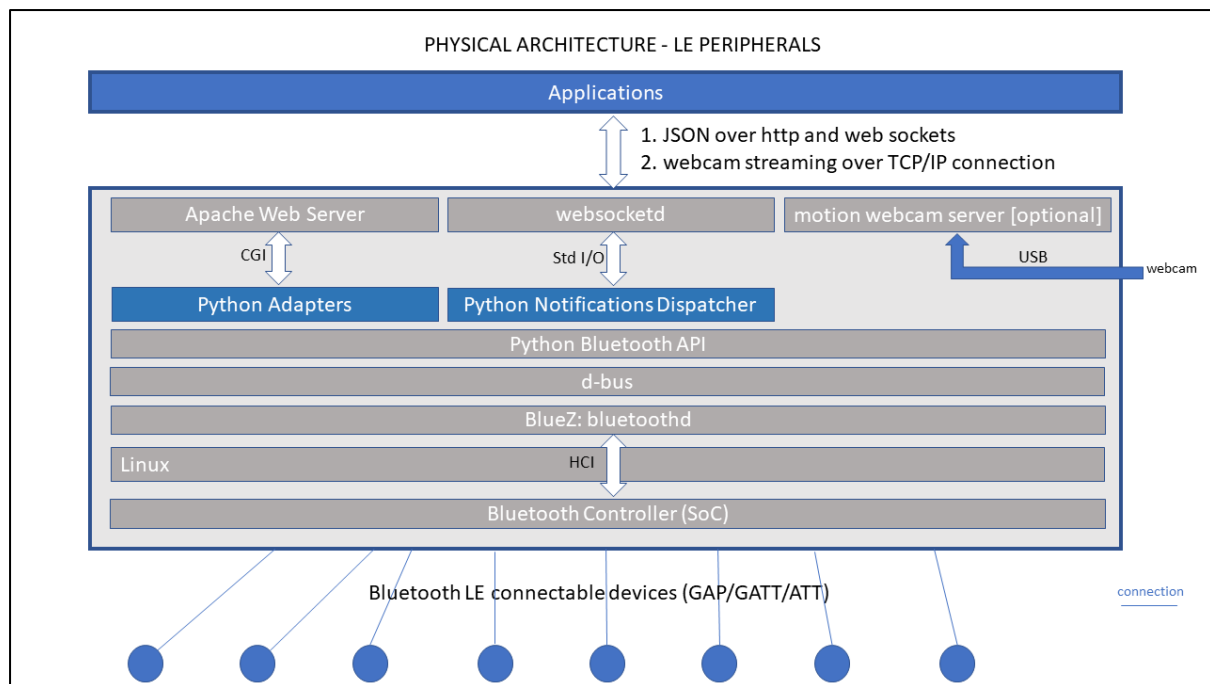
Figure 2 - bluetoothd and the serialisation of HCI commands and events

Consequently, a Bluetooth internet gateway that is based on Linux may only service Bluetooth LE GAP/GATT devices or Bluetooth mesh devices but not both at the same time.

The API provided by BlueZ for LE GAP/GATT devices is very different to the one provided for use with Bluetooth mesh devices too and so adapter code for one is very different to code for the other.

#### 4.3 Selected Physical Architecture

The physical architecture for a Bluetooth internet gateway which supports LE Peripheral devices is shown in Figure 3. You can see that it's only at a level of detail and concerning which BlueZ service is required that this architecture differs from the generic architecture of Figure 1.



**Figure 3 - Physical Architecture of a Bluetooth internet gateway supporting LE Peripheral devices**

Study Figure 3 and the explanatory text for each component below. Note that blue rectangles are components we will develop from scratch. Others are ready-made components that we shall install, configure and integrate. By selecting largely off the shelf components, we've saved ourselves a lot of work.

#### Applications

Applications fall outside of scope of our gateway architecture. They can be written in any language, provided the gateway API can be used and the gateway's TCP/IP application protocols are supported.

#### Gateway API and Supporting Protocols

The gateway shall support HTTP and web sockets for API communication.

The API shall use the HTTP GET for making requests which do not change the state of remote objects (e.g. characteristic values or device state), with an HTTP query string used to pass associated parameters.

The API shall use the HTTP PUT for making requests which do change the state of remote objects (e.g. characteristic values or device state), with JSON objects in the HTTP request body used to encode and transport associated parameters.

Responses will be delivered as JSON objects in all cases.

In the case of Bluetooth notifications and indications, web sockets shall be used since this use case requires bi-directional and server-initiated communication. Enabling and disabling notifications and indications shall be achieved by writing an API JSON control object to the web socket and characteristic value notifications and indications will be delivered to the application using the same web socket connection, also expressed as a JSON object.

Streaming USB webcam video data over a TCP/IP connection may be supported.

## Application Protocol Handlers

This layer of the architecture consists of two mandatory components and one optional component. Given HTTP and web sockets have been chosen as the TCP/IP interfaces to our gateway, we need one handler for each.

The [Apache](#) web server is an open source, free of charge, mature, reliable and easy to use HTTP server daemon. It has an extensive collection of plugins called modules, with which its functionality can be extended, and custom code to be executed in response to HTTP requests can be written in a multitude of languages. For these reasons, it has been selected as the component that will act as an HTTP handler for our gateway.

Choosing the Apache HTTP server has implications for other aspects of the physical architecture, including the platform the gateway will run on. Apache runs on many platforms, another of its advantages. But if you had envisaged implementing your gateway on a constrained device like a microcontroller, it will not be suitable. It's perfect for this education resource, however.

Web sockets require a server to service connections and so we'll be using a web socket server implementation called [websocketd](#). It's simple, lightweight and perfect for our needs.

We'll support plugging a USB webcam into our gateway so we can point it at a device or two, just because we can, which is often a good enough reason to do something! To that end, our architecture will include the open source webcam server software, [motion](#).

## Adapters

Adapters shall be written in the popular programming language [Python](#). The Apache web server will invoke a particular python script, according to the URL in HTTP requests and by writing to standard output, the python scripts will be able to return a JSON object back over the connection to the requesting HTTP client.

There are many ways to integrate custom code with web servers. The simplest, supported by most web servers, is called the Common Gateway Interface (CGI). It's not designed for high performance or efficiency. There are alternative approaches which fare better in those respects. But neither of these issues is a priority for us. The expectation is that a small number of concurrent users will make a small number of requests at a time.

## Bluetooth LE API

A Python Bluetooth LE API, which the adapter scripts can use has been written specifically for this Bluetooth SIG developer study guide. During the development stage of our project, you'll install it and get familiar with it.

## Bluetooth LE Stack

We'll be using Linux and therefore will use its standard BlueZ stack which is included in Linux distributions and specifically, per section 4.2, we'll run the *bluetoothd* daemon. The hardware platform must have a Bluetooth LE adapter for BlueZ to work.

## Operating System

Our gateway will run on the Linux operating system.

## Hardware Platform

You can run the gateway on any computer which supports Linux and has a Bluetooth adapter. We used a Raspberry Pi 4 model B and we'll be referring to this environment in the development stage.



## 4.4 Gateway Development

By the end of this section, you should have a working gateway which meets the requirements which were selected in module 02 other than those relating to Bluetooth mesh.

*The initial implementation will not be secure and should not be connected to the internet or other insecure networks!*

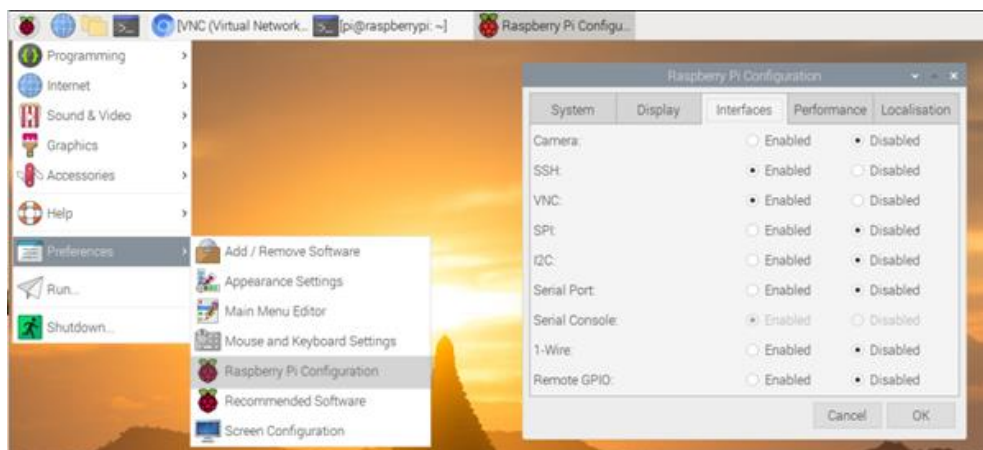
### 4.4.1 Platform Set-up

We need to establish a platform upon which to build the gateway. We'll define this as a computer which has the required Bluetooth capabilities and supports our selected programming language, Python.

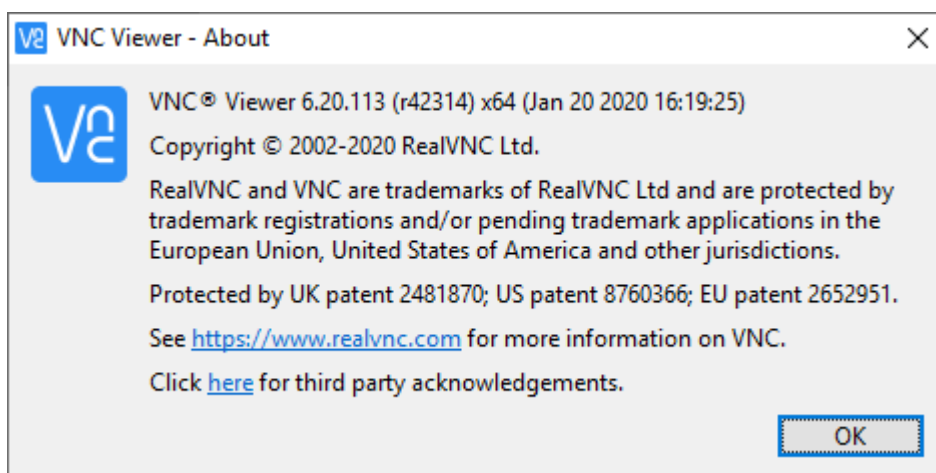
#### 4.4.1.1 Task - Raspberry Pi set-up

Set up your Raspberry Pi by following the instructions at <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up>.

Enable access to your Raspberry Pi via SSH and VNC by going into the desktop menu and navigating to Preferences/Raspberry Pi Configuration.



You may now work directly on your Raspberry Pi, with a USB keyboard and mouse connected to it and a monitor connected via HDMI. Alternatively, with a suitable VNC Viewer application on your usual PC, you can use remote access to work with the Raspberry Pi.



Your final option is to use an SSH terminal such as Putty from <https://putty.org/>.

Determine the IP address of your Raspberry Pi by running either *ifconfig* or the alternative command, *ip a*. Make a note of the IP address..

```
pi@raspberrypi:~$ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether dc:a6:32:0e:83:e0 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 25 bytes 1484 (1.4 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 25 bytes 1484 (1.4 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.39 netmask 255.255.255.0 broadcast 192.168.0.255
    inet6 fe80::4666:142d:850b:c0f6 prefixlen 64 scopeid 0x20<link>
    ether dc:a6:32:0e:83:e1 txqueuelen 1000 (Ethernet)
    RX packets 10909 bytes 7774618 (7.4 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 10963 bytes 8064070 (7.6 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

pi@raspberrypi:~$
```

Windows users note: One of the commands for acquiring your IP address on a Linux machine is *ifconfig*. On Windows however, the command is *ipconfig*. Make sure you issue the command for Linux!

#### 4.4.1.2 Task - Bluetooth check

Check the version of BlueZ running on your machine by launching the *bluetooth* tool and entering the *version* command.

```
pi@raspberrypi:~$ bluetoothctl
Agent registered
[bluetooth]# version
Version 5.50
[bluetooth]#
```

The version reported should be at least version 5.50. Exit *bluetoothctl* by entering the *quit* command.

#### 4.4.1.3 Task - Python check

Check that Python 3 is installed and that it is functioning, as follows:

```
pi@raspberrypi:~$ python3 --version

Python 3.7.3
pi@raspberrypi:~$ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World

>>> quit()
```

## 4.4.2 Apache Web Server

### 4.4.2.1 Task - Install Apache2

Install the Apache2 web server as follows:

```
sudo apt install apache2
```

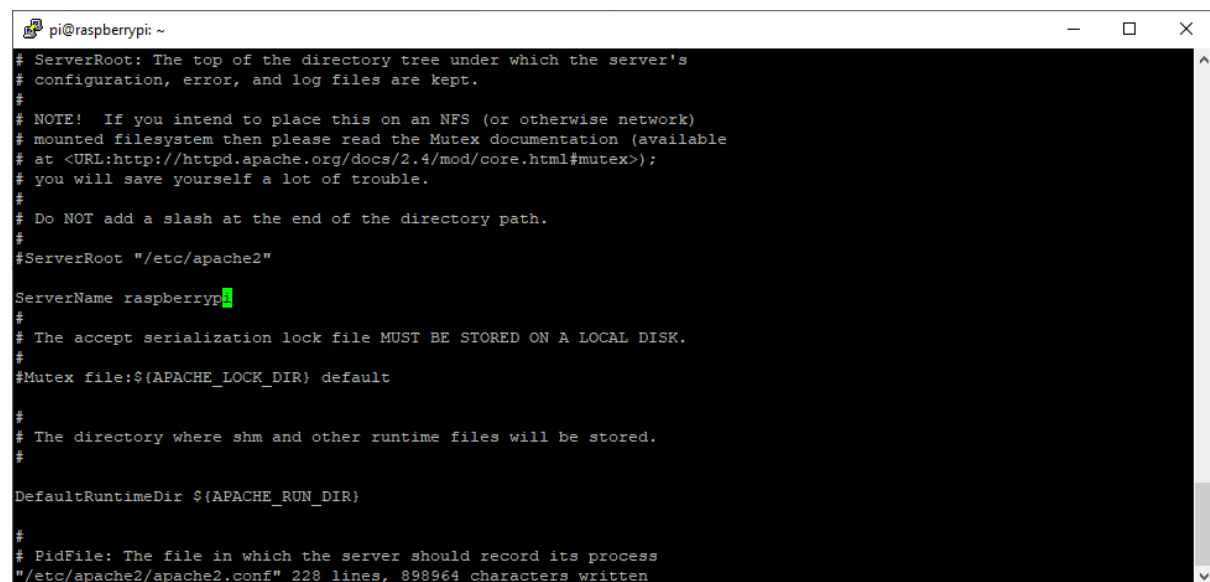
Check that Apache is running using *service* like this:

```
pi@raspberrypi:~ $ sudo service apache2 status
● Apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/Apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2020-04-27 13:42:48 BST; 3min 10s ago
     Docs: https://httpd.apache.org/docs/2.4/
   Main PID: 2950 (Apache2)
    Tasks: 55 (limit: 4915)
   Memory: 4.0M
   CGroup: /system.slice/Apache2.service
           └─2950 /usr/sbin/Apache2 -k start
             └─2951 /usr/sbin/Apache2 -k start
               └─2952 /usr/sbin/Apache2 -k start

Apr 27 13:42:48 raspberrypi systemd[1]: Starting The Apache HTTP Server...
Apr 27 13:42:48 raspberrypi Apachectl[2939]: AH00558: Apache2: Could not reliably determine
the server's fully qualified domain name, using 127.0.1.1. Set the 'ServerName' directive
globally to suppress this
Apr 27 13:42:48 raspberrypi systemd[1]: Started The Apache HTTP Server.
```

Note the highlighted message regarding ServerName. Address this by editing the `/etc/Apache2/Apache2.conf` configuration file, adding a `ServerName` directive with a value of *raspberrypi* and restarting Apache.

```
sudo vi /etc/apache2/apache2.conf
```



```
pi@raspberrypi: ~
# ServerRoot: The top of the directory tree under which the server's
# configuration, error, and log files are kept.
#
# NOTE! If you intend to place this on an NFS (or otherwise network)
# mounted filesystem then please read the Mutex documentation (available
# at <URL:http://httpd.apache.org/docs/2.4/mod/core.html#mutex>);
# you will save yourself a lot of trouble.
#
# Do NOT add a slash at the end of the directory path.
#
#ServerRoot "/etc/apache2"

ServerName raspberrypi
#
# The accept serialization lock file MUST BE STORED ON A LOCAL DISK.
#
#Mutex file:${APACHE_LOCK_DIR} default
#
# The directory where shm and other runtime files will be stored.
#

DefaultRuntimeDir ${APACHE_RUN_DIR}
#
# PidFile: The file in which the server should record its process
"/etc/apache2/apache2.conf" 228 lines, 898964 characters written
```

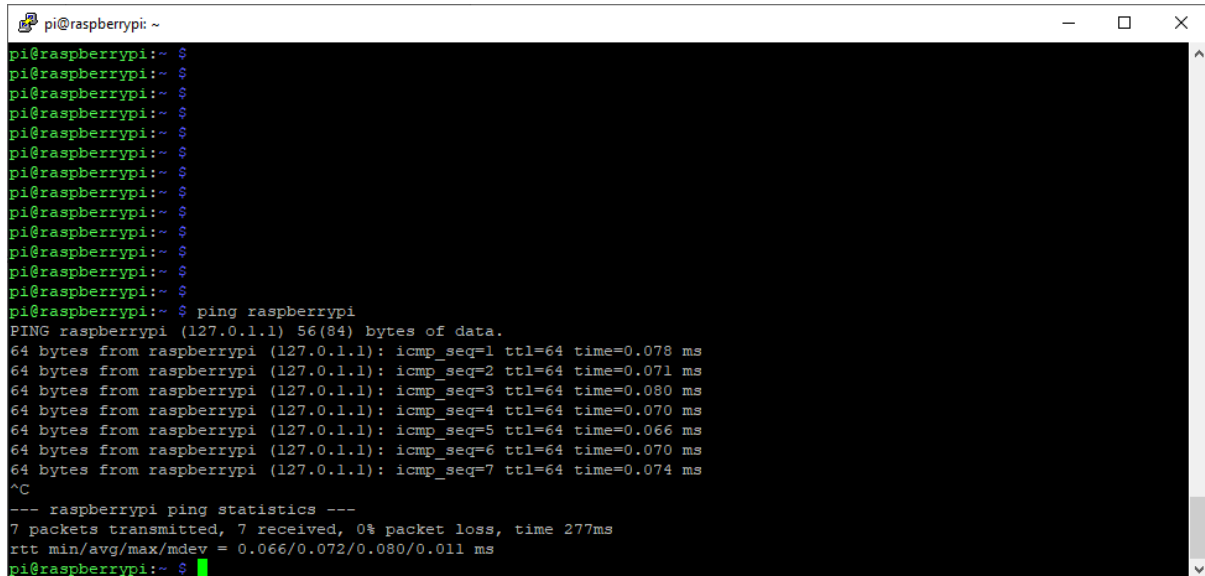
Restart the Apache2 service and check its status:

```
pi@raspberrypi:~ $ sudo service apache2 restart
pi@raspberrypi:~ $ sudo service apache2 status
● Apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/Apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2020-04-27 13:52:17 BST; 6s ago
     Docs: https://httpd.apache.org/docs/2.4/
   Process: 3527 Exec Start=/usr/sbin/Apachectl start (code=exited, status=0/SUCCESS)
```

```
Main PID: 3531 (Apache2)
Tasks: 55 (limit: 4915)
Memory: 3.8M
CGroup: /system.slice/Apache2.service
├─3531 /usr/sbin/Apache2 -k start
├─3532 /usr/sbin/Apache2 -k start
└─3533 /usr/sbin/Apache2 -k start

Apr 27 13:52:17 raspberrypi systemd[1]: Starting The Apache HTTP Server...
Apr 27 13:52:17 raspberrypi systemd[1]: Started The Apache HTTP Server.
```

You should also be able to ping the host name *raspberrypi*.

A terminal window titled 'pi@raspberrypi: ~' with standard window controls. It shows a series of empty prompts followed by a 'ping raspberrypi' command. The output displays 'PING raspberrypi (127.0.1.1) 56(84) bytes of data.' followed by seven lines of ping results, each showing '64 bytes from raspberrypi (127.0.1.1): icmp\_seq=X ttl=64 time=0.XXX ms' for X from 1 to 7. Below this, it shows '--- raspberrypi ping statistics ---' and summary statistics: '7 packets transmitted, 7 received, 0% packet loss, time 277ms' and 'rtt min/avg/max/mdev = 0.066/0.072/0.080/0.011 ms'. The prompt returns to 'pi@raspberrypi:~ \$'.

