

Bluetooth Gateway Smart Starter Kit

Implement a Bluetooth Gateway using Node.js



Revision History

Version	Date	Description	Author
V1.0.1	2016-02-19	hapi in navible is v9.0.3.	KR
v.1.02	2016-02-24	Moved the solution installation to a new doc.	VG

Contents

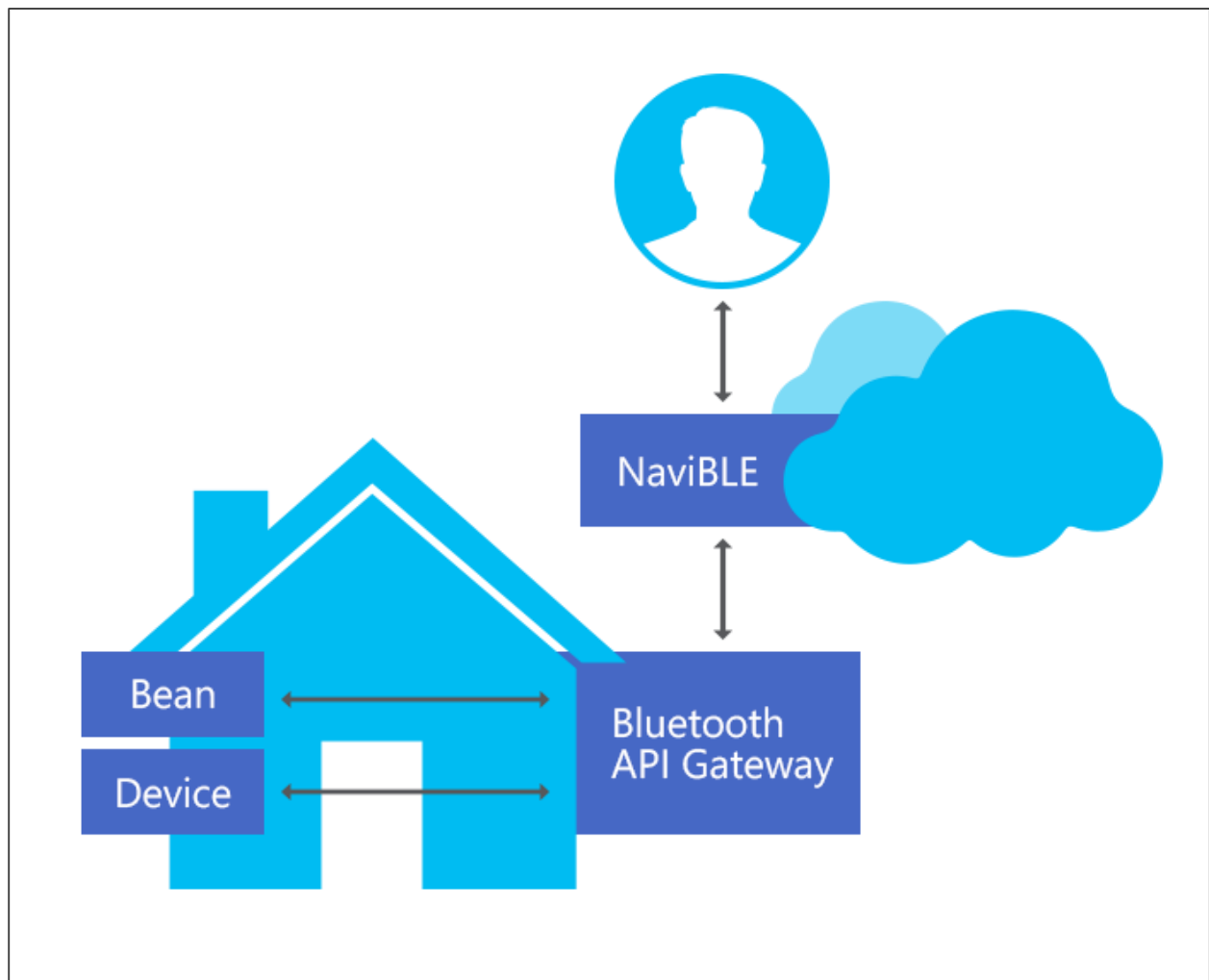
1 Overview.....	5
1.1 Prerequisites for this lab.....	6
2 Install and start the Gateway server.....	9
3 Setting up NaviBLE	15
3.1 Create a node app	18
3.2 [Optional] Install hapi.js	19
3.3 [Optional] Install other tools	20
3.4 Create a hapi server.....	22
3.5 Set up routes for the server	23
3.6 Create the /routes/index.js file	27
3.7 Create the nodes/index view	28
3.8 Create a stylesheet	31
4 Using Gateway server to explore the attributes of the target devices.....	36
5 Using NaviBLE to handle the requests and UX.	37
5.1 Define a route for getting services from a node	37
5.2 Implement the route using the Bluetooth Gateway	37
5.3 Create a view to display the results	38
6 Using NaviBLE to list characteristics of each service	41
6.1 Define a route for getting characteristics of a service	41

6.2	Implement the route using the Bluetooth Gateway	41
6.3	Create a view to display the results	43
7	Getting and setting values for a characteristic	46
7.1	Define routes and add them to the server.....	46
7.2	Implement the route using the Bluetooth Gateway	47
7.3	Create a view to display the results	49
7.3.1	Implement the postHandler	53
8	(Optional)Using JavaScript libraries with NaviBLE.....	58
8.1.1	Request the Bean's temperature data.....	59
8.1.2	Request the Bean's accelerometer and LED data.....	61
8.1.3	To interpret the accelerometer data	62
8.2	Turn on LED.....	63

1 Overview

This sample Gateway project contains two servers: a frontend web server which will be responding to the user requests and route those request to the corresponding handlers at the gateway server and a Gateway server which will be interacting with the target devices in range and handle the read/write requests.

This guide describes how to install and implement both the Gateway and NaviBLE.(the frontend web server). You can use the NaviBLE website to explore the services and characteristics supported by any Bluetooth devices connected to your Bluetooth Gateway.



The sample code for NaviBLE