Apache is a modular system and by default, various modules are enabled. Find out which are currently enabled using the *apache2ctl* command as shown.

```
pi@raspberrypi:~ $ apache2ctl -t -D DUMP_MODULES
Loaded Modules:
 core_module (static)
 so_module (static)
 watchdog_module (static)
 http_module (static)
 log_config_module (static)
 logio_module (static)
 version_module (static)
 unixd_module (static)
 access_compat_module (shared)
 alias_module (shared)
 auth_basic_module (shared)
 authn_core_module (shared)
 authn_file_module (shared)
 authz_core_module (shared)
 authz_host_module (shared)
 authz_user_module (shared)
 autoindex_module (shared)
 deflate_module (shared)
 dir_module (shared)
 env_module (shared)
 filter_module (shared)
 mime_module (shared)
 mpm_event_module (shared)
 negotiation_module (shared)
 reqtimeout_module (shared)
 setenvif_module (shared)
 status_module (shared)
```

Our plan is to use the Common Gateway Interface (CGI) for executing Python scripts in response to certain HTTP requests. For this to work, we need the Apache *cgid* module enabled. As you can see

from the list of modules reported by *apache2ctl*, by default it is not. Enable it using *a2enmod*, list the enabled modules again to check it has been enabled, and then restart Apache.

```
pi@raspberrypi:~ $ sudo a2enmod cgid
Enabling module cgid.
To activate the new configuration, you need to run:
    systemctl restart Apache2

pi@raspberrypi:~ $ Apache2ctl -t -D DUMP_MODULES
Loaded Modules:
  core_module (static)
  so_module (static)
  watchdog_module (static)
  http_module (static)
  log_config_module (static)
  logio_module (static)
  version_module (static)
  unixd_module (static)
  access_compat_module (shared)
  alias_module (shared)
  auth_basic_module (shared)
  authn_core_module (shared)
  authn_file_module (shared)
  authz_core_module (shared)
  authz_host_module (shared)
  authz_user_module (shared)
  autoindex_module (shared)
  cgid_module (shared)
  deflate_module (shared)
  dir_module (shared)
  env_module (shared)
  filter_module (shared)
  mime_module (shared)
  mpm_event_module (shared)
  negotiation_module (shared)
  reqtimeout_module (shared)
  setenvif_module (shared)
  status_module (shared)

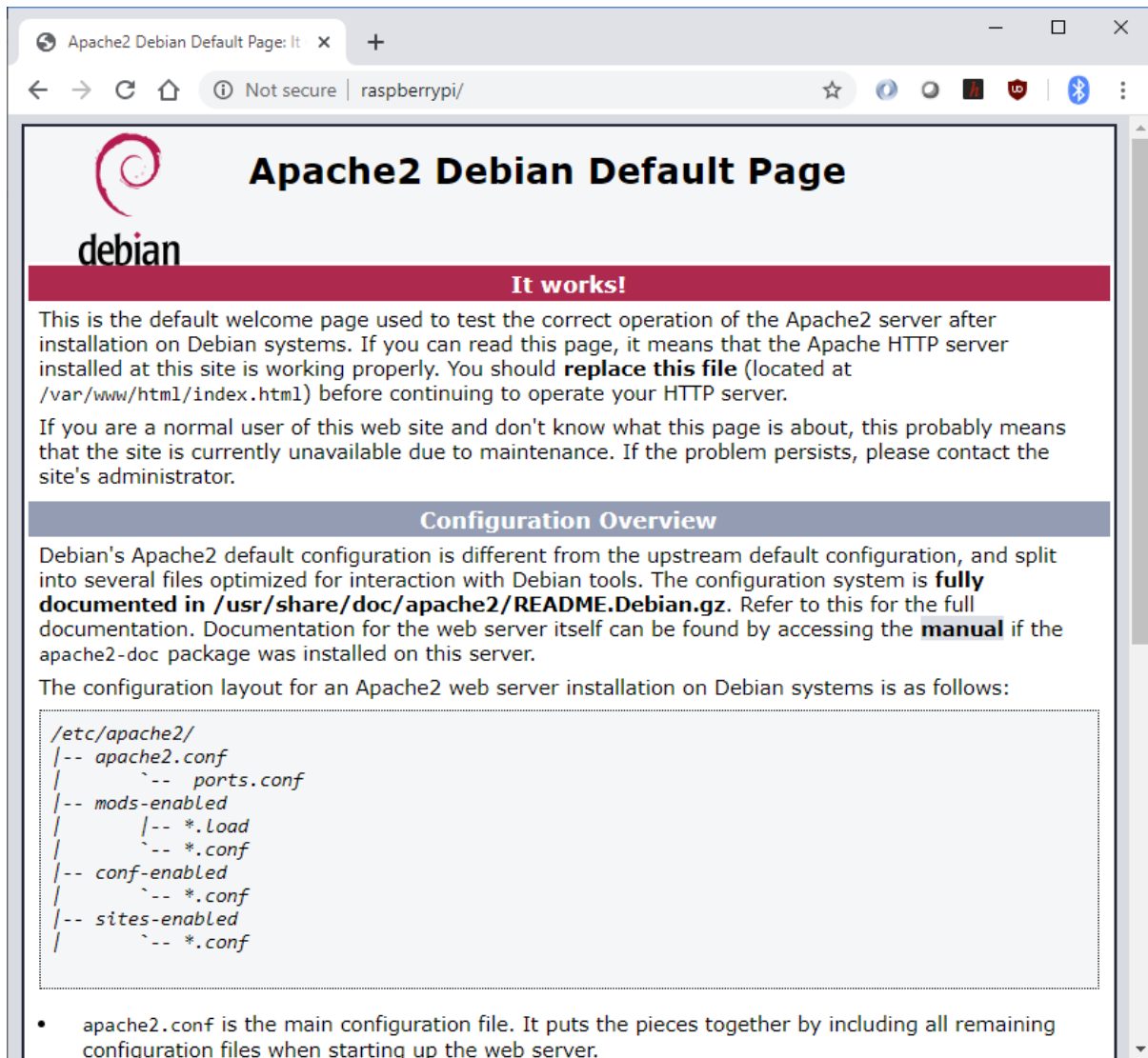
pi@raspberrypi:~ $ sudo service apache2 restart
```

If you are intending to test from a machine other than the gateway itself (and this is recommended), add an entry to the local *hosts* file on the other computer, and associate name *raspberrypi* with the IP address you acquired from your Raspberry Pi earlier. On Windows, this involves starting an editor as Administrator and editing the file `c:\windows\system32\drivers\etc\hosts` and adding an entry such as:

```
192.168.0.39    raspberrypi
```

You should now be able to ping *raspberrypi* from this computer, provided it is connected to the same LAN as the Raspberry Pi.

Check that you can reach the default home page of your Apache web server by going to <http://raspberrypi> in your web browser. You should see:



#### 4.4.2.2 Task - Verify that CGI and Python script execution is working

Using the editor of your choice, create a script file called `hello.py` located at `/usr/lib/cgi-bin`. It should contain the following, simple Python script:

```
#!/usr/bin/python3
print ("Content-type: text/html\n\n")
print ("Hello World")
```

**Note:** for this and many other steps in the exercises, you must ensure you have the required permissions to perform the required action. The easiest way to do this is to prefix any commands with `sudo`. This will temporarily give your command the superuser (root) privileges.

Change the file's owner and group to `www-data` (the user which Apache runs as) and permissions to `775` so that both the `www-data` user and group members have read, write and execute access.

```
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/hello.py
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/hello.py
pi@raspberrypi:~ $ ls -l /usr/lib/cgi-bin/
total 4
```

```
-rwxrwxr-x 1 www-data 77 Apr 27 14:03 hello.py
```

It will be assumed from now on that you will give all CGI scripts created in this directory, this ownership and set of permissions.

Edit `/etc/group` with `sudo vi /etc/group` and add the `pi` user to the `www-data` group.



```
pi@raspberrypi: ~  
proxy:x:13:  
kmem:x:15:  
dialout:x:20:pi  
fax:x:21:  
voice:x:22:  
cdrom:x:24:pi  
floppy:x:25:  
tape:x:26:  
sudo:x:27:pi  
audio:x:29:pi  
dip:x:30:  
www-data:x:33:pi  
backup:x:34:  
operator:x:37:  
list:x:38:  
irc:x:39:  
src:x:40:  
gnats:x:41:  
shadow:x:42:  
utmp:x:43:  
video:x:44:pi  
sasl:x:45:  
plugdev:x:46:pi  
staff:x:50:  
games:x:60:pi  
users:x:100:pi  
nogroup:x:65534:  
systemd-journal:x:101:  
systemd-timesync:x:102:  
systemd-network:x:103:  
systemd-resolve:x:104:  
input:x:105:pi
```

Now your default Linux user will have write and execute access to CGI scripts, which will make working with these scripts easier.

Go to `http://raspberrypi/cgi-bin/hello.py` in your browser. You should see a web page containing only “Hello World.”.

If you did, you have verified that your Apache web server is both working and correctly configured to be able to execute Python scripts. If not, carefully check the previous steps to ensure you have not missed anything. The last few entries of the Apache error log, at `/var/log/apache2/error.log` may provide some insight.

#### 4.4.3 websocketd

Our architecture includes the use of web sockets for the enabling, disabling and delivery of Bluetooth notifications and indications and a daemon which will act as a dispatcher for web socket traffic called `websocketd`.

##### 4.4.3.1 Task - Install `websocketd`

Go to the `websocketd` web site at <http://websocketd.com/>. Follow the instructions there to download and install `websocketd` and to test it with a simple Python script.

**Tip:** You need the Linux ARM version for a Raspberry Pi

```
# expand the package archive  
  
pi@raspberrypi:~/Downloads $ unzip websocketd-0.3.0-linux_arm.zip  
Archive:  websocketd-0.3.0-linux_arm.zip  
  inflating: websocketd  
  inflating: README.md  
  inflating: LICENSE  
  inflating: CHANGES
```

```
# which directories are already on the PATH?

pi@raspberrypi:~/Downloads $ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games

# copy binary to a suitable directory

pi@raspberrypi:~/Downloads $ sudo cp websocketd /usr/local/bin

# check websocketd is available

pi@raspberrypi:~/Downloads $ which websocketd
/usr/local/bin/websocketd
```

#### 4.4.3.2 Task - Verify websocketd

Next we'll check that websocketd is working by setting up a simple web page which will receive a series of numbers over a web socket connection, generated by a Python script running on the Raspberry Pi. Follow the steps below to create a web page and the required Python script.

```
# create directory and web page

pi@raspberrypi:~ $ sudo mkdir /var/www/html/wstest
pi@raspberrypi:~ $ sudo vi /var/www/html/wstest/index.html

# index.html must contain the following:

<html>
<head>
  <script src="js/wstest.js"></script>
</head>
<body>
  <h2>Web Socket Test</h2>
  <table><tr><td><button id="btn_test" class="button" onclick="onTest();"
/>TEST</button></td></tr></table>
  <div id="message" class="important"></div>
  <pre id="content"></pre>

</body>
</html>

# create a directory for JavaScript files and the JavaScript needed by our test page

pi@raspberrypi:~ $ sudo mkdir /var/www/html/wstest/js
pi@raspberrypi:~ $ sudo vi /var/www/html/wstest/js/wstest.js

# Content of wstest.js:

function onTest() {

    var ws = new WebSocket('ws://raspberrypi:8080/');

    ws.onmessage = function(event) {
        document.getElementById('content').innerHTML = 'Notification: ' + event.data;
    };

}

# set ownership and permissions
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /var/www/html/wstest/
pi@raspberrypi:~ $ sudo chmod -R 775 /var/www/html/wstest/

# create a Python script which will write to websockets
pi@raspberrypi:~ $ sudo vi /usr/lib/cgi-bin/wstest.py

# /usr/lib/cgi-bin/wstest.py content:

#!/usr/bin/python3
from sys import stdout
```



```

from time import sleep

# Count from 1 to 10 with a sleep
for count in range(0, 10):
    print(count + 1)
    stdout.flush()
    sleep(0.5)

# set ownership and permissions
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/wstest.py
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/wstest.py

```

Now start websocketd, listening on port 8080 like this:

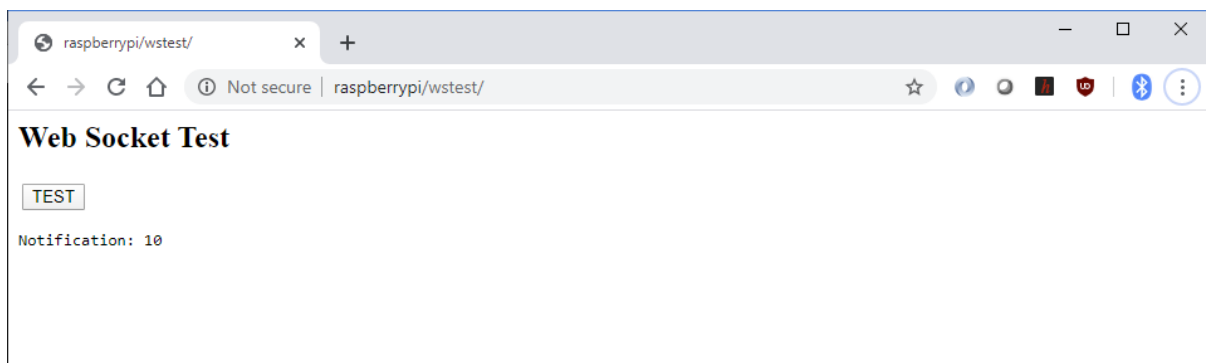
```

pi@raspberrypi:~ $ websocketd --port=8080 /usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:25:55 +0100 | INFO    | server      | | Serving using application   :
/usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:25:55 +0100 | INFO    | server      | | Starting WebSocket server  :
ws://raspberrypi:8080/

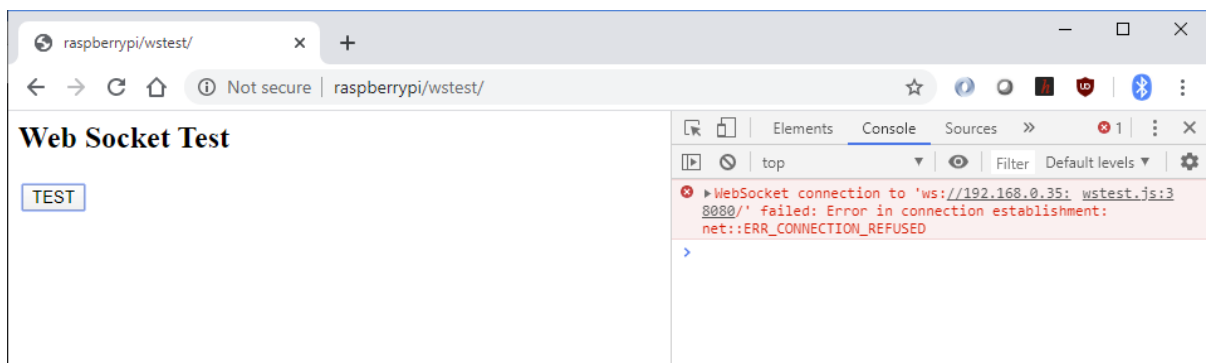
```

In a browser, access the test page here: <http://raspberrypi/wstest/>

Click the start button and you should see a series of integer values, from 1 to 10, appearing on the web page. These values were generated by the Python script, which was launched by websocketd and written to the web socket connection.



If it didn't work for you, open the developer console in your browser. Using Google Chrome on Windows, CTRL+SHIFT+I will achieve this result quickly. The console should offer clues, the most likely issues being that either websocketd is not running, it's running but listening on a different port to that which is used in the JavaScript or the JavaScript specifies the wrong IP address / host name.



After successfully executing, the Python script will exit and the websocketd console will indicate that the web socket has been disconnected.

```

pi@raspberrypi:~ $ websocketd --port=8080 /usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:28:40 +0100 | INFO | server | | Serving using application :
/usr/lib/cgi-bin/wstest.py
Tue, 28 Apr 2020 08:28:40 +0100 | INFO | server | | Starting WebSocket server :
ws://raspberrypi:8080/
Tue, 28 Apr 2020 08:28:42 +0100 | ACCESS | session | url:'http://192.168.0.35:8080/'
id:'1588058922060392454' remote:'192.168.0.20' command:'/usr/lib/cgi-bin/wstest.py'
origin:'http://raspberrypi' | CONNECT
Tue, 28 Apr 2020 08:28:47 +0100 | ACCESS | session | url:'http://192.168.0.35:8080/'
id:'1588058922060392454' remote:'192.168.0.20' command:'/usr/lib/cgi-bin/wstest.py'
origin:'http://raspberrypi' pid:'2531' | DISCONNECT

```

## 4.4.4 Bluetooth API

A simple Bluetooth API, to be used from your Python scripts has been provided with this study guide.

### 4.4.4.1 Task - Bluetooth API installation

Create the directory `/usr/lib/cgi-bin/bluetooth`

Install the Bluetooth API by copying the Python scripts in the study guide package directory `implementation\solutions\peripherals\python_bluetooth_api` to the new directory on your Raspberry Pi.

Set the ownership of the files and directory to `www-data:www-data` and permissions to `775`.

List the contents of the `/usr/lib/cgi-bin/bluetooth` directory. It should look like this:

```

-rw-r--r-- 1 pi pi 724 Apr 27 15:03 bluetooth_constants.py
-rw-r--r-- 1 pi pi 200 Apr 27 15:03 bluetooth_exceptions.py
-rw-r--r-- 1 pi pi 6043 Apr 27 15:03 bluetooth_gap.py
-rw-r--r-- 1 pi pi 14350 Apr 27 15:03 bluetooth_gatt.py
-rw-r--r-- 1 pi pi 1549 Apr 27 15:03 bluetooth_general.py
-rw-r--r-- 1 pi pi 1230 Apr 27 15:03 bluetooth_utils.py

```

Each of the files is a Python *module* and contains a series of functions and/or variables. The names should give you an idea of the purpose of each module. Feel free to examine them but take care not to change anything. We shall not be learning about how the API itself works, since the scope of this resource is gateway development, and when it comes to building a real gateway, you are most likely to be using a 3<sup>rd</sup> party API rather than developing one yourself.

You'll gain familiarity with this Bluetooth API as you develop adapter scripts.

### 4.4.4.2 Task - Bluetooth DBus Communication

The Bluetooth API uses the Linux DBus inter-process communication system. A default security policy restricts use of DBus for Bluetooth purposes to processes owned by users that are a member of the *bluetooth* system group. You can see this in the `/etc/dbus-1/system.d/bluetooth.conf` file, here:

```

<!-- allow users of bluetooth group to communicate -->
<policy group="bluetooth">
  <allow send_destination="org.bluez"/>
</policy>

```

Edit `/etc/group` and add the *www-data* and *pi* users to the bluetooth group.

```

pi@raspberrypi:~ $ sudo vi /etc/group
pi@raspberrypi:~ $ cat /etc/group|grep bluet
bluetooth:x:112:www-data,pi
# you need to start a new bash shell for the change to be activated - do this by entering
# su - pi

```

```
# or exit this shell and start a new one

pi@raspberrypi:~ $ groups
pi adm dialout cdrom sudo audio www-data video plugdev games users input netdev bluetooth
gpio i2c spi
```

Now restart the DBus daemon:

```
pi@raspberrypi:~ $ sudo service dbus restart
```

## 4.4.4 Adapter Code Development

### 4.4.4.0 Introduction

#### 4.4.4.0.2 Approach

You will now proceed to develop the gateway adapter scripts which will use the Bluetooth API and service requests received over HTTP from the Apache web server. This is the primary code development task of this study guide.

We'll proceed one use case at a time and will generally be concerned with validating the request, extracting and validating parameters, executing one or more calls to our Python Bluetooth API, and transforming the results into a JSON object to be sent back to the HTTP client which submitted the original request. Each task will start with an illustration of the API, in terms of the HTTP request and its parameters and the JSON object(s) to be returned in the response.

For some of you, this may be the first time you have programmed in Python, so we'll take it very slowly to begin with. You should quickly appreciate that each script is going to be relatively short and with the exception of notifications and indications handling, they are all quite similar. So, after the first task, which will advance in baby steps, we'll go faster.

There are numerous Python tutorials on the internet, including this one:

<https://docs.python.org/3/tutorial/index.html>

**Note: we're using Python 3 and not Python version 2**

The adapter code is available in the implementation/solutions/peripherals folder of the study guide package as two distinct versions. At this stage, the solution version which corresponds to the work you will undertake in this section, is in

implementation/solutions/peripherals/gateway\_adapters\_no\_security directory.

#### 4.4.4.0.3 Testing

You will need to test your code and to do so, you will need one or more suitable Bluetooth LE peripheral devices. A suitable device is one which is connectable and contains GATT characteristics supporting read, write and notify operations. You can use any device you wish to, but the study guide assumes you are using a BBC micro:bit and includes both some source code and two versions of a binary hex file which you can install over USB by dragging and dropping onto the USB mass storage device relating to the micro:bit. The file named `microbit-example_device.hex` does not require pairing, whereas the `microbit-example_device_pairing_required.hex` binary will require you to pair your micro:bit with your Raspberry Pi. For now, use the version which does not require pairing. We'll cover pairing when we cover security.

#### 4.4.4.0.1 Preparation

Create a directory for the gateway adapter code and set its ownership and permissions so that it can be used by both the Apache web server and you, logged into the default *pi* account.

```
pi@raspberrypi:~ $ sudo mkdir /usr/lib/cgi-bin/gateway
pi@raspberrypi:~ $ sudo chown www-data:www-data /usr/lib/cgi-bin/gateway/
pi@raspberrypi:~ $ sudo chmod 775 /usr/lib/cgi-bin/gateway/
```

#### 4.4.4.1 Task - Implement the Device Discovery adapter script

##### 4.4.4.1.1 API Details

Request
do_discover_devices.py
HTTP Method
GET
Query String Arguments
scantime=3000 where scantime is the duration over which Bluetooth scanning will be performed, in milliseconds.
Response Description
Returns a JSON format array of objects, each representing a single, discovered Bluetooth LE device. Each device object contains attributes which are available for it, including but not limited to the Bluetooth device address ( <i>bdaddr</i> ), signal strength ( <i>RSSI</i> ), whether or not it is paired with the gateway computer ( <i>paired</i> ) and its connected status ( <i>connected</i> ).
Response Example - Successful Execution
<pre>[{   "ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb",   "paired": false,   "bdaddr": "E9:CD:D7:25:4D:19",   "connected": false,   "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"],   "RSSI": -85,   "services_resolved": false,   "ad_service_data": "00C3FE7223C0193364FAA7B12F74D81A14914440" }, {   "paired": false,   "ad_manufacturer_data_cid": 76,   "ad_manufacturer_data": "0215B9407F30F5F8466EAF925556B57FE6DAEC8EBD4B8",   "bdaddr": "EB:CF:D9:27:4F:1B",   "connected": false,   "UUIDs": [],   "RSSI": -75,   "services_resolved": false }, {   "ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb",   "paired": false,   "bdaddr": "E6:D4:CD:29:E9:58",   "connected": false,   "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"],   "RSSI": -75,   "services_resolved": false,   "ad_service_data": "00A4DB33F3353D1FB3F336847E2AB38913914440" }, {   "paired": false,   "ad_manufacturer_data_cid": 117,   "ad_manufacturer_data": "4204012067190D0002013200000000000000000000000000",   "bdaddr": "8C:79:F5:1D:60:02",   "connected": false,   "UUIDs": [],   "RSSI": -93,   "services_resolved": false }]</pre>
Response Example - Error - Invalid Args
<pre>{"result": 8}</pre>

##### 4.4.4.1.2 Coding

Create a new file called do\_discover\_devices.py in the /usr/lib/cgi-bin/gateway directory.

```
vi /usr/lib/cgi-bin/gateway/do_discover_devices.py
```

Start by adding the following lines to the top of the file:

```
#!/usr/bin/python3
import os
import json
import sys
import cgi
sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants
```

The first line is the *shebang* and tells the parent interpreter which interpreter to use to execute the remainder of the script. In our case, it's the Python 3 interpreter at `/usr/bin/python3`.

Like many programming languages, Python allows code to be grouped in modules. The *import* statements load modules whose functions are to be used. Some modules are built into the language and others are external. These modules are searched for along a search path (you can check how this works in the Python documentation). Our Python Bluetooth API is implemented as a series of modules, all of which are in the `/usr/lib/cgi-bin/bluetooth/` directory. We use a function from the Python `sys` module to dynamically insert this directory into the search path and then import the `bluetooth_gap` module, which contains some functions we are going to need.

This script may only be invoked using an HTTP GET so we shall validate this and return an HTTP status 405 (Method Not Allowed) response if this check fails.

Update `do_discover_devices.py` so that it looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
import cgi
sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
else:
    print("ERROR: Not called by HTTP")
```

The `os` module allows access to environment variables and the HTTP method used to invoke the script may be found in a variable called `REQUEST_METHOD`. If this variable is not found, it is assumed that the script was invoked from the command line and an error printed to the console.

Otherwise, we check the HTTP method and if anything other than a GET was used, write HTTP headers to standard out, indicating a status 405. If GET was used, we execute code which will handle the request and at this stage, have merely set up the HTTP Content-Type header, which indicates that the content will be a JSON object.

Note that the print statement outputs the argument which follows to standard out. When executed as a CGI call, this results in the output being sent back to the calling http client, so this is the mechanism use to send responses back to the client.

Arguments are passed as a query string which is available to the script via the cgi module's FieldStorage API. This adapter script expects a parameter called scantime which is the time in milliseconds for which Bluetooth scanning should take place, so our next task is to check that this has been provided and if not, return a JSON object indicating that there was a validation error.

Update do\_discover\_devices.py so that it looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
import cgi
sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "scantime" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
else:
    print("ERROR: Not called by HTTP")
```

Finally, update the script so that it makes a call to Bluetooth API function bluetooth\_gap.discover\_devices, which will return its result in a Python format and convert this to JSON. Output the JSON object using *print* to finalise the response.

```
#!/usr/bin/python3
import os
import json
import sys
import cgi

sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants
import bluetooth_utils

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "scantime" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            scantime = querystring.getfirst("scantime", "2000")
            devices_discovered = bluetooth_gap.discover_devices(int(scantime))
```

```
        devices_discovered_json = json.JSONEncoder().encode(devices_discovered)
        print(devices_discovered_json)
    else:
        print("ERROR: Not called by HTTP")
```

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

#### 4.4.4.1.3 Testing

You can test this script from the command line of your Raspberry Pi, using *curl*. Run the following command and observe the result:

##### command

```
pi@raspberrypi:~ $ curl --request GET http://localhost/cgi-bin/gateway/do_discover_devices.py?scantime=3000
```

##### result

```
[{"ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb", "paired": false,
"bdaddr": "E6:D4:CD:29:E9:58", "connected": false, "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"], "RSSI": -67, "services_resolved": false, "ad_service_data": "00A4DB33F3353D1FB3F336847E2AB38913914440"}, {"ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb", "paired": false, "bdaddr": "E9:CD:D7:25:4D:19", "connected": false, "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"], "RSSI": -80, "services_resolved": false, "ad_service_data": "00C3FE7223C0193364FAA7B12F74D81A14914440"}, {"name": "telly", "paired": false, "ad_manufacturer_data_cid": 196, "ad_manufacturer_data": "073302131580", "bdaddr": "66:D5:68:46:41:72", "connected": false, "UUIDs": ["0000feb9-0000-1000-8000-00805f9b34fb"], "RSSI": -90, "services_resolved": false}, {"paired": false, "ad_manufacturer_data_cid": 76, "ad_manufacturer_data": "0215B9407F30F5F8466EAFF925556B57FE6DAEC8EBD4B8", "bdaddr": "EB:CF:D9:27:4F:1B", "connected": false, "UUIDs": [], "RSSI": -73, "services_resolved": false}, {"paired": false, "ad_manufacturer_data_cid": 117, "ad_manufacturer_data": "4204012067190D000201320000000000000000000000000000", "bdaddr": "8C:79:F5:1D:60:02", "connected": false, "UUIDs": [], "RSSI": -84, "services_resolved": false}]
```

To make the result easier to read, go to <https://jsonlint.com/> and paste the JSON object, starting with “[” and ending with ”]” into the code window and click the Validate JSON button. It should contain no errors and format nicely, so it looks something like this:

```
[{
  "ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb",
  "paired": false,
  "bdaddr": "E6:D4:CD:29:E9:58",
  "connected": false,
  "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"],
  "RSSI": -67,
  "services_resolved": false,
  "ad_service_data": "00A4DB33F3353D1FB3F336847E2AB38913914440"
}, {
  "ad_service_data_uuid": "0000fe9a-0000-1000-8000-00805f9b34fb",
  "paired": false,
  "bdaddr": "E9:CD:D7:25:4D:19",
  "connected": false,
  "UUIDs": ["0000fe9a-0000-1000-8000-00805f9b34fb"],
  "RSSI": -80,
  "services_resolved": false,
  "ad_service_data": "00C3FE7223C0193364FAA7B12F74D81A14914440"
}, {
  "name": "telly",
  "paired": false,
  "ad_manufacturer_data_cid": 196,
  "ad_manufacturer_data": "073302131580",
  "bdaddr": "66:D5:68:46:41:72",
  "connected": false,
  "UUIDs": ["0000feb9-0000-1000-8000-00805f9b34fb"],
```



```

        "RSSI": -90,
        "services_resolved": false
    }, {
        "paired": false,
        "ad_manufacturer_data_cid": 76,
        "ad_manufacturer_data": "0215B9407F30F5F8466EAFF925556B57FE6DAEC8EBD4B8",
        "bdaddr": "EB:CF:D9:27:4F:1B",
        "connected": false,
        "UUIDs": [],
        "RSSI": -73,
        "services_resolved": false
    }, {
        "paired": false,
        "ad_manufacturer_data_cid": 117,
        "ad_manufacturer_data": "4204012067190D0002013200000000000000000000000000000000",
        "bdaddr": "8C:79:F5:1D:60:02",
        "connected": false,
        "UUIDs": [],
        "RSSI": -84,
        "services_resolved": false
    }
}

```

If your test produced no output, you probably have errors in your script, or maybe it is in the wrong directory or has the wrong permissions. Look at the end of `/var/log/apache2/error.log` for clues.

**Tip:** Python has very strict [rules about indentation](#).

Now execute the following tests to check that the validation of the HTTP method and the expected arguments is working correctly.

```

pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --request PUT
http://localhost/cgi-bin/gateway/do_discover_devices.py?scantime=3000
Status-Line: HTTP/1.0 405 Method Not Allowed

```

```

pi@raspberrypi:~ $ curl --request GET http://localhost/cgi-
bin/gateway/do_discover_devices.py?xxxxxxx=3000
{"result": 8}

```

Result codes are defined in the `bluetooth_constants.py` file in the `bluetooth` directory:

```

RESULT_OK = 0
RESULT_ERR = 1
RESULT_ERR_NOT_CONNECTED = 2
RESULT_ERR_NOT_SUPPORTED = 3
RESULT_ERR_SERVICES_NOT_RESOLVED = 4
RESULT_ERR_WRONG_STATE = 5
RESULT_ERR_ACCESS_DENIED = 6
RESULT_EXCEPTION = 7
RESULT_ERR_BAD_ARGS = 8
RESULT_ERR_NOT_FOUND = 9

```

#### 4.4.4.2 Task - Implement the Device Connect adapter script

##### 4.4.4.2.1 API Details

Request
do_connect.py
HTTP Method
PUT
Arguments Object
<pre>{"bdaddr" : "D9:64:36:0E:87:01"}</pre> <p>where bdaddr is the Bluetooth device address of the device you wish to connect to.</p>
Response Description
Returns a JSON object containing a result code.
Response Example - Successful Execution
<pre>{"result": 0}</pre>
Response Example - Error - Invalid Args
<pre>{"result": 8}</pre>
Response Example - Error - Not Found
<pre>{"result": 9}</pre>

##### 4.4.4.2.2 Coding

Create a new file called do\_connect.py in the /usr/lib/cgi-bin/gateway directory.

```
vi /usr/lib/cgi-bin/gateway/do_connect.py
```

Exactly the same code pattern applies to this adapter. We start with the required shebang, import the modules to be used, validate the HTTP method, which must be PUT in this case, and check for the availability of the required argument(s), which in this case is the address of the Bluetooth device to be connected to, with the name "bdaddr". Start with the following code:

```
#!/usr/bin/python3
import os
import json
import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'PUT':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        if not "bdaddr" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = args["bdaddr"]
else:
    print("ERROR: Not called by HTTP")
```

All that remains is to use the `connect(bdaddr)` function in the `bluetooth_gap` module to connect to the device with the specified address and return the result. Update your code so it looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gap
import bluetooth_constants

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'PUT':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        if not "bdaddr" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = args["bdaddr"]
            rc = bluetooth_gap.connect(bdaddr)
            result['result'] = rc
            print(json.JSONEncoder().encode(result))
else:
    print("ERROR: Not called by HTTP")
```

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

#### 4.4.4.2.3 Testing

You can test this script from the command line of your Raspberry Pi, using *curl*. Run the following command and observe the result:

```
pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --data '{"bdaddr" :
"D9:64:36:0E:87:01" }' --request PUT http://localhost/cgi-bin/gateway/do_connect.py
{"result": 0}
```

If you are using a BBC micro:bit with the supplied code running on it, the LED matrix will display a letter C to indicate that a Bluetooth connection has been accepted.

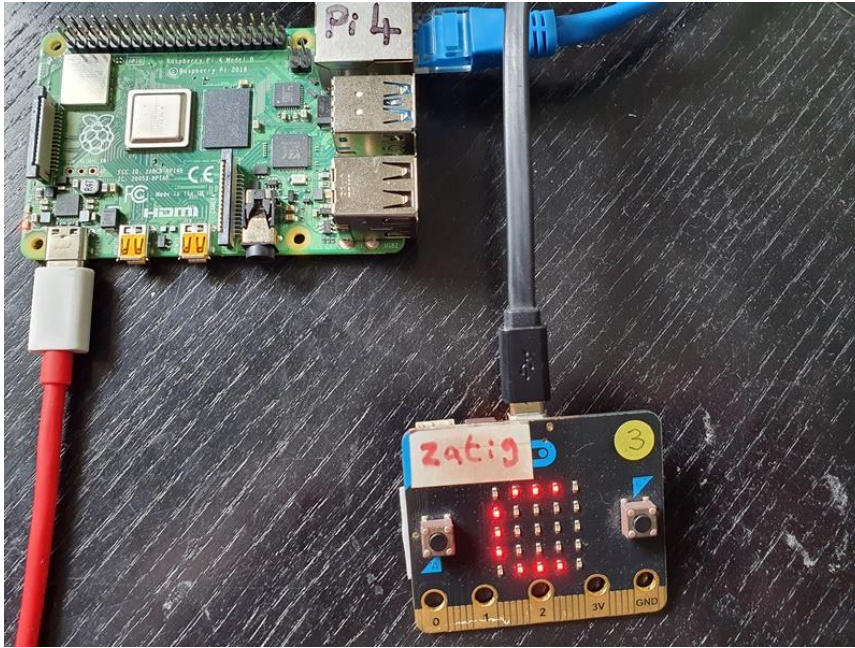


Figure 4 - micro:bit after accepting a Bluetooth connection

If you get a result code of (ERR\_NOT\_FOUND), run the device discovery script and then try again.

Check the HTTP method and argument presence validation.

```
pi@raspberrypi:~ $ curl --data '{"bdaddr" : "D9:64:36:0E:87:01" }' --request GET
http://localhost/cgi-bin/gateway/do_connect.py
Status-Line: HTTP/1.0 405 Method Not Allowed

pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --data '{"xxxxxx" :
"D9:64:36:0E:87:01" }' --request PUT http://localhost/cgi-bin/gateway/do_connect.py
{"result": 8}
```

Obviously, you should substitute the Bluetooth device address for your test device in the JSON data. You should be able to determine this address from the result of running the device discovery script.

Once again, if your test produced no output, check the end of /var/log/apache2/error.log for clues.

### 4.4.4.3 Task - Implement the Device Disconnect adapter script

#### 4.4.4.3.1 API Details

Request
do_disconnect.py
HTTP Method
PUT
Arguments Object
<pre>{"bdaddr" : "D9:64:36:0E:87:01"}</pre> where bdaddr is the Bluetooth device address of the device you wish to disconnect from.
Response Description
Returns a JSON object containing a result code.
Response Example - Successful Execution
<pre>{"result": 0}</pre>
Response Example - Error - Invalid Args
<pre>{"result": 8}</pre>
Response Example - Error - Not Found
<pre>{"result": 9}</pre>

#### 4.4.4.3.2 Coding

Make a copy of the do\_connect.py file called do\_disconnect.py. The new script should only differ from the do\_connect.py script in one respect. Instead of calling the bluetooth\_gap.connect function, it needs to call the bluetooth\_gap.disconnect. Edit the new script accordingly.

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

#### 4.4.4.3.3 Testing

Test the disconnect script by resetting your device, connecting to it and then disconnecting.

```
pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --data '{"bdaddr" :
"D9:64:36:0E:87:01" }' --request PUT http://localhost/cgi-bin/gateway/do_connect.py
{"result": 0}
```

If you get a result code of (ERR\_NOT\_FOUND), run the device discovery script and then try again.

Check the HTTP method and argument presence validation.

```
pi@raspberrypi:~ $ curl --request GET http://localhost/cgi-bin/gateway/do_connect.py
Status-Line: HTTP/1.0 405 Method Not Allowed

pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --data '{"xxxxxx" :
"D9:64:36:0E:87:01" }' --request PUT http://localhost/cgi-bin/gateway/do_connect.py
{"result": 8}
```

Obviously, you should substitute the Bluetooth device address for your test device in the JSON data. You should be able to determine this address from the result of running the device discovery script.

If you are using a BBC micro:bit with the supplied code running on it, the LED matrix will display a letter D to indicate that the Bluetooth disconnection has been closed.



Once again, if your test produced no output, check the end of `/var/log/apache2/error.log` for clues.

#### 4.4.4.4 Task - Implement the Service Discovery adapter script

##### 4.4.4.4.1 API Details

Request
do_service_discovery.py
HTTP Method
GET
Query String Arguments
bdaddr=D9:64:36:0E:87:01 where bdaddr is the Bluetooth device address of the device you wish to perform service discovery on.
Response Description
Returns an array of JSON objects, each representing a GATT service. Each of these objects contains an array of JSON objects which represent the characteristics owned by the service. Each characteristic object contains an array of JSON objects which represent the descriptors owned by the characteristic.
Response Example - Successful Execution
<pre>[{   "characteristics": [{     "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c/char000d",     "UUID": "e95d93b1-251d-470a-a062-fa1922dfa9a8",     "descriptors": [],     "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c",     "properties": ["read", "write"]   }],   "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c",   "UUID": "e95d93b0-251d-470a-a062-fa1922dfa9a8" }, {   "characteristics": [{     "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char0030",     "UUID": "e95d1b25-251d-470a-a062-fa1922dfa9a8",     "descriptors": [],     "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c",     "properties": ["read", "write"]   }, {     "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d",     "UUID": "e95d9250-251d-470a-a062-fa1922dfa9a8",     "descriptors": [{       "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d/desc002f",       "UUID": "00002902-0000-1000-8000-00805f9b34fb",       "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d"     }],     "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c",     "notifying": false,     "properties": ["read", "notify"]   }],   "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c",   "UUID": "e95d6100-251d-470a-a062-fa1922dfa9a8" }, {   "characteristics": [{     "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010",     "UUID": "e97d3b10-251d-470a-a062-fa1922dfa9a8",     "descriptors": [{       "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010/desc0012",       "UUID": "00002902-0000-1000-8000-00805f9b34fb",       "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010"     }],     "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f",     "notifying": false,     "properties": ["write-without-response", "notify"]   }], }</pre>



```

    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f",
    "UUID": "e97dd91d-251d-470a-a062-fa1922dfa9a8"
  }, {
    "characteristics": [{
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036",
      "UUID": "e95dda91-251d-470a-a062-fa1922dfa9a8",
      "descriptors": [{
        "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036/desc0038",
        "UUID": "00002902-0000-1000-8000-00805f9b34fb",
        "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036"
      }],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032",
      "notifying": false,
      "properties": ["read", "notify"]
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033",
      "UUID": "e95dda90-251d-470a-a062-fa1922dfa9a8",
      "descriptors": [{
        "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033/desc0035",
        "UUID": "00002902-0000-1000-8000-00805f9b34fb",
        "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033"
      }],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032",
      "notifying": false,
      "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032",
    "UUID": "e95d9882-251d-470a-a062-fa1922dfa9a8"
  }, {
    "characteristics": [{
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0016",
      "UUID": "00002a25-0000-1000-8000-00805f9b34fb",
      "descriptors": [],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
      "properties": ["read"]
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0018",
      "UUID": "00002a26-0000-1000-8000-00805f9b34fb",
      "descriptors": [],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
      "properties": ["read"]
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014",
      "UUID": "00002a24-0000-1000-8000-00805f9b34fb",
      "descriptors": [],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
      "properties": ["read"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
    "UUID": "0000180a-0000-1000-8000-00805f9b34fb"
  }, {
    "characteristics": [{
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001e",
      "UUID": "e95d5404-251d-470a-a062-fa1922dfa9a8",
      "descriptors": [],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
      "properties": ["write-without-response", "write"]
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b",
      "UUID": "e95d9775-251d-470a-a062-fa1922dfa9a8",
      "descriptors": [{
        "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b/desc001d",
        "UUID": "00002902-0000-1000-8000-00805f9b34fb",
        "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b"
      }],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
      "notifying": false,
      "properties": ["read", "notify"]
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0020",

```



```

        "UUID": "e95d23c4-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "properties": ["write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022",
        "UUID": "e95db84c-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022/desc0024",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "notifying": false,
        "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
    "UUID": "e95d93af-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char002a",
        "UUID": "e95d0d2d-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026",
        "UUID": "e95d7b77-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028",
        "UUID": "e95d93ee-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
        "properties": ["write"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
    "UUID": "e95dd91d-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003d",
        "UUID": "e95dfb24-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a",
        "UUID": "e95dca4b-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a/desc003c",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
        "notifying": false,
        "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
    "UUID": "e95d0753-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009",
        "UUID": "00002a05-0000-1000-8000-00805f9b34fb",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009/desc000b",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009"
        }],

```

<pre>                 "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008",                 "notifying": false,                 "properties": ["indicate"]             }},             "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008",             "UUID": "00001801-0000-1000-8000-00805f9b34fb"         }     ] </pre>
<b>Response Example - Error - Device is not connected</b>
<pre>{"result": 2}</pre>
<b>Response Example - Error - Invalid Args</b>
<pre>{"result": 8}</pre>
<b>Response Example - Error - Not Found</b>
<pre>{"result": 9}</pre>

Review the sample output and familiarise yourself with its objects and their properties. Note that there are a number of handle properties. These are unique identifiers for a specific instance of GATT attribute and are allocated by BlueZ. You should assume that they are volatile and may change across connections.

#### 4.4.4.4.2 Coding

Make a copy of the `do_connect.py` file called `do_service_discovery.py`. The new script will use the same nested if structure for checking the HTTP method and the presence of the `bdaddr` argument. The else block which executes when the validation checks have passed, will differ significantly from that which is currently in place, so delete that code.

Service discovery is a GATT procedure, so replace the `import bluetooth_gap` statement with one which imports the `bluetooth_gatt` module instead.

Your `do_service_discovery.py` file should currently look like this:

```
#!/usr/bin/python3
import os
import json
from sys import stdin, stdout
import cgi

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:

# this is where we will implement the service discovery code

else:
    print("ERROR: Not called by HTTP")
```

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

Initiate service discovery in the second else block by calling the Bluetooth API `bluetooth_gatt.discover_services` function, with an argument equal to the specified Bluetooth device address in the `bdaddr` property of the JSON object received over HTTP. Follow this by some temporary code which prints the results of this call, which will be a Python data structure. This is so you can see what you'll be working with in the next step.

```
#!/usr/bin/python3
import os
import json
from sys import stdin, stdout
import cgi

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = querystring.getfirst("bdaddr")
            try:
                services_discovered = bluetooth_gatt.get_services(bdaddr)
                # temporary code starts
                print(str(services_discovered))
                # temporary code ends

            except bluetooth_exceptions.StateError as e:
                result = {}
                result['result'] = e.args[0]
                print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")
```

Test the code as it currently stands, to generate some raw output. You'll need to connect to your device before you can run the service discovery script.

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --request GET http://localhost/cgi-bin/gateway/do_service_discovery.py?bdaddr= D9:64:36:0E:87:01
[{'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c', 'UUID': 'e95d93b0-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c', 'UUID': 'e95d6100-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f', 'UUID': 'e97dd91d-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032', 'UUID': 'e95d9882-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013', 'UUID': '0000180a-0000-1000-8000-00805f9b34fb'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a', 'UUID': 'e95d93af-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025', 'UUID': 'e95dd91d-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039', 'UUID': 'e95d0753-251d-470a-a062-fa1922dfa9a8'}, {'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008', 'UUID': '00001801-0000-1000-8000-00805f9b34fb'}]
```

The raw output is hard to read. Despite this not being JSON, you can still format it with the <https://jsonlint.com/> validation tool. The test output in the previous panel looks like this after formatting:

```
[{
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c',
  'UUID': 'e95d93b0-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c',
  'UUID': 'e95d6100-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f',
  'UUID': 'e97dd91d-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032',
  'UUID': 'e95d9882-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013',
  'UUID': '0000180a-0000-1000-8000-00805f9b34fb'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
  'UUID': 'e95d93af-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025',
  'UUID': 'e95dd91d-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039',
  'UUID': 'e95d0753-251d-470a-a062-fa1922dfa9a8'
}, {
  'path': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008',
  'UUID': '00001801-0000-1000-8000-00805f9b34fb'
}]
```

Clearly we have a list of GATT services, with their UUID and an attribute called the *path*. BlueZ uses the term *path* rather than *handle*, for the unique identifier of a GATT attribute in the remote device's attribute table.

The Bluetooth API, which is designed solely for use with BlueZ, also uses the term *path* in its function arguments and results. However, we require our gateway API to use the more standard term *handle* and so one of the things the adapter code must do is to replace attribute names containing *path*, as returned by BlueZ, with names involving the more standard term *handle*.

To complete this script, you must add a nested for loop which acquires an array of characteristics for each of the services in our initial array, and then for each characteristic, acquires an array of the descriptors owned. Attribute names must be modified per the following table:

Old Name	New Name
path	handle
service_path	service_handle
characteristic_path	characteristic_handle

A results data structure, containing the acquired and renamed services, characteristics and descriptors in a series of nested arrays, must be also built.

The end result of this process should be a nested set of arrays as described in the API Details section, encoded as a JSON object and returned to the HTTP client by writing it to standard output in the way in which you have done in previous scripts.

The `bluetooth_gatt` Bluetooth API module contains the following functions, which we will use:

```
get_characteristics(bdaddr, service_path)
get_descriptors(bdaddr, characteristic_path)
```

Note that each required a `bdaddr` argument and the BlueZ path (identifier) of the parent attribute whose child characteristics or descriptors we are requesting.

Update your script to acquire and rename characteristics for each service obtained:

```
#!/usr/bin/python3
import os
import json
from sys import stdin, stdout
import cgi

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = querystring.getfirst("bdaddr")
            try:
                services_discovered = bluetooth_gatt.get_services(bdaddr)

                for service in services_discovered:
                    # rename BlueZ-specific parameter name to the more abstract 'handle'
                    service['handle'] = service.pop('path')
                    characteristics_discovered = bluetooth_gatt.get_characteristics(bdaddr,
service['handle'])

                    service['characteristics'] = characteristics_discovered
                    for characteristic in characteristics_discovered:
                        characteristic['handle'] = characteristic.pop('path')
                        characteristic['service_handle'] =
characteristic.pop('service_path')

            # temporary code starts
            print(str(services_discovered))
            # temporary code ends

            except bluetooth_exceptions.StateError as e:
                result = {}
                result['result'] = e.args[0]
                print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")
```

The new code iterates through the services in the `services_discovered` array. These are Python [dictionaries](#), which are similar to JSON objects, containing name/value pairs. For each service, the property called *path* is removed from the dictionary and replaced with a property called *handle* with the same value. Characteristics belonging to the service are then acquired using the `get_characteristics` function. The discovered characteristics are added as property of the service and then for each characteristic returned, its *path* property is renamed *handle* and its *service\_path*

property is renamed *service\_handle*. Once again, some temporary code prints the results obtained so far.

Run an interim test and format the output for easier scrutiny.

```
pi@raspberrypi:~ $ curl --request GET http://localhost/cgi-bin/gateway/do_service_discovery.py?bdaddr=D9:64:36:0E:87:01

# formatted output

[[
  {
    'characteristics': [{
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c/char000d',
      'UUID': 'e95d93b1-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c',
      'properties': ['read', 'write']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c',
    'UUID': 'e95d93b0-251d-470a-a062-fa1922dfa9a8'
  }, {
    'characteristics': [{
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char0030',
      'UUID': 'e95d1b25-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c',
      'properties': ['read', 'write']
    }, {
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d',
      'UUID': 'e95d9250-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c',
      'notifying': False,
      'properties': ['read', 'notify']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c',
    'UUID': 'e95d6100-251d-470a-a062-fa1922dfa9a8'
  }, {
    'characteristics': [{
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010',
      'UUID': 'e97d3b10-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f',
      'notifying': False,
      'properties': ['write-without-response', 'notify']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f',
    'UUID': 'e97dd91d-251d-470a-a062-fa1922dfa9a8'
  }, {
    'characteristics': [{
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036',
      'UUID': 'e95dda91-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032',
      'notifying': False,
      'properties': ['read', 'notify']
    }, {
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033',
      'UUID': 'e95dda90-251d-470a-a062-fa1922dfa9a8',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032',
      'notifying': False,
      'properties': ['read', 'notify']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032',
    'UUID': 'e95d9882-251d-470a-a062-fa1922dfa9a8'
  }, {
    'characteristics': [{
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0016',
      'UUID': '00002a25-0000-1000-8000-00805f9b34fb',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013',
      'properties': ['read']
    }, {
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0018',
      'UUID': '00002a26-0000-1000-8000-00805f9b34fb',
      'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013',
      'properties': ['read']
    }, {
      'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014',
```

```

        'UUID': '00002a24-0000-1000-8000-00805f9b34fb',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013',
        'properties': ['read']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013',
    'UUID': '0000180a-0000-1000-8000-00805f9b34fb'
}, {
    'characteristics': [{
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001e',
        'UUID': 'e95d5404-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
        'properties': ['write-without-response', 'write']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b',
        'UUID': 'e95d9775-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
        'notifying': False,
        'properties': ['read', 'notify']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0020',
        'UUID': 'e95d23c4-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
        'properties': ['write']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022',
        'UUID': 'e95db84c-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
        'notifying': False,
        'properties': ['read', 'notify']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a',
    'UUID': 'e95d93af-251d-470a-a062-fa1922dfa9a8'
}, {
    'characteristics': [{
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char002a',
        'UUID': 'e95d0d2d-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025',
        'properties': ['read', 'write']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026',
        'UUID': 'e95d7b77-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025',
        'properties': ['read', 'write']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028',
        'UUID': 'e95d93ee-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025',
        'properties': ['write']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025',
    'UUID': 'e95dd91d-251d-470a-a062-fa1922dfa9a8'
}, {
    'characteristics': [{
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003d',
        'UUID': 'e95dfb24-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039',
        'properties': ['read', 'write']
    }, {
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a',
        'UUID': 'e95dca4b-251d-470a-a062-fa1922dfa9a8',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039',
        'notifying': False,
        'properties': ['read', 'notify']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039',
    'UUID': 'e95d0753-251d-470a-a062-fa1922dfa9a8'
}, {
    'characteristics': [{
        'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009',
        'UUID': '00002a05-0000-1000-8000-00805f9b34fb',
        'service_handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008',
        'notifying': False,
        'properties': ['indicate']
    }],
    'handle': '/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008',
    'UUID': '00001801-0000-1000-8000-00805f9b34fb'
}

```

```
}]
```

For each characteristic, we need to proceed in fundamentally the same way, obtaining the descriptors owned and updating their property names. Change your script so that it looks like this. Note that our final step is to convert the Python data structure to a JSON object.

```
#!/usr/bin/python3
import os
import json
from sys import stdin, stdout
import cgi

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = querystring.getfirst("bdaddr")
            try:
                services_discovered = bluetooth_gatt.get_services(bdaddr)

                for service in services_discovered:
                    # rename BlueZ-specific parameter name to the more abstract 'handle'
                    service['handle'] = service.pop('path')
                    characteristics_discovered = bluetooth_gatt.get_characteristics(bdaddr,
service['handle'])

                    service['characteristics'] = characteristics_discovered
                    for characteristic in characteristics_discovered:
                        characteristic['handle'] = characteristic.pop('path')
                        characteristic['service_handle'] =
characteristic.pop('service_path')
                        descriptors_discovered = bluetooth_gatt.get_descriptors(bdaddr,
characteristic['handle'])

                        characteristic['descriptors'] = descriptors_discovered
                        for descriptor in descriptors_discovered:
                            descriptor['handle'] = descriptor.pop('path')
                            descriptor['characteristic_handle'] =
descriptor.pop('characteristic_path')

                        attributes_discovered_json = json.JSONEncoder().encode(services_discovered)
                        print(attributes_discovered_json)
            except bluetooth_exceptions.StateError as e:
                result = {}
                result['result'] = e.args[0]
                print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")
```

#### 4.4.4.3 Testing

Test the completed service discovery script by resetting your device, connecting to it and then invoking the new script. The following example presents output results which have been formatted for easier reading.



```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ curl --request GET http://localhost/cgi-bin/gateway/do_service_discovery.py?bdaddr=D9:64:36:0E:87:01
[[{"characteristics": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c/char000d", "UUID": "e95d93b1-251d-470a-a062-fa1922dfa9a8", "descriptors": [], "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c", "properties": ["read", "write"]}], {"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c", "UUID": "e95d93b0-251d-470a-a062-fa1922dfa9a8"}], [{"characteristics": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char0030", "UUID": "e95dlb25-251d-470a-a062-fa1922dfa9a8", "descriptors": [], "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c", "properties": ["read", "write"]} ], [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d", "UUID": "e95d9250-251d-470a-a062-fa1922dfa9a8", "descriptors": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d/desc002f", "UUID": "00002902-0000-1000-8000-00805f9b34fb", "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d"}, {"service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c", "notifying": false, "properties": ["read", "notify"]} ]}], {"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c", "UUID": "e95d6100-251d-470a-a062-fa1922dfa9a8"}], [{"characteristics": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010", "UUID": "e97d3b10-251d-470a-a062-fa1922dfa9a8", "descriptors": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010/desc0012", "UUID": "00002902-0000-1000-8000-00805f9b34fb", "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f/char0010"}, {"service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f", "notifying": false, "properties": ["write-without-response", "notify"]} ]}], {"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000f", "UUID": "e97dd91d-251d-470a-a062-fa1922dfa9a8"}], [{"characteristics": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036", "UUID": "e95dda91-251d-470a-a062-fa1922dfa9a8", "descriptors": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036/desc0038", "UUID": "00002902-0000-1000-8000-00805f9b34fb", "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0036"}, {"service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032", "notifying": false, "properties": ["read", "notify"]} ]}], [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033", "UUID": "e95dda90-251d-470a-a062-fa1922dfa9a8", "descriptors": [{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033/desc0035", "UUID": "00002902-0000-1000-8000-00805f9b34fb", "characteristic_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032/char0033"} ]}]
```

```

        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032",
        "notifying": false,
        "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0032",
    "UUID": "e95d9882-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0016",
        "UUID": "00002a25-0000-1000-8000-00805f9b34fb",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
        "properties": ["read"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0018",
        "UUID": "00002a26-0000-1000-8000-00805f9b34fb",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
        "properties": ["read"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014",
        "UUID": "00002a24-0000-1000-8000-00805f9b34fb",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
        "properties": ["read"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",
    "UUID": "0000180a-0000-1000-8000-00805f9b34fb"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001e",
        "UUID": "e95d5404-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "properties": ["write-without-response", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b",
        "UUID": "e95d9775-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b/desc001d",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char001b"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "notifying": false,
        "properties": ["read", "notify"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0020",
        "UUID": "e95d23c4-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "properties": ["write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022",
        "UUID": "e95db84c-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022/desc0024",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a/char0022"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
        "notifying": false,
        "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service001a",
    "UUID": "e95d93af-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char002a",
        "UUID": "e95d0d2d-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",

```

```

        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026",
        "UUID": "e95d7b77-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028",
        "UUID": "e95d93ee-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
        "properties": ["write"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
    "UUID": "e95dd91d-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003d",
        "UUID": "e95dfb24-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
        "properties": ["read", "write"]
    }, {
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a",
        "UUID": "e95dca4b-251d-470a-a062-fa1922dfa9a8",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a/desc003c",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039/char003a"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
        "notifying": false,
        "properties": ["read", "notify"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0039",
    "UUID": "e95d0753-251d-470a-a062-fa1922dfa9a8"
}, {
    "characteristics": [{
        "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009",
        "UUID": "00002a05-0000-1000-8000-00805f9b34fb",
        "descriptors": [{
            "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009/desc000b",
            "UUID": "00002902-0000-1000-8000-00805f9b34fb",
            "characteristic_handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008/char0009"
        }],
        "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008",
        "notifying": false,
        "properties": ["indicate"]
    }],
    "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0008",
    "UUID": "00001801-0000-1000-8000-00805f9b34fb"
}]

```

Check the HTTP method and argument presence validation in the usual way.

Once again, if your test produced no output, check the end of `/var/log/apache2/error.log` for clues.

#### 4.4.4.5 Task - Implement the Read Characteristic Value adapter script

##### 4.4.4.5.1 API Details

<b>Request</b>
do_read_characteristic.py
<b>HTTP Method</b>
GET
<b>Query String Arguments</b>
bdaddr=D9:64:36:0E:87:01 handle=/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026 where bdaddr is the Bluetooth device address of the device containing the characteristic whose value you wish to read and handle is the BlueZ path to the required characteristic.
<b>Response Description</b>
Returns a JSON object containing the bdaddr and handle arguments from the response and a hex encoded byte array value.
<b>Response Example - Successful Execution</b>
<pre>{"bdaddr": "D9:64:36:0E:87:01", "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014", "result": 0, "value": "424243206D6963726F3A626974"}</pre> <p>The value attribute is hex encoded.</p>
<b>Response Example - Error - Device is not connected</b>
<pre>{"result": 2}</pre>
<b>Response Example - Error - Invalid Args</b>
<pre>{"result": 8}</pre>
<b>Response Example - Error - Not Found</b>
<pre>{"result": 9}</pre>

##### 4.4.4.5.2 Coding

Create a new file called do\_read\_characteristic.py in the /usr/lib/cgi-bin/gateway directory. Give it the same nested if statement for validating the HTTP method (GET) and arguments (bdaddr and handle) so that it looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
import cgi
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        if not "handle" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
```

```
else:
    print("ERROR: Not called by HTTP")
```

Update the code to create a result object, with the bdaddr and handle properties populated from the input arguments, read the characteristic value by calling `bluetooth_gatt.read_characteristic(bdaddr,handle)` and add the value as a hex encoded string and a property called `result`, with a value of zero if there were no errors. There's a function called `bluetooth_utils.byteArrayToHexString` which will perform the required hex encoding. Your code should look something like this:

```
#!/usr/bin/python3
import os
import json
import sys
import cgi
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        if not "handle" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = querystring.getfirst("bdaddr")
            handle = querystring.getfirst("handle")
            result = {}
            result['bdaddr'] = bdaddr
            result['handle'] = handle
            try:
                value = bluetooth_gatt.read_characteristic(bdaddr,handle)

# temporary code starts
            print(str(value))
            print(''.join(chr(i) for i in value))
# temporary code ends

            result['value'] = bluetooth_utils.byteArrayToHexString(value);
            result['result'] = 0
            print(json.JSONEncoder().encode(result))
        except bluetooth_exceptions.StateError as e:
            result['result'] = e.args[0]
            print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")
```

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

Note the temporary lines of code, which will help us to test this script. After testing, these lines should be removed.

#### 4.4.4.2.3 Testing

To test this script, we need a characteristic which supports the GATT read procedure. The BBC micro:bit supports the Device Information Service and this has a characteristic called Model Number String, with UUID equal to 00002a24-0000-1000-8000-00805f9b34fb when expressed as a full, 128-bit value. Run your service discovery test again and look for the characteristic with this UUID in the output. From this you should be able to obtain the BlueZ path which identifies this characteristic instance. For example:

```
    }, {  
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014",  
      "UUID": "00002a24-0000-1000-8000-00805f9b34fb",  
      "descriptors": [],  
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013",  
      "properties": ["read"]  
    }  
  ],  
}
```

Test the read characteristic script by connecting to your device, performing service discovery and then issuing an HTTP request similar to this one, using curl:

```
pi@raspberrypi:~ $ curl --request GET "http://localhost/cgi-bin/gateway/do_read_characteristic.py?bdaddr=D9:64:36:0E:87:01&handle=/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014"  
Content-Type: application/json; charset=utf-8  
  
[66, 66, 67, 32, 109, 105, 99, 114, 111, 58, 98, 105, 116]  
BBC micro:bit  
{ "bdaddr": "D9:64:36:0E:87:01", "handle":  
  "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014", "result": 0, "value":  
  "424243206D6963726F3A626974" }
```

**Tip:** the URL parameter to *curl* must be enclosed in quotes in this case. This is because without them, the shell will interpret the ampersand in the query string as marking the end of the command and the second parameter (handle) will be lost.

The JSON output is highlighted. The two interim lines show the characteristic value as a decimal byte array and then interpreted as an ASCII string. If you tested against the Model Number String characteristic in a BBC micro:bit, this is the result you should have obtained.

Once satisfied everything is working, remove the temporary lines.

#### 4.4.4.6 Task - Implement the Write Characteristic Value adapter script

##### 4.4.4.6.1 API Details

Request
do_write_characteristic.py
HTTP Method
PUT
Arguments Object
<pre>{ "bdaddr" : "D9:64:36:0E:87:01" , "handle" :   "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026", "value" : "48656c6c6f"   where bdaddr is the Bluetooth device address of the device containing the characteristic   whose value you wish to write to, handle is the BlueZ path to the required characteristic   and value is a hex encoded array of bytes.</pre>
Response Description
Returns a JSON object containing a result code.
Response Example - Successful Execution
<pre>{ "bdaddr": "D9:64:36:0E:87:01", "handle":   "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0013/char0014", "result": 0 }</pre> <p>The value attribute is hex encoded.</p>
Response Example - Error - Device is not connected
<pre>{ "result": 2 }</pre>
Response Example - Error - Invalid Args
<pre>{ "result": 8 }</pre>
Response Example - Error - Not Found
<pre>{ "result": 9 }</pre>

##### 4.4.4.6.2 Coding

Create a new file called do\_write\_characteristic.py in the /usr/lib/cgi-bin/gateway directory. Give it the same nested if statement for validating the HTTP method (GET) and arguments (bdaddr and handle) so that it looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'PUT':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        if not "bdaddr" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "handle" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "value" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
```

```
else:
    print("ERROR: Not called by HTTP")
```

Update the code to create a result object, with the bdaddr and handle properties populated from the input arguments, read the characteristic value by calling `bluetooth_gatt.write_characteristic(bdaddr,handle,value)` and add to the response object, the characteristic value as a hex encoded string and a property called result, with a value of zero if there were no errors. There's a function called `bluetooth_utils.byteArrayToHexString` which will perform the required hex encoding. Your code should look something like this:

```
#!/usr/bin/python3
import os
import json
import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'PUT':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        if not "bdaddr" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "handle" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "value" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = args["bdaddr"]
            handle = args["handle"]
            value = args["value"]
            result = {}
            result['bdaddr'] = bdaddr
            result['handle'] = handle
            try:
                rc = bluetooth_gatt.write_characteristic(bdaddr,handle,value)
                result['result'] = rc
                print(json.JSONEncoder().encode(result))
            except bluetooth_exceptions.StateError as e:
                result['result'] = e.args[0]
                print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")
```

Set the ownership and permissions of the new Python script:

```
pi@raspberrypi:~ $ sudo chown -R www-data:www-data /usr/lib/cgi-bin/gateway/*
pi@raspberrypi:~ $ sudo chmod -R 775 /usr/lib/cgi-bin/gateway/*
```

#### 4.4.4.6.3 Testing

To test this script, we need a characteristic which supports the GATT write procedure. The BBC micro:bit, when running the provided binary, supports the LED Service which provides access to the



LED display. This service includes a characteristic called LED Text with UUID equal to e95d93ee-251d-470a-a062-fa1922dfa9a8. Run your service discovery test again and look for the characteristic with this UUID in the output. From this you should be able to obtain the BlueZ path which identifies this characteristic instance. For example:

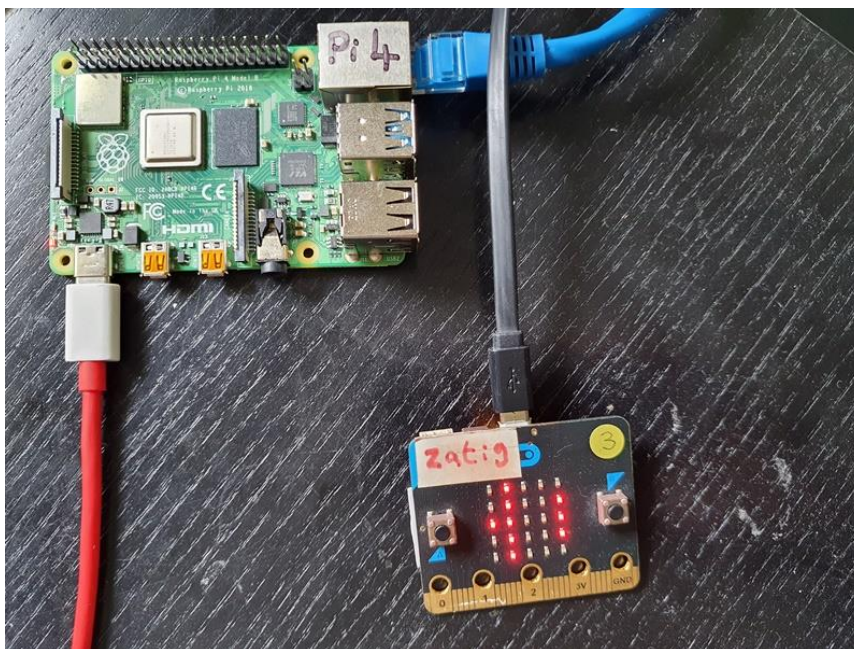
```
    }, {
      "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028",
      "UUID": "e95d93ee-251d-470a-a062-fa1922dfa9a8",
      "descriptors": [],
      "service_handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025",
      "properties": ["write"]
    }
  ],
}
```

Test the write characteristic script by connecting to your device, performing service discovery and then issuing an HTTP request similar to this one, using curl:

```
pi@raspberrypi:~ $ curl --header "Content-Type: application/json" --data '{"bdaddr" :
"D9:64:36:0E:87:01" , "handle" :
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028" , "value" : "48656c6c6f" }' -
-request PUT http://localhost/cgi-bin/gateway/do_write_characteristic.py
Content-Type: application/json; charset=utf-8

{"bdaddr": "D9:64:36:0E:87:01", "handle":
"/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0028", "result": 0}
```

The expected JSON output is highlighted. If you tested against the LED Text characteristic in a BBC micro:bit, you should have seen “Hello” scroll across the micro:bit display.



#### 4.4.4.7 Task - implement support for notifications and indications

##### 4.4.4.7.1 Architecture and Approach

For this set of requirements, we will be using a web socket connection for both control purposes, enabling and disabling notifications and indications on specified characteristics, and for the delivery of characteristic value notifications and indications.

The procedure for enabling and receiving notifications or indications will be as follows:

1. Client establishes a web socket connection to the web socket daemon running on the gateway.
2. Client writes a JSON control object to the connection to enable notifications on a specified characteristic.
3. Client repeatedly receive a JSON object containing a characteristic value.
4. Eventually, the client writes another JSON control object to the connection to disable notifications.

On accepting a connection from a client, the websocketd daemon, will fork a process which executes an instance of a script, specified to websocketd when running it. That script will read from standard input to receive input from the web socket connection and write to standard output to write to it. When the client closes the web socket connection, websocketd will terminate the associated script.

##### 4.4.4.7.2 API Details

Request - Enabling Notifications
<pre>{   "bdaddr" : "D9:64:36:0E:87:01",   "handle" : "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c/char000d",   "command" : 1 };</pre>
Request - Enabling Notifications
<pre>{   "bdaddr" : "D9:64:36:0E:87:01",   "handle" : "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service000c/char000d",   "command" : 0 };</pre>
Request - Exit Notifications Adapter
<pre>{   "command" : 9 };</pre>
Response - Characteristic Value
<pre>{"bdaddr": "D9:64:36:0E:87:01", "handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service0025/char0026", "result": 0, "value": "1A"}</pre>
Response - Error
<pre>{"result": error code}</pre>

##### 4.4.4.7.3 Coding

To implement support for notifications and indications, we need a Python script which websocketd will run and communicate with over the standard input and output streams.

Create do\_notifications.py in /usr/lib/cgi-bin/gateway and populate it with the initial code as shown here:

```
#!/usr/bin/python3
import json
from sys import stdin, stdout

import sys
```

```

sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

bdaddr = None
handle = None

def notification_received(path,value):
    # to be completed

keep_going = 1
bluetooth_utils.log("==== do_notifications =====\n")
while keep_going == 1:
    line = stdin.readline()
    bluetooth_utils.log(line+"\n")
    if len(line) == 0:
        # means websocket has closed
        keep_going = 0
    else:
        line = line.strip()
        notifications_control = json.loads(line)
        bluetooth_utils.log("command="+str(notifications_control['command'])+"\n");
        result = {}

print("WebSocket handler has exited")
stdout.flush()

```

As you can see, this script will sit in a while loop, reading from the standard input stream, over which data written to the web socket connection by the client, will be received. An empty line indicates that the web socket connection has been closed, so we set the *keep\_going* loop control to 0 and exit. Otherwise, we strip white space from the beginning and end of the line, parse the string as a JSON object called *notifications\_control* and prepare to process the object and return a result.

The *command* property of the *notifications\_control* object, determines the action to be taken. A value of 1 means the client is requesting that notifications/indications be enabled on a characteristic whose handle and device address is also specified. Update your script to accommodate this case, as follows:

```

#!/usr/bin/python3
import json
from sys import stdin, stdout

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils

bdaddr = None
handle = None

def notification_received(path,value):
    # TODO

keep_going = 1
bluetooth_utils.log("==== do_notifications =====\n")
while keep_going == 1:
    line = stdin.readline()
    bluetooth_utils.log(line+"\n")
    if len(line) == 0:
        # means websocket has closed
        keep_going = 0
    else:
        line = line.strip()
        notifications_control = json.loads(line)

```

```

bluetooth_utils.log("command="+str(notifications_control['command'])+"\n");
result = {}

if notifications_control['command'] == 1:
    bdaddr = notifications_control['bdaddr']
    handle = notifications_control['handle']
    result['bdaddr'] = bdaddr
    result['handle'] = handle
    try:
        bluetooth_gatt.enable_notifications(bdaddr, handle, notification_received)
        result['result'] = bluetooth_constants.RESULT_OK;
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.StateError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.UnsupportedError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()

print("WebSocket handler has exited")
stdout.flush()

```

In the new code block, we extract the `bdaddr` and `handle` parameters from the JSON control object and then use the Bluetooth API to enable notifications/indications on the attribute identified by that handle value. The third argument, *notification\_received* is the name of a function, defined near the top of this script. The Bluetooth API will call this function when the device delivers a characteristic value notification or indication, so that it can be passed to the web socket client. Currently, this function is empty.

We write a response object containing *result* to the output stream to reply to the client. If something goes wrong, a Python exception will be caught, and its type indicated in the response object. Two potential issues are dealt with; either the device is in the wrong state because it has not been connected to or service discovery has not yet taken place, or notifications/indications are not supported by the attribute identified by the specified handle.

Next, add code to handle the case where the client wants to disable notifications / indications. Rather than repeat the whole script, only the section involving the required change is shown here:

```

if notifications_control['command'] == 1:
    bdaddr = notifications_control['bdaddr']
    handle = notifications_control['handle']
    result['bdaddr'] = bdaddr
    result['handle'] = handle
    try:
        bluetooth_gatt.enable_notifications(bdaddr, handle, notification_received)
        result['result'] = bluetooth_constants.RESULT_OK;
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.StateError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.UnsupportedError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()

elif notifications_control['command'] == 0:
    bdaddr = notifications_control['bdaddr']
    handle = notifications_control['handle']
    result['bdaddr'] = bdaddr
    result['handle'] = handle
    try:
        bluetooth_utils.log("calling disable notifications\n")

```

```

        rc = bluetooth_gatt.disable_notifications(bdaddr, handle)
        bluetooth_utils.log("done calling disable_notifications\n")
        result['result'] = bluetooth_constants.RESULT_OK;
        print(json.JSONEncoder().encode(result))
        stdout.flush()
        bluetooth_utils.log("finished\n")
    except bluetooth_exceptions.StateError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.UnsupportedError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()

print("WebSocket handler has exited")
stdout.flush()

```

The new *elif* block is very similar to the block handling enabling notifications/indications. The key difference is the use of the Bluetooth API function *bluetooth\_gatt.disable\_notifications*.

The final control request which our script must handle, has the *command* property set to a value of 9 and this indicates that the processing loop and script should be exited from. Update your script to handle this case.

```

elif notifications_control['command'] == 0:
    bdaddr = notifications_control['bdaddr']
    handle = notifications_control['handle']
    result['bdaddr'] = bdaddr
    result['handle'] = handle
    try:
        bluetooth_utils.log("calling disable_notifications\n")
        rc = bluetooth_gatt.disable_notifications(bdaddr, handle)
        bluetooth_utils.log("done calling disable_notifications\n")
        result['result'] = bluetooth_constants.RESULT_OK;
        print(json.JSONEncoder().encode(result))
        stdout.flush()
        bluetooth_utils.log("finished\n")
    except bluetooth_exceptions.StateError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()
    except bluetooth_exceptions.UnsupportedError as e:
        result['result'] = e.args[0]
        print(json.JSONEncoder().encode(result))
        stdout.flush()

elif notifications_control['command'] == 9:
    keep_going = 0

```

Handling command 9 is very simple. We just need to set the loop control variable *keep\_going* to 0.

All that remains is to implement the *notification\_received* callback function. Add the following:

```

def notification_received(path, value):
    result = {}
    result['bdaddr'] = bdaddr
    result['handle'] = path
    result['value'] = bluetooth_utils.byteArrayToHexString(value)
    print(json.JSONEncoder().encode(result))
    stdout.flush()

```

The function received the path (handle) identifying the characteristic that the value relates to, and a value as a Python variable. We create a JSON object and populate it with the Bluetooth device address that was provided when notifications were enabled, the characteristic path name “handle”,

per our API conventions and the characteristic value as a hex encoded byte array. We then write the object to the standard output stream for transmission over the web socket connection to the client.

#### 4.4.4.7.4 Testing

Testing our notifications adapter script, will take a different approach to the testing of the other adapters which involved sending HTTP requests using the command line tool, *curl*. Instead, we will use a Python module and commands issued within the interactive Python interpreter. Install the WebSocket-client module as follows:

```
sudo pip3 install websocket-client
```

Start the websocketd daemon with parameters as shown here:

```
websocketd --port=8082 /usr/lib/cgi-bin/gateway/do_notifications.py
```

In another terminal, start a Python interpreter by executing the command *python3*. From within the shell that starts, issue the following commands to test enabling and receiving notifications, disabling and exiting the script.

```
# get ready
import websocket
import json
ws = websocket.WebSocket()
ws.connect("ws://localhost:8082")
command = {}

# enable temperature notifications (device must have been connected to using curl)
command['command'] = 1
command['bdaddr'] = "D9:64:36:0E:87:01"
command['handle'] = "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d"
ws.send(json.JSONEncoder().encode(command))
ws.recv()

# receive multiple notifications, executing ws.recv() around once every second
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()
ws.recv()

# disable notifications
command['command'] = 0
ws.send(json.JSONEncoder().encode(command))
ws.recv()

# terminate and exit
command['command'] = 9
ws.send(json.JSONEncoder().encode(command))
ws.recv()
```

Your session should look something like this example:

```
pi@raspberrypi:~ $ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import websocket
ws.send(json.JSONEncoder().encode(command))
ws.recv()
ws.recv()>>> import json
```

[illegible]

If you have problems, watch the output from the `websocketd` daemon, which should look something like this under normal circumstances:

```
pi@raspberrypi:/usr/lib/cgi-bin/gateway $ websocketd --port=8082 /usr/lib/cgi-  
bin/gateway/do_notifications.py
```

```

Thu, 30 Apr 2020 11:25:35 +0100 | INFO | server | | Serving using application :
/usr/lib/cgi-bin/gateway/do_notifications.py
Thu, 30 Apr 2020 11:25:35 +0100 | INFO | server | | Starting WebSocket server :
ws://raspberrypi:8082/
Thu, 30 Apr 2020 11:26:01 +0100 | ACCESS | session | url:'http://localhost:8082/'
id:'1588242361756632758' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' | CONNECT
Thu, 30 Apr 2020 11:27:13 +0100 | ACCESS | session | url:'http://localhost:8082/'
id:'1588242361756632758' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4861' | DISCONNECT

```

Python scripting errors in your do\_notifications.py script will give rise to ERROR lines in the websocketd daemon output, like this:

```

Thu, 30 Apr 2020 11:22:39 +0100 | ACCESS | session | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' | CONNECT
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' |
ERROR:dbus.connection:Exception in handler for D-Bus signal:
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' | Traceback
(most recent call last):
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' | File
"/usr/lib/python2.7/dist-packages/dbus/connection.py", line 230, in maybe_handle_message
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' |
self._handler(*args, **kwargs)
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' | File
"../bluetooth/bluetooth_gatt.py", line 231, in properties_changed
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' |
notifications_callback(path,value)
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' | File
"/usr/lib/cgi-bin/gateway/do_notifications.py", line 20, in notification_received
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' |
bluetooth_utils.log("sending value "+value+"\n")
Thu, 30 Apr 2020 11:22:55 +0100 | ERROR | stderr | url:'http://localhost:8082/'
id:'1588242159872482194' remote ':::1' command:'/usr/lib/cgi-
bin/gateway/do_notifications.py' origin:'http://localhost:8082' pid:'4797' | TypeError:
cannot concatenate 'str' and 'dbus.Array' objects

```

The Bluetooth API also includes the bluetooth\_utils.log() function, which you can use to output information to a log file called log.txt.

```

def notification_received(path,value):
    result = {}
    result['bdaddr'] = bdaddr
    result['handle'] = path
    result['value'] = bluetooth_utils.byteArrayToHexString(value)
    bluetooth_utils.log("sending value "+bluetooth_utils.byteArrayToHexString(value)+"\n")
    print(json.JSONEncoder().encode(result))
    stdout.flush()

```

```

pi@raspberrypi:/usr/lib/cgi-bin/gateway $ tail -f log.txt
command=1
===== do_notifications =====

```



```
{"handle": "/org/bluez/hci0/dev_D9_64_36_0E_87_01/service002c/char002d", "bdaddr": "D9:64:36:0E:87:01", "command": "1"}  
command=1
```

#### *4.4.4.8 Task Completed*

Congratulations! If you've made it this far, then hopefully this means you have a set of working adapter scripts which use the Bluetooth API to provide support for device discovery, connecting and disconnecting to a device, service discovery, reading and writing characteristics and working with notifications and indications. In short, you have a working gateway!

## 5. Project - Building Gateway Applications

### 5.1 Introduction

You should now have a working gateway which supports connecting to and interacting with Bluetooth LE Peripherals. This section will review and explore applications that use gateways of this sort and will give you some experience of developing one. If you didn't complete the exercises in section 4, you can still set up a working gateway, using the adapter scripts in the `implementation/solutions/peripherals/gateway_adapters_no_security` directory.

Applications can take many different forms. All that is required, when developing a gateway client application is that it be possible to work with HTTP and web sockets. You may be using a language which has high level APIs for both these things, or you may be working with a constrained microcontroller, with rudimentary support for TCP/IP sockets only, meaning that you will need to develop the HTTP and web socket capability.

The most common platforms to use a Bluetooth internet gateway are likely to be mobile applications and web applications, all of which have rich collections of APIs.

We're going to focus on web applications since they are the easiest and fastest to work with and even if you have no prior experience with HTML, JavaScript, AJAX and the web socket APIs, the learning curve is shallow.

To begin with, we will install and explore a full application which is included in this study guide. You will then develop a simple application which will allow you to control aspects of a BBC micro:bit.

### 5.2 Explore Blue

#### 5.2.1 Task - Set up Explore Blue

Explore Blue is a generalized gateway application which allows you to discover Bluetooth devices and explore and use the capabilities they are discovered to have. It's designed for developers and technical architects, not for end users.

Install Explore Blue by following these steps:

1. Create the directory `/var/www/html/explore_blue` and copy the contents of the study guide package at `implementation/solutions/peripherals/explore_blue` into it.
2. Set owner and group of the new directory and its contents to `www-data` with `chown -R www-data:www-data /var/www/html/explore_blue`. Set permissions with `chmod -R 775 /var/www/html/explore_blue`

To enable the system, you must start the `websocketd` daemon, ensure it listens on the correct port and that the script it launches, `do_notifications.py`, has access to the Bluetooth API modules in `/usr/lib/cgi-bin/bluetooth`. Therefore you should now

1. Review `explore_blue/js/constants.js`. You will see the websocket URL defined there and note that the host name to be used is `raspberrypi` and the port is 8888. You should already have taken steps to ensure that the hostname `raspberrypi` resolves to the IP address of your gateway machine, when used on the remote PC you are running a web browser on.
2. On your Raspberry Pi, change to the gateway script directory with `cd /usr/lib/cgi-bin/gateway`

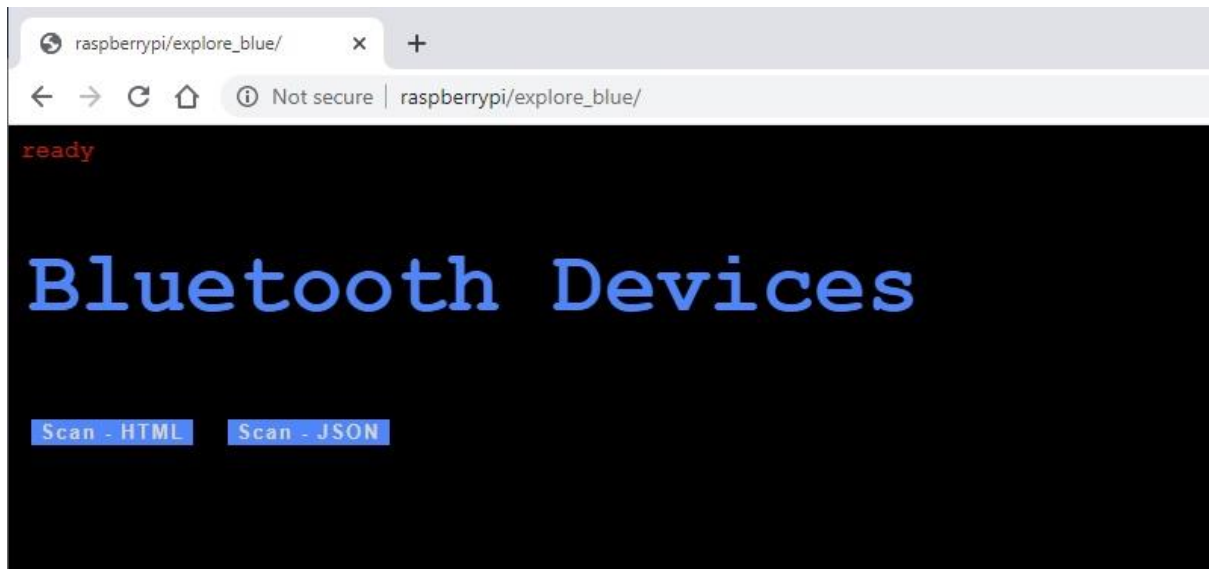
3. Launch the websocketd daemon with the command `websocketd --port=8888 /usr/lib/cgi-bin/gateway/do_notifications.py`

You should now be in a position to use the Explore Blue application.

### 5.2.2 Task - Experiment with Explore Blue

Power up your BBC micro:bit and any other devices you have available for testing.

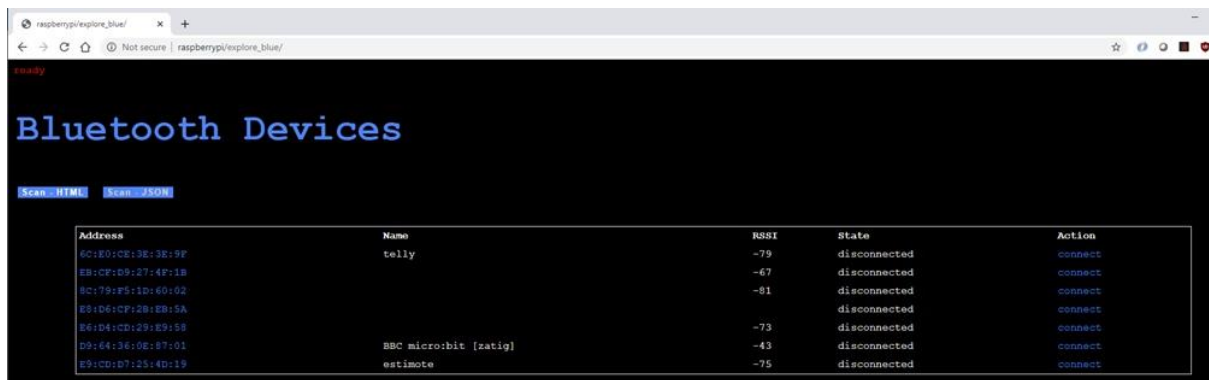
Go to [http://raspberrypi/explore\\_blue/](http://raspberrypi/explore_blue/) in your web browser to access the home page.



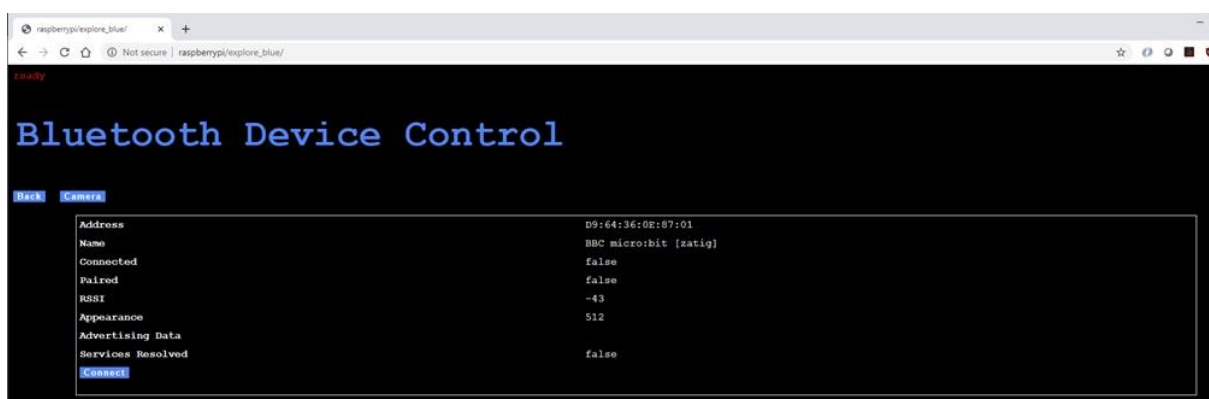
The two buttons, each initiate device discovery, scanning for three seconds. The *Scan - HTML* button returns a formatted, HTML list of the device discovered whereas the *Scan - JSON* button displays the JSON object returned by the gateway API. Click this button first and review the results.



Now click the *Scan - HTML* button.

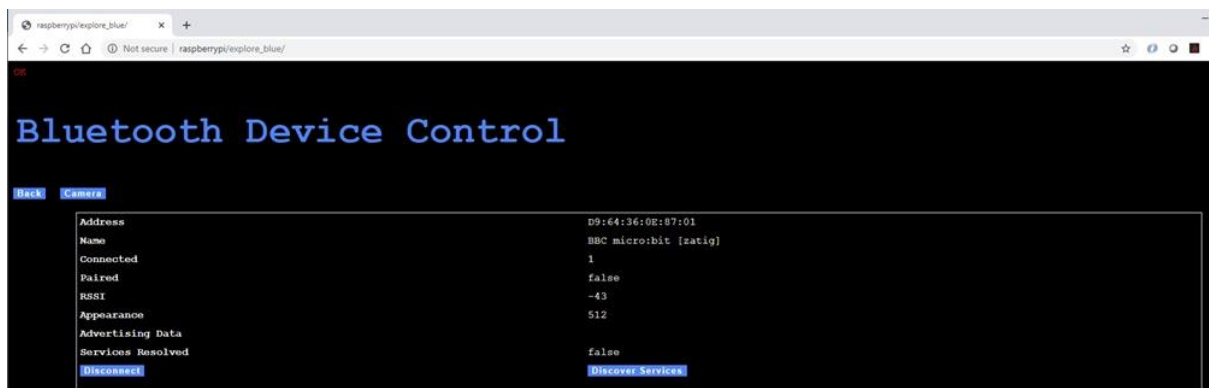


Click the device address link next to your BBC micro:bit.

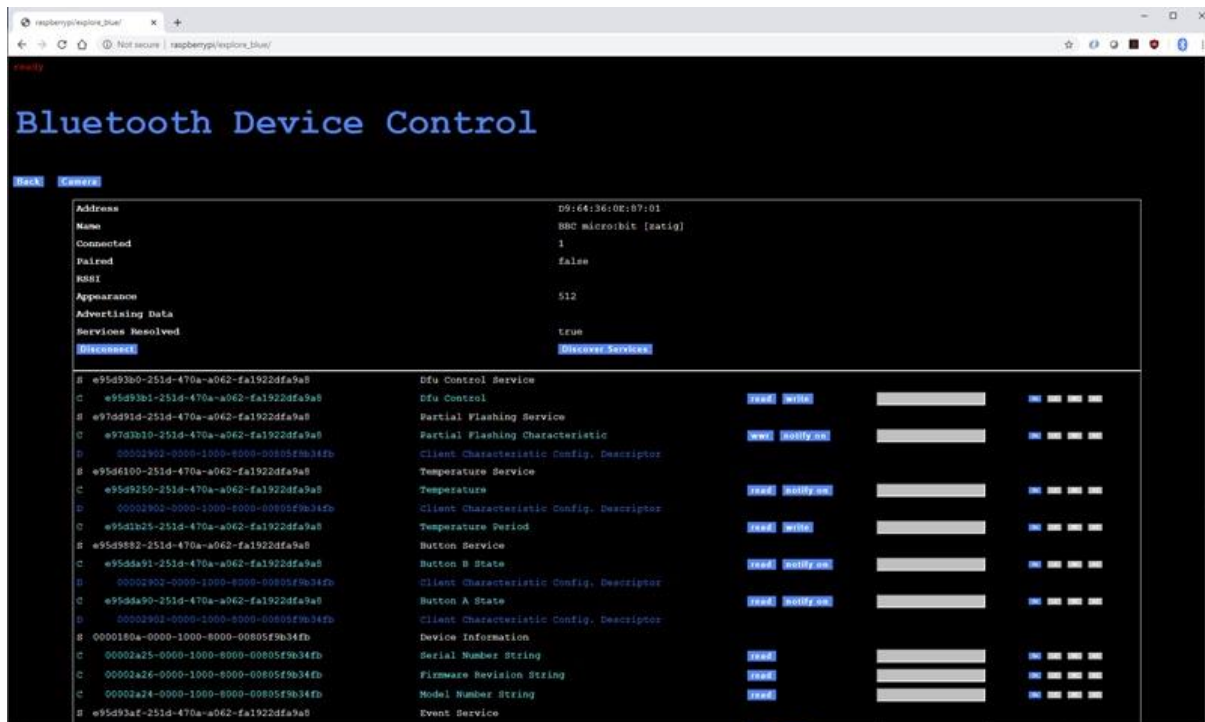


If the device is currently disconnected from the gateway, you will see the page as shown above, with a *Connect* button available. If it has already been connected to, you will see the next version of this page.

If necessary, click the *Connect* button to have the gateway connect to the micro:bit.

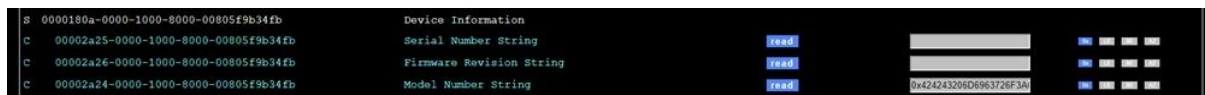


Next, click the *Discover Services* button. This will produce a formatted list of services, characteristics and descriptors with their UUIDs, names (if known) and buttons relating to the operations supported, as defined by the attribute's GATT properties. On the right-hand side, there's a field for displaying (not entering) values and four buttons which allow the raw byte array value of a characteristic to be interpreted for display purposes as either a hex value, an integer in little endian format, an integer in big endian format or as a text string.

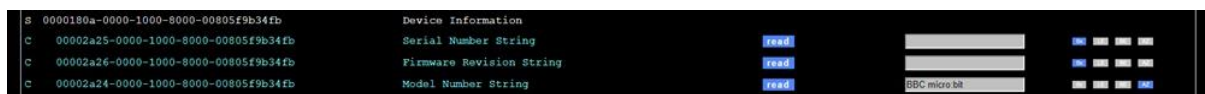


Note that the names of attributes are determined by looking for the UUID in a JavaScript Map object defined in the file `js/uuids.js`. Feel free to add your own values for devices you are using. Remember to clear your browser cache and reload the Explore Blue page if you do make changes.

Find the Device Information Service and the Model Number String characteristic which it contains. Click the *read* button next to this characteristic. The value of the characteristic will be read from the remote device and displayed in hex format in the value field.



Now click on the Az button next to the value. This will cause the value to be interpreted and displayed as a text string. It should say “BBC micro:bit”.



Next, write a short text string to the LED Text characteristic, which belongs to the LED Service. Clicking on the *write* button next to this characteristic will cause an input form to pop up. Type “Hello” into the value field and select Text from the drop down list of data formats and then click submit.



You should see your input text scroll across the display of your micro:bit (provided you are running the code supplied with this resource).

Finally, find the Temperature Service. There's a characteristic called Temperature which supports notifications. Click the *notify on* button to enable them. You will start to see temperature values appearing in the value field, in hex format. Click the *LE* or *BE* button to convert this to a decimal value which is in Celsius. Note that it doesn't matter whether you choose LE or BE because this is a single byte field. Also note that the temperature returned by the micro:bit is derived from the CPU temperature and so may not be a particularly accurate reflection of the ambient temperature.



If everything worked as expected, you have a working gateway application with which to monitor and control all manner of Bluetooth LE device via your gateway. Explore, enjoy and learn!

## 5.3 Task - Develop the Micro:bit Controller Application

### 5.3.1 Overview

You will now have the opportunity to complete the development of a simple web application which you can use with a BBC micro:bit. Your application will

1. Allow you to discover, select and connect to your micro:bit.
2. Automatically perform service discovery after being connected to.
3. Read and display the serial number and firmware revision string.
4. Provide a user interface which allows the user to:
  - a. control the LED pattern displayed on the micro:bit
  - b. send text for display on the micro:bit LED matrix
  - c. display a graph of real time temperature values
5. Optionally, stream video from a webcam which is pointing at the micro:bit so a remote user can see the results of their actions.

### 5.3.2 Preparation

#### 5.3.2.1 Task - Install the motion daemon

Ideally, you should plug a webcam into one of the USB ports of your Raspberry Pi. You can then point it at the micro:bit to be controlled and watch it respond from within the gateway application. This is an optional step but recommended.

To support the streaming of images from a webcam to a web page, we need to install some software called *motion*. See <https://motion-project.github.io/index.html> for further details about *motion*.

To install *motion*, follow execute the command `sudo apt install motion`.

Edit the configuration file `/etc/motion/motion.conf`, find the following properties and set them to the values indicated. If a property is not present in the default file, add it.

```
framerate 50
stream_maxrate = 25
daemon on
stream_quality 100
stream_localhost off
```

Do not yet attempt to run *motion*. We want to run it as a non-root user for security reasons and a few additional steps are required to achieve this.

Edit the file `/etc/default/motion` and set the property `start_motion_daemon=yes`

Run the command `sudo systemctl daemon-reload`

Run the command `sudo /etc/init.d/motion restart`

Yes, use the **restart** argument. This is correct.

Check the status of motion with:

```
pi@raspberrypi:~ $ sudo /etc/init.d/motion status
● motion.service - LSB: Start Motion detection
   Loaded: loaded (/etc/init.d/motion; generated)
   Active: active (running) since Thu 2021-06-24 07:30:49 BST; 2s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 3941 ExecStart=/etc/init.d/motion start (code=exited, status=0/SUCCESS)
    Tasks: 3 (limit: 4915)
   CGroup: /system.slice/motion.service
           └─3969 /usr/bin/motion

pi@raspberrypi:~ $ ps -ef|grep motion
motion  4030      1  10  07:33 ?        00:00:07 /usr/bin/motion
pi       4036    1090  0  07:34 pts/0    00:00:00 grep --color=auto motion
```

Note that the `/usr/bin/motion` binary is running in a process owned by user *motion*. This is what we wanted.

#### 5.3.2.2 Task - Run the *websocketd* daemon

If it's not already running, start the *websocketd* daemon with the commands:

```
cd /usr/lib/cgi-bin/gateway
websocketd --port=8082 /usr/lib/cgi-bin/gateway/do_notifications.py
```

#### 5.3.2.3 Task - Install the *incomplete* application

The study guide package includes an incomplete version of the micro:bit controller application. Create the directory `/var/www/html/microbit` on your Raspberry Pi and then copy the contents of the study guide's `implementation\start_state\peripherals\microbit_controller` directory into it.

#### 5.3.2.4 Task - Application Orientation

When finished, the micro:bit controller application will look like the screenshot shown in Figure 5.



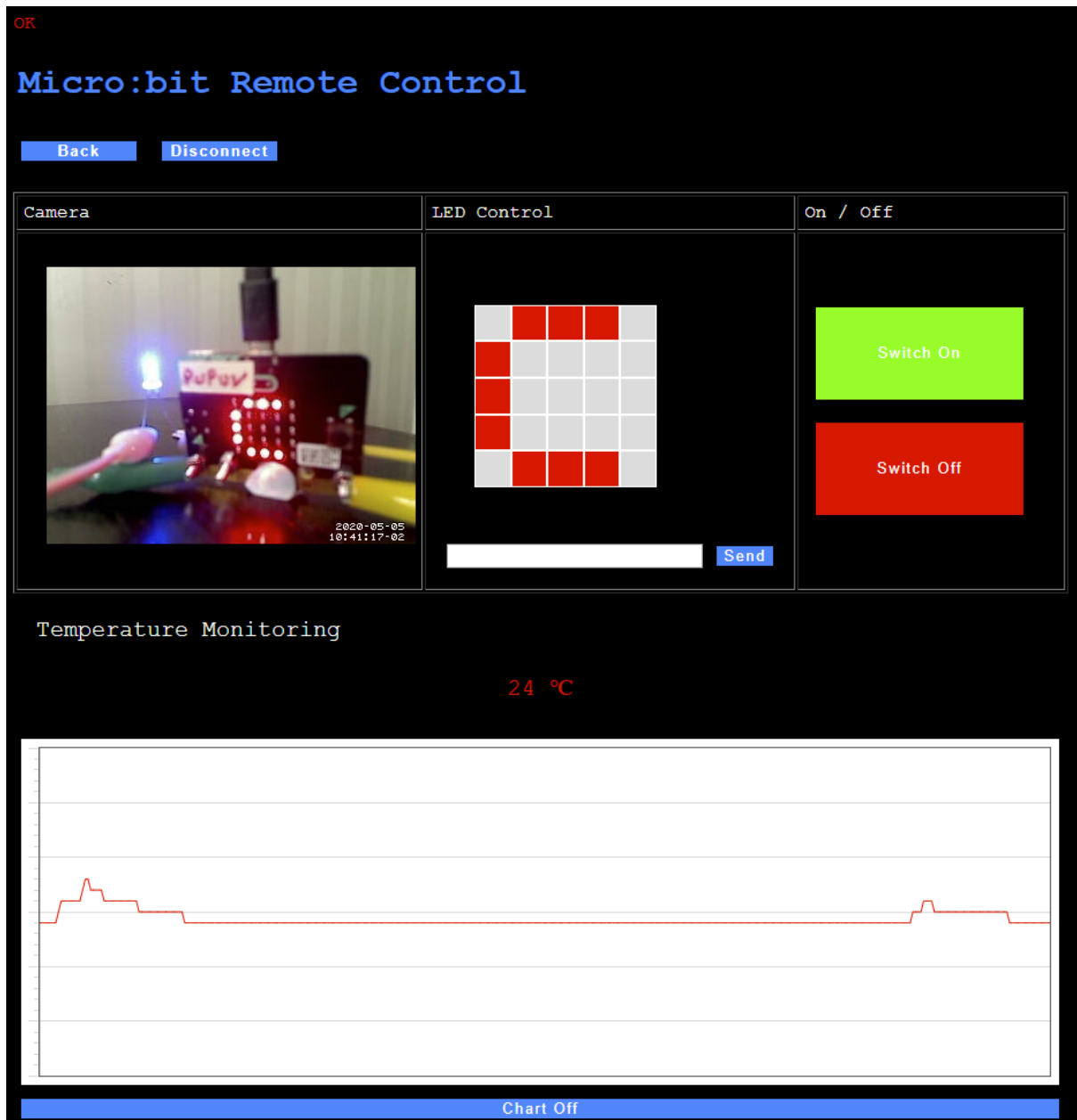


Figure 5 - The micro:bit controller gateway application

As you can see, it displays a live video feed from the webcam and has a number of functions which use the gateway to exchange data with the micro:bit over Bluetooth.

In the LED Control section, by clicking on the 5 x 5 grid of cells, the corresponding LED lights on the front of the micro:bit can be switched on and off. Text may be entered into the input field and sent, causing the message to scroll across the micro:bit display.

The on / off section includes two buttons. Clicking them will cause something that supports digital I/O and that is connected to GND and Pin 0 of the micro:bit to switch on or off. In the screenshot, you can see an LED, connected with the micro:bit with a resistor (not shown) in series.

At the bottom of the page, the Temperature Monitoring section allows temperature data notifications to be collected from the micro:bit. The latest temperature value is displayed numerically. All values up to a limit of 3600 (one reading per second for one hour) are plotted in a line graph.

The starter version of the application which you have installed, contains everything it needs except for some of the functions which invoke gateway API operations. Your next task will be to complete the application by implementing these functions.

The functions which handle device discovery are complete. Review the function *get\_devices\_json* to see how this works. In addition, service discovery is also complete. See *doServiceDiscovery* to see what's involved and in particular, note the way characteristics of interest are found and their representative objects stored for later use. You will be using these objects in your code.

#### 5.3.2.5 Task - Invoking HTTP requests from JavaScript

Micro:bit Controller is a web application. Consequently, it consists of HTML, CSS and JavaScript. JavaScript is used to write functions which provide most of the dynamic aspects of the application and it is in JavaScript that gateway related functions must be written.

JavaScript includes the XMLHttpRequest API which allows HTTP requests to be sent and the response to be received asynchronously and handled.

Study the following examples, which show how to use JavaScript to initiate HTTP GET and HTTP PUT operations. In each case, a JSON object is returned as the response. GET operations pass parameters in a URL query string. PUT operations pass parameters as a JSON object in the HTTP request entity body.

Figure 6 illustrates how XMLHttpRequest is used to formulate and send an HTTP GET request with a query string used to encode parameters.

```
function httpGet() {  
  
    var xhttp = new XMLHttpRequest();  
  
    // this is a callback function which is called when the request state changes  
    xhttp.onreadystatechange = function() {  
  
        // this means the request has completed, a response received and the HTTP status  
        // of the request is OK (status == 200)  
  
        if (this.readyState == 4 && this.status == 200) {  
            result = JSON.parse(this.responseText);  
            if (result.result == 0) {  
                // it worked! Perhaps the JSON object contains other data?  
            }  
        }  
    };  
  
    var target = "do_something.py";  
    // configure the request  
    xhttp.open("GET", CGI_ROOT+target+"?paramA="+data_1+"&paramB="+data_2, true);  
  
    // send it  
    xhttp.send();  
}
```

Figure 6 - HTTP GET with query string from JavaScript

Figure 7 shows how to send an HTTP PUT request, with parameters passed as a JSON object.

```
function httpPut() {  
  
    var xhttp = new XMLHttpRequest();  
  
    // this is a callback function which is called when the request state changes  
    xhttp.onreadystatechange = function() {  
  
        // this means the request has completed, a response received and the HTTP status
```

```

        // of the request is OK (status == 200)

        if (this.readyState == 4 && this.status == 200) {
            result = JSON.parse(this.responseText);
            if (result.result == 0) {
                // it worked! Perhaps the JSON object contains other data?
            }
        }
    };

    var args = {};
    args.paramA = data_1;
    args.paramB = data_2;
    var json = JSON.stringify(args);

    var target = "do_something.py";
    // configure the request
    xhttp.open("PUT", CGI_ROOT+target, true);

    // set a HTTP header to indicate that the body contains a JSON object
    xhttp.setRequestHeader('Content-type','application/json; charset=utf-8');

    // send it with the JSON data in the entity body of the request
    xhttp.send(json);
}

```

**Figure 7 - HTTP PUT with JSON parameters from JavaScript**

GET and PUT are very similar, as you can see.

### 5.3.3 Task - Completing the micro:bit controller application

Open the file `js/gateway.js` in a text editor. This file contains all of the functions which the micro:bit controller application uses to interact with the gateway. The only problem is, most of these functions are currently empty placeholders. Your task is to complete these functions.

#### 5.3.3.1 Connecting and Disconnecting

This is handled by a single function called *toggleConnectionState*. Find it in your editor.

Most of this function is already in place. Your task is to formulate the parameters, set up the HTTP request and send it. This request changes the state of the remote device from connected to disconnected or from disconnected to connected and therefore must use the HTTP PUT method.

A single parameter called *bdaddr* is required. This parameter must be set to the *bdaddr* property of the selected device. There's a JavaScript object called *selected\_device* which contains properties of the device which was clicked on in the scan results table and so, the value required is in *selected\_device.bdaddr*.

*selected\_device* also has a property called *connected*. If this is currently *true* then your code must send a disconnection request, with the target script named *do\_disconnect.py*. Otherwise it must target the script called *do\_connect.py*.

Using the HTTP examples in 5.3.2.5, try to complete this function.

Test the ability to connect and disconnect from your micro:bit. If you get stuck, look at the corresponding code in `implementation\solutions\peripherals\microbit_controller\js\gateway.js`.

#### 5.3.3.2 Reading the LED matrix state

Find the function named *readLedMatrixState* in your editor. This function will be automatically called when connected to the micro:bit. Its purpose is to read the LED Matrix State characteristic, which returns a 5-byte array of values. Each byte uses the least significant 5 bits to indicate the

on/off state of each of the 5 LEDs in one row. The first byte in the array represents the first row (counting from the top) of the LED grid, and so on.

Complete the gateway API request, which should appear between the two TODO comments. It must use HTTP GET and pass the `bdaddr` property of the selected device and the `handle` property of the `led_matrix_state_characteristic` object as parameters. This time you must pass arguments in a query string.

Using the HTTP examples in 5.3.2.5, try to complete this function.

Reset your micro:bit. Discover and select it in the device discovery screen. Connect to it by pressing the Connect button. This should cause a “C” to appear on the micro:bit display and your completed `readLedMatrixState` function should retrieve the LED Matrix State characteristic value and, using functions already in the code, decode it and update the GUI to reflect this.

If you get stuck, look at the corresponding code in `implementation\peripherals\solutions\microbit_controller\js\gateway.js`.

#### 5.3.3.3 Writing the LED matrix state

The state of the local, graphical representation of the LED matrix is held in a JavaScript array of 5 bytes called `led_rows`. Clicking on cells in the LED matrix user interface (UI) will change the values in this array. Each time this happens, the associated LED Matrix State characteristic needs to be sent the new value so that the physical LED grid on the micro:bit is updated to match the pattern on the UI.

This will happen in the currently incomplete function, `onUpdateLedMatrix()`. Find this function and complete the code required to send an HTTP PUT containing the necessary parameters, to the gateway script `do_write_characteristic.py`. Note that the value field is hex encoded.

To test, connect to your micro:bit and click the cells in the LED grid on the UI.

If you get stuck, look at the corresponding code in `implementation\peripherals\solutions\microbit_controller\js\gateway.js`.

#### 5.3.3.4 Temperature Notifications

Enabling, disabling and handling temperature notifications takes place in the function `toggleNotifications()`. The gateway API uses a web socket connection and JSON objects for requests and responses rather than HTTP requests.

Find this function in your code and review the code. You should note that

1. A Web socket is created and uses a URL which is in a constant defined in the `constants.js` file
2. The web socket has three callback functions names `onerror`, `onopen` and `onmessage`.
3. A new web socket is created only when the current notifying state of the temperature characteristic object is false. Otherwise, the web socket object, stored when originally created, is restored for use when disabling notifications.
4. When a new web socket is opened, we need to send a JSON object, per our API design which will result in notifications being enabled on the temperature characteristic in the remote device.
5. When this function is called and the characteristic is already notifying, we need to send a JSON object, per our API design which will result in notifications being disabled.

6. The `onmessage` callback function gets called whenever the gateway sends a notification object. The string representation of the JSON object is turned into an actual JSON object in the line `result = JSON.parse(event.data)`. After that, if the object has a *value* property, we conclude that this is a characteristic notification and everything that follows concerns the user interface in one way or another.

To gain some experience using the notifications API over web sockets, two parts of the function are incomplete. Find the TO DO comments and complete the missing code, in the first instance to send a JSON control object to enable notifications and in the second, to disable them.

If you get stuck, look at the corresponding code in  
`implementation\peripherals\solutions\microbit_controller\js\gateway.js`.

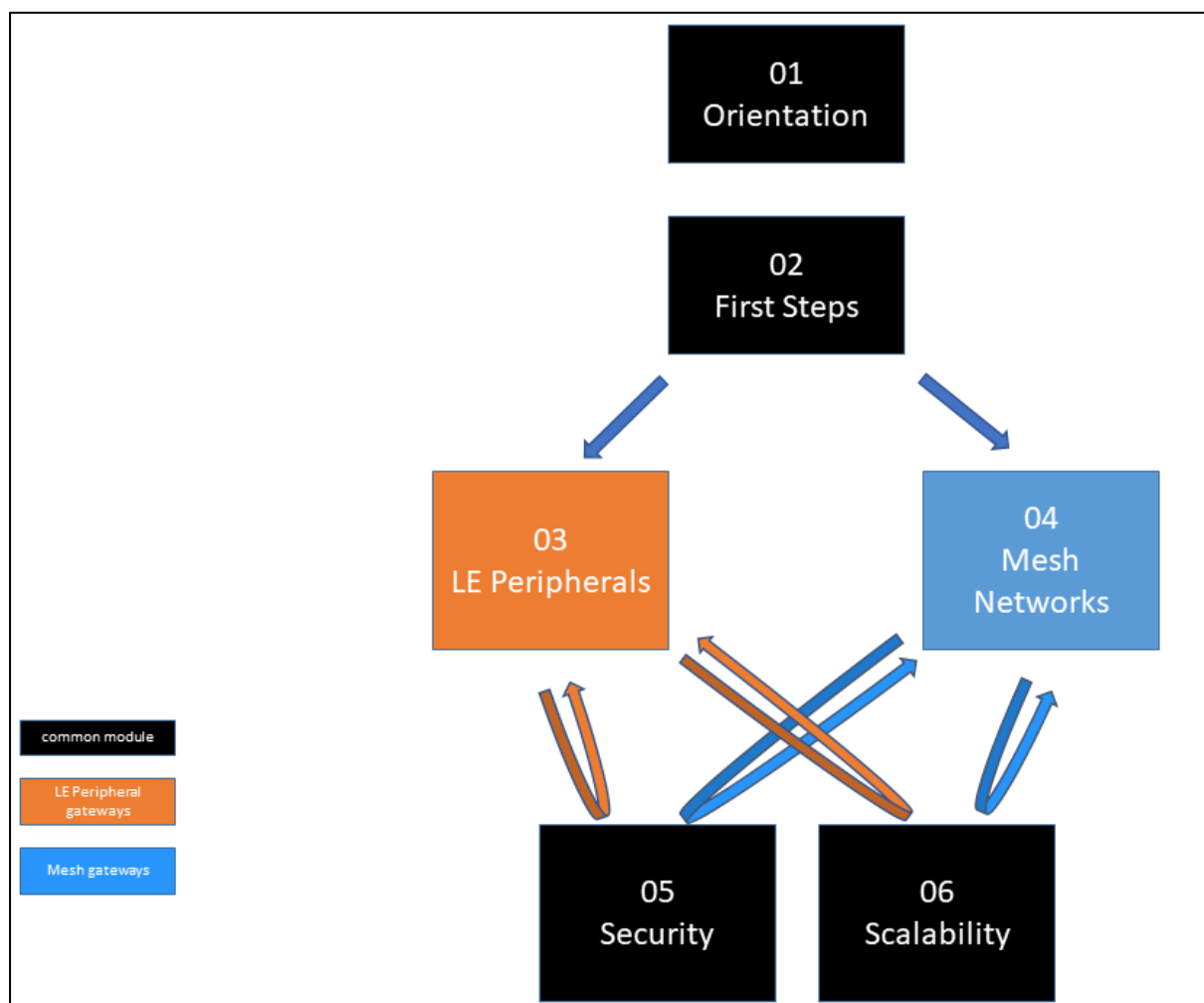
## 6. Security

### 6.1 Warning!

At this point, you should have a working gateway and two working gateway applications. However, you should note the following:

*The current implementation is insecure and should not be connected to the internet or other insecure networks!*

It's insecure because we have not yet turned our attention to the subject of security. Module 05 covers security for both LE Peripheral gateways and gateways for Bluetooth mesh. Proceed now to study module 05 and start to secure your LE Peripheral gateway. Return here when you have completed module 05 and then complete those remaining steps that apply specifically to the gateway for LE Peripherals.



## 6.2 Further Security Measures for the LE Peripherals Gateway

Module 05 identified generally applicable actions to take in securing a Bluetooth internet gateway of either of the two types dealt with in this study guide. The table of issues and countermeasures from section 3.1.4 of module 5 is repeated here but highlights any special actions required to secure the gateway for LE Peripherals.

Ref	Short Name	LE Peripheral Gateway Countermeasure
1	No user authentication or access control	The countermeasures from module 05 apply.
2	TCP/IP port visibility	The countermeasures from module 05 apply.
3	Data over TCP/IP is not private	The countermeasures from module 05 apply.
4	Multiple TCP/IP ports in use	The countermeasures from module 05 apply.
5	websocketd is directly accessible	The countermeasures from module 05 apply.
6	motion is directly accessible	The countermeasures from module 05 apply.
7	Bluetooth data in flight is not necessarily private	For Bluetooth data in flight between the gateway and Bluetooth device to be private, Bluetooth LE Peripheral devices must be paired with the gateway. It should be possible to configure the gateway to only allow access to paired devices if required.
8	Not all Bluetooth devices should be accessible	The general countermeasures from module 05 apply but implementation details will vary according to the type of gateway. As such, a Bluetooth firewall which controls access to LE Peripheral devices and their capabilities will be designed and built in this module's project.
9	Not all Bluetooth device capabilities should be available	In the case of LE Peripheral devices, the Bluetooth firewall will allow access to specific GATT attributes identified by UUID, to be blocked.
10	Physical access	The countermeasures from module 05 apply.
11	Gateway server login control	The countermeasures from module 05 apply.
12	User access rights	The countermeasures from module 05 apply.

Continue below to implement actions which are needed to secure the LE Peripheral gateway.

## 6.3 Encrypted web sockets communication

Create a script with which to start websocketd like this:

```
pi@raspberrypi:~ $ vi start_secure_ws.sh
#!/bin/bash
cd /usr/lib/cgi-bin/gateway
websocketd --address=raspberrypi --port=8082 --ssl --sslcert=/etc/apache2/ssl/apache.crt --
sslkey=/etc/apache2/ssl/apache.key /usr/lib/cgi-bin/gateway/do_notifications.py
```

Save the file and change its permissions with `chmod 755 start_secure_ws.sh`

The arguments to websocketd cause communication to be encrypted with SSL and port 8082 to be used. We'll adjust gateway applications to use encrypted web sockets communication later.

## 6.4 Authentication

Create user credentials as described in module 05 section 3.2.2.3 and then continue here.

Secure each web application directory with a <Directory> entry in the virtual host config which we have enabled for SSL use:

```
sudo vi /etc/apache2/sites-enabled/000-default-ssl.conf

# inside the configuration block which starts with...
<IfModule mod_ssl.c>
    <VirtualHost _default_:443>

# add these two Directory entries

        <Directory /var/www/html/explore_blue/>
            # Choose authentication protocol
            AuthType Basic

            # Define the security realm
            AuthName "Explore Blue"

            # Location of the user password file
            AuthUserFile /etc/apache2/auth.users

            # Valid users can access this folder and no one else
            Require valid-user
        </Directory>

        <Directory /var/www/html/microbit/>
            # Choose authentication protocol
            AuthType Basic

            # Define the security realm
            AuthName "Microbit Controller"

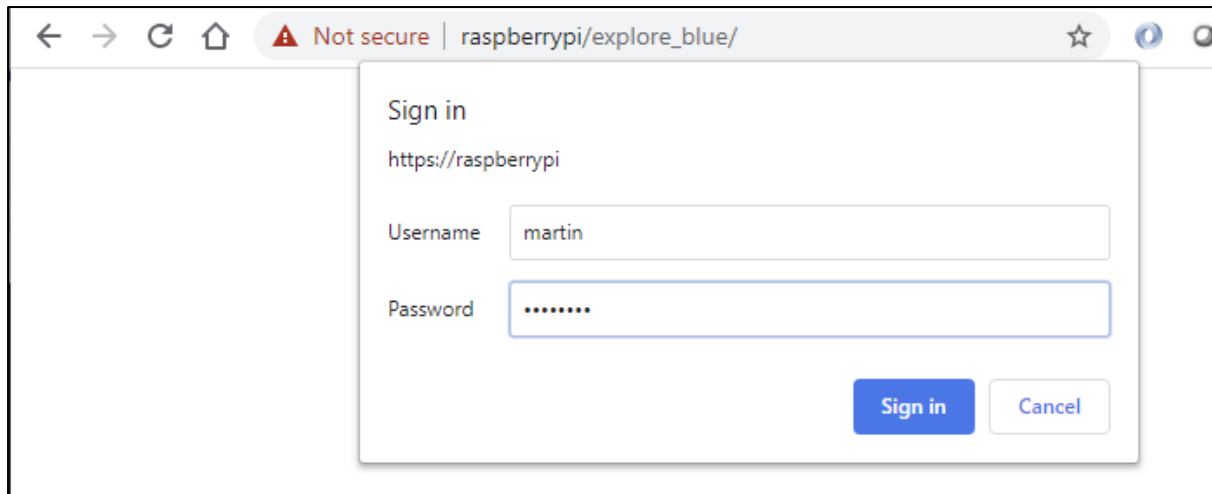
            # Location of the user password file
            AuthUserFile /etc/apache2/auth.users

            # Valid users can access this folder and no one else
            Require valid-user
        </Directory>
```

Restart apache with `sudo service apache2 restart`

Try to access the two gateway applications at [https://raspberrypi/explore\\_blue](https://raspberrypi/explore_blue) and <https://raspberrypi/microbit>. You will now be challenged to enter a valid user ID and password.





## 6.5 Reverse Proxying and the LE Peripherals Gateway

### 6.5.1 Web server configuration changes

After taking the generally applicable actions defined in module 05, complete the steps required to use reverse proxying for the LE Peripherals gateway as follows:

Edit `/etc/apache2/sites-enabled/000-default-ssl.conf` and add or set the following properties:

```
SSLProxyEngine on
ProxyRequests Off
ProxyPass "/ws/" "wss://raspberrypi:8082/"
ProxyPass "/camera/" "http://raspberrypi:8081/"
```

Add the following Location blocks, which will ensure that any requests which will be forwarded to the web sockets daemon or to the webcam *motion* daemon are also subject to authentication.

```
<Location "/ws/">
    AuthType Basic
    AuthName "web sockets"
    AuthUserFile /etc/apache2/auth.users
    Require valid-user
</Location>

<Location "/camera/">
    AuthType Basic
    AuthName "webcam authentication"
    AuthUserFile /etc/apache2/auth.users
    Require valid-user
</Location>
```

Restart the apache web server.

### 6.5.2 Adjusting the gateway applications

The gateway applications each need two small changes. The URL to be used for web sockets must use the secure protocol scheme “wss://” rather than “ws://” and instead of addressing the websocketd port directly, should now use a URL which will be matched by the reverse proxy configuration and forwarded.

Edit `constants.js` in each gateway applications and make the required change as shown here:

```
//let NOTIFICATIONS_SERVER = 'ws://raspberrypi:8082';  
let NOTIFICATIONS_SERVER = 'wss://raspberrypi:443/ws/';
```

In addition, the URL used for streaming images from a connected webcam must be changed. In the HTML files for the relevant gateway applications (/var/www/html/explore\_blue/camera.html, /var/www/html/microbit/index.html and /var/www/html/mesh/index.html) change the camera img URL to

```
<img src= "../camera/?action=stream" />
```

Clear your browser cache, reload the application web pages and the websocket communication should now be secured and the webcam stream should now work via the reverse proxy.

## 6.6 Task - Securing the Bluetooth Devices

To secure Bluetooth LE Peripheral devices, we need to address two sets of issues. To ensure data over the air is encrypted, devices must be paired with the gateway. In addition, we need a component which was referred to as a *Bluetooth firewall* in module 05 to restrict access to selected device only and to block access to specified GATT attributes.

### 6.5.1 Pairing with the gateway

Our gateway uses BlueZ and so pairing is to be achieved by the gateway administrator using the command line tool *bluetoothctl*.

You must know the device address of your micro:bit to be able to pair it. Find it with the gateway explore\_blue application and note it down.

Install the binary file microbit-example\_device\_pairing\_required.hex from the study guide directory implementation\solutions\peripherals\microbit\bin onto your micro:bit. To access the GATT attributes on this device, the gateway must first be paired with the micro:bit.

Follow these steps to accomplish this:

```
pi@raspberrypi:~ $ sudo service bluetooth restart  
pi@raspberrypi:~ $ bluetoothctl  
Agent registered  
  
# at this point, put the micro:bit into pairing mode by holding down buttons A and B at the  
# same  
# time and pressing and releasing the reset button. Keep buttons A and B pressed while the  
# micro:bit resets.  
  
[bluetooth]# scan on  
Discovery started  
[CHG] Controller DC:A6:32:0E:83:E2 Discovering: yes  
[CHG] Device D9:64:36:0E:87:01 RSSI: -52  
[NEW] Device 8C:79:F5:1D:60:02 8C-79-F5-1D-60-02  
[NEW] Device 48:9A:53:FA:11:E6 48-9A-53-FA-11-E6  
[NEW] Device EB:CF:D9:27:4F:1B EB-CF-D9-27-4F-1B  
[NEW] Device E6:D4:CD:29:E9:58 E6-D4-CD-29-E9-58  
[CHG] Device E9:CD:D7:25:4D:19 RSSI: -77  
[CHG] Device E9:CD:D7:25:4D:19 ServiceData Key: 0000fe9a-0000-1000-8000-00805f9b34fb  
[CHG] Device E9:CD:D7:25:4D:19 ServiceData Value:  
00 c3 fe 72 23 c0 19 33 64 fa a7 b1 2f 74 d8 1a ...r#...3d.../t..  
14 91 44 40 ..De  
[NEW] Device E8:D6:CF:2B:EB:5A E8-D6-CF-2B-EB-5A  
[NEW] Device 62:67:C6:4B:D8:7C 62-67-C6-4B-D8-7C  
[NEW] Device 94:65:2D:A8:F9:19 My Phone  
[CHG] Device E6:D4:CD:29:E9:58 RSSI: -79  
[bluetooth]# scan off  
[CHG] Device 94:65:2D:A8:F9:19 RSSI is nil
```

```
[CHG] Device 62:67:C6:4B:D8:7C TxPower is nil
[CHG] Device 62:67:C6:4B:D8:7C RSSI is nil
[CHG] Device E8:D6:CF:2B:EB:5A RSSI is nil
[CHG] Device E9:CD:D7:25:4D:19 RSSI is nil
[CHG] Device E6:D4:CD:29:E9:58 RSSI is nil
[CHG] Device EB:CF:D9:27:4F:1B RSSI is nil
[CHG] Device 48:9A:53:FA:11:E6 TxPower is nil
[CHG] Device 48:9A:53:FA:11:E6 RSSI is nil
[CHG] Device 8C:79:F5:1D:60:02 RSSI is nil
[CHG] Device D9:64:36:0E:87:01 RSSI is nil
[CHG] Controller DC:A6:32:0E:83:E2 Discovering: no
Discovery stopped
[bluetooth]# pair D9:64:36:0E:87:01
Attempting to pair with D9:64:36:0E:87:01
[CHG] Device D9:64:36:0E:87:01 Connected: yes
[CHG] Device D9:64:36:0E:87:01 UUIDs: 00001800-0000-1000-8000-00805f9b34fb
[CHG] Device D9:64:36:0E:87:01 UUIDs: 00001801-0000-1000-8000-00805f9b34fb
[CHG] Device D9:64:36:0E:87:01 UUIDs: 0000180a-0000-1000-8000-00805f9b34fb
[CHG] Device D9:64:36:0E:87:01 UUIDs: e95d93b0-251d-470a-a062-fa1922dfa9a8
[CHG] Device D9:64:36:0E:87:01 UUIDs: e97dd91d-251d-470a-a062-fa1922dfa9a8
[CHG] Device D9:64:36:0E:87:01 Connected: no
[CHG] Device D9:64:36:0E:87:01 Paired: yes
Pairing successful
[bluetooth]# quit
```

## 6.5.2 The Bluetooth firewall

In the study guide directory

implementation\solutions\peripherals\gateway\_adapters\_with\_security, you'll find a Python module called bluetooth\_firewall.py, modified versions of the adapter scripts which use this module to apply various rules and a configuration file, config.json which provides the Bluetooth firewall with its configuration. Let's review that file now.

```
{
    "enabled": true,
    "paired_devices_only": false,
    "device_acceptlist": [ "D9:64:36:0E:87:01", "EF:4F:05:0C:29:EB"],
    "**": [{
        "service_uuid": "e95d93b0-251d-470a-a062-fa1922dfa9a8"
    },
    {
        "service_uuid": "e97dd91d-251d-470a-a062-fa1922dfa9a8"
    },
    {
        "service_uuid": "00001801-0000-1000-8000-00805f9b34fb"
    }
    ],
    "D9:64:36:0E:87:01": [
        {"characteristic_uuid": "e95d1b25-251d-470a-a062-fa1922dfa9a8"},
        {"characteristic_uuid": "e95d0d2d-251d-470a-a062-fa1922dfa9a8"},
        {"descriptor_uuid": "00002902-0000-1000-8000-00805f9b34fb"}
    ]
}
```

The properties of the config.json object are explained in table

Property	Explanation
----------	-------------

enabled	Allows the Bluetooth firewall to be easily enabled (true) or disabled (false).
paired_devices_only	If true then only devices which have been paired with the gateway will be available to gateway applications.
device_acceptlist	The addresses of devices which may be accessed by gateway applications are listed here.
*	This property represents all device addresses and its value consists of a list of GATT service UUIDs which are never to be available for any operation (including discovery). If a service is reject-listed then all of its characteristics and descriptors are also reject-listed.
[device address]	This property reject-lists specified GATT attributes (in this case, two characteristics and a descriptor) with respect to this particular device only.

As supplied, config.json allows only two devices to be accessed. Specific services and characteristics are blocked.

UUID	GATT Attribute
e95d93b0-251d-470a-a062-fa1922dfa9a8	DFU Control Service (DFU = Device Firmware Update)
e97dd91d-251d-470a-a062-fa1922dfa9a8	Partial Flashing Service
00001801-0000-1000-8000-00805f9b34fb	Generic Attribute Service
00002902-0000-1000-8000-00805f9b34fb	Client Characteristic Configuration Descriptor
e95d0d2d-251d-470a-a062-fa1922dfa9a8	Scrolling Delay characteristic
e95d1b25-251d-470a-a062-fa1922dfa9a8	Temperature Period characteristic

These configuration values are supplied as examples only. Your requirements will almost certainly vary.

Other configuration examples:

#### Allow all devices

```
"device_acceptlist": ["*"]
```

#### Block a specific service and all it contains for all devices

```
"*": [ {
    "service_uuid": "e95d93b0-251d-470a-a062-fa1922dfa9a8"
  },
  {
    "service_uuid": "e97dd91d-251d-470a-a062-fa1922dfa9a8"
  },
  {
    "service_uuid": "00001801-0000-1000-8000-00805f9b34fb"
  }
]
```

#### Block particular characteristics regardless of the owning service for a given device

```
"D9:64:36:0E:87:01": [{
  "characteristic_uuid": "e95d1b25-251d-470a-a062-fa1922dfa9a8",
  "characteristic_uuid": "e95d0d2d-251d-470a-a062-fa1922dfa9a8"
}]
```

#### Block a characteristic in a specific service on a specific device

```
"EF:4F:05:0C:29:EB": [
  {
    "service_uuid" : "00001802-0000-1000-8000-00805f9b34fb",
    "characteristic_uuid" : "00002A06-0000-1000-8000-00805f9b34fb"
  }
]
```

#### Block a descriptor in all cases

```
"*": [
  {
    "descriptor_uuid" : "00002902-0000-1000-8000-00805f9b34fb"
  }
]
```

#### Block a descriptor when owned by a particular characteristic

```
"*": [
  {
    "characteristic_uuid" : "e95dda90-251d-470a-a062-fa1922dfa9a8",
    "descriptor_uuid" : "00002902-0000-1000-8000-00805f9b34fb"
  }
]
```

The `bluetooth_firewall` Python module provides the following functions for use by adapter scripts:

Function	Description
<code>device_is_allowed(bdaddr)</code>	Returns true or false according to whether or not the configuration indicates that the device with this address should be accessible.

<code>characteristic_is_allowed(bdaddr, characteristic_handle)</code>	Indicates whether or not the characteristic with specified handle is accessible on the device with the specified address.
<code>descriptor_is_allowed(bdaddr, descriptor_handle)</code>	Indicates whether or not the descriptor with specified handle is accessible on the device with the specified address.
<code>filter_services(bdaddr, service_list)</code>	Removes services from the supplied list which are not to be accessible on this device.
<code>filter_characteristics(bdaddr, service_uuid, characteristics_list)</code>	Removes characteristics from the supplied list which are not to be accessible on this device.
<code>filter_descriptors(bdaddr, service_uuid, characteristic_uuid, descriptors_list)</code>	Removes descriptors from the supplied list which are not to be accessible on this device.

Take a look at the adapter scripts in the `implementation\solutions\peripherals\gateway_adapters_with_security` directory and study the way in which the `bluetooth_firewall` functions are used to control access to and visibility of devices, as controlled by the `config.json` file.

For example, `do_discover_devices.py` looks like this:

```
#!/usr/bin/python3
import json
import cgi
import bluetooth_firewall

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gap

print("Content-Type: application/json;charset=utf-8")
print()
querystring = cgi.FieldStorage()
scantime = querystring.getfirst("scantime", "2000")
devices_discovered = bluetooth_gap.discover_devices(int(scantime))
devices_allowed = bluetooth_firewall.filter_devices(devices_discovered)
devices_allowed_json = json.JSONEncoder().encode(devices_allowed)
print(devices_allowed_json)
```

And `do_write_characteristic.py` looks like this:

```
#!/usr/bin/python3
import os
import json
import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_constants
import bluetooth_utils
import bluetooth_firewall

if 'REQUEST_METHOD' in os.environ:
    result = {}
    args = json.load(sys.stdin)
    if os.environ['REQUEST_METHOD'] != 'PUT':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
```

```

        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        if not "bdaddr" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "handle" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        elif not "value" in args:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = args["bdaddr"]
            handle = args["handle"]
            value = args["value"]
            result = {}
            result['bdaddr'] = bdaddr
            result['handle'] = handle
            if not bluetooth_firewall.characteristic_is_allowed(bdaddr, handle):
                print('Status-Line: HTTP/1.0 403 Forbidden')
            else:
                try:
                    rc = bluetooth_gatt.write_characteristic(bdaddr, handle, value)
                    result['result'] = rc
                    print(json.JSONEncoder().encode(result))
                except bluetooth_exceptions.StateError as e:
                    result['result'] = e.args[0]
                    print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")

```

The service discovery script, `do_service_discovery.py` is the most complicated, having to filter GATT attributes at each level of the services/characteristics/descriptors hierarchy:

```

#!/usr/bin/python3
import os
import json
from sys import stdin, stdout
import cgi

import sys
sys.path.insert(0, '../bluetooth')
import bluetooth_gatt
import bluetooth_exceptions
import bluetooth_firewall

if 'REQUEST_METHOD' in os.environ:
    result = {}
    if os.environ['REQUEST_METHOD'] != 'GET':
        print('Status: 405 Method Not Allowed')
        print()
        print("Status-Line: HTTP/1.0 405 Method Not Allowed")
        print()
    else:
        print("Content-Type: application/json;charset=utf-8")
        print()
        querystring = cgi.FieldStorage()
        if not "bdaddr" in querystring:
            result['result'] = bluetooth_constants.RESULT_ERR_BAD_ARGS
            print(json.JSONEncoder().encode(result))
        else:
            bdaddr = querystring.getfirst("bdaddr")
            try:
                services_discovered = bluetooth_gatt.get_services(bdaddr)
                # firewall service filtering
                allowed_services = bluetooth_firewall.filter_services(bdaddr,
services_discovered)

                for service in allowed_services:
                    # rename BlueZ-specific parameter name to the more abstract 'handle'
                    service['handle'] = service.pop('path')

```

```

        characteristics_discovered = bluetooth_gatt.get_characteristics(bdaddr,
service['handle'])

        # firewall characteristic filtering
        allowed_characteristics =
bluetooth_firewall.filter_characteristics(bdaddr, service['UUID'],
characteristics_discovered)
        service['characteristics'] = allowed_characteristics
        for characteristic in allowed_characteristics:
            characteristic['handle'] = characteristic.pop('path')
            characteristic['service_handle'] =
characteristic.pop('service_path')
            descriptors_discovered = bluetooth_gatt.get_descriptors(bdaddr,
characteristic['handle'])

            # firewall descriptor filtering
            allowed_descriptors = bluetooth_firewall.filter_descriptors(bdaddr,
service['UUID'], characteristic['UUID'], descriptors_discovered)
            characteristic['descriptors'] = allowed_descriptors
            for descriptor in allowed_descriptors:
                descriptor['handle'] = descriptor.pop('path')
                descriptor['characteristic_handle'] =
descriptor.pop('characteristic_path')

            attributes_discovered_json = json.JSONEncoder().encode(allowed_services)
            print(attributes_discovered_json)
        except bluetooth_exceptions.StateError as e:
            result = {}
            result['result'] = e.args[0]
            print(json.JSONEncoder().encode(result))
    else:
        print("ERROR: Not called by HTTP")

```

Copy the contents of implementation\solutions\peripherals\gateway\_adapters\_with\_security into your /usr/lib/cgi-bin/gateway directory, overwriting the old versions of the adapter scripts.

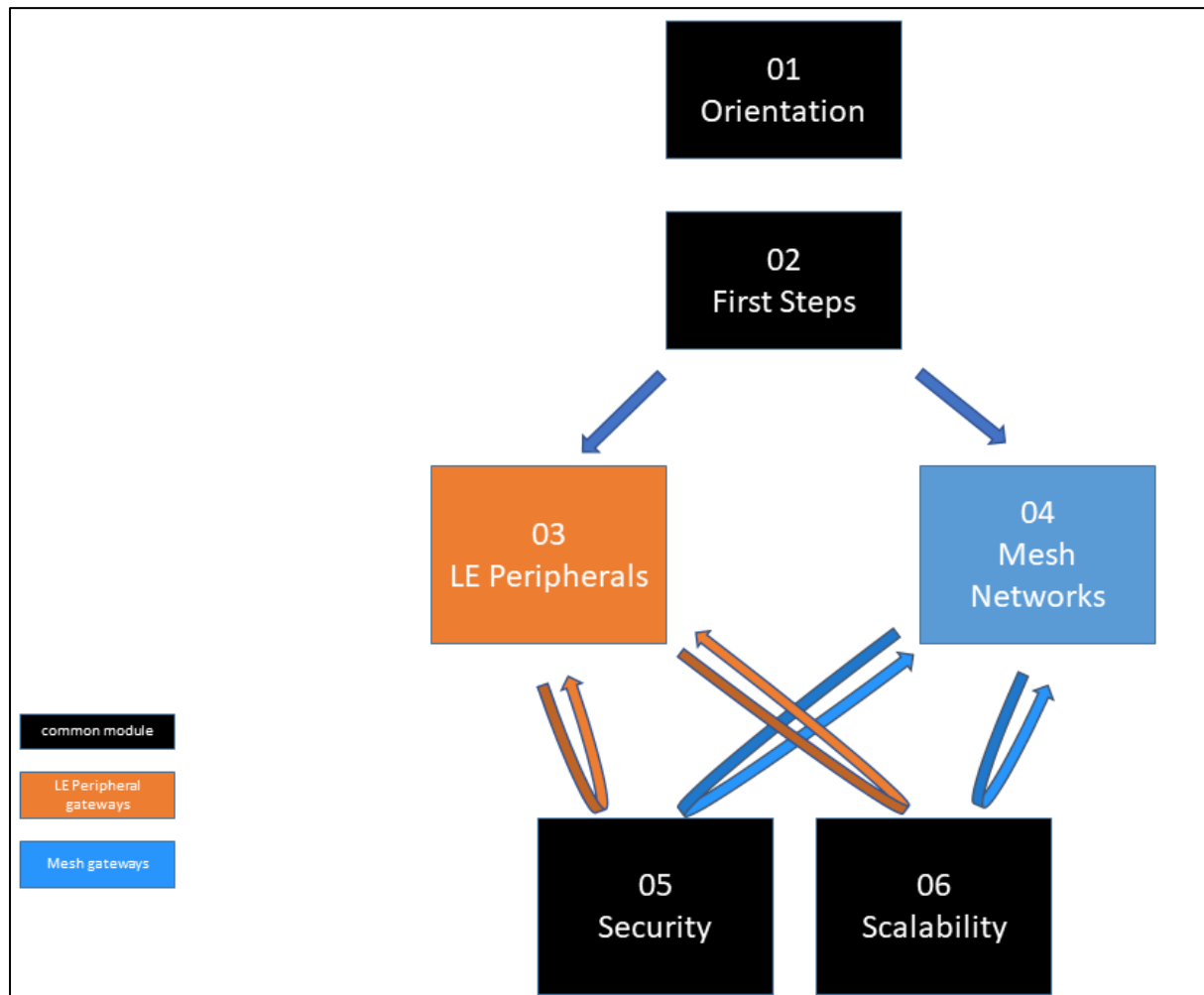
Edit config.json so that only one or two of your Bluetooth devices, including the micro:bit you have been using, are included in the device accept list. Exclude attribute UUIDs of your choice. Test Explore Blue and notice the impact your Bluetooth firewall has on the results.

**Comment:** The HTTP Status value 403 means Forbidden. This seems a reasonable status code to use when the firewall denies access. But is this wise from a security perspective? Informing the client that its request has been forbidden could be interpreted as informing the client that there is something of value being protected there and signal to a hacker that this is an area worthy of further work in pursuit of an exploit. This issue is called [information leakage](#).



## 7. Scalability

Module 06 covers the topic of scalability for both LE Peripheral gateways and gateways for Bluetooth mesh. Proceed now to study module 06.



## 8. Close

This brings us to the end of this module. It is hoped that you have learned all that you wanted to learn about Bluetooth internet gateways for LE Peripherals and feel ready to include Bluetooth devices in your Internet of Things (IoT) solutions.

If you are interested in Bluetooth internet gateways for Bluetooth mesh networks, proceed now to study module 04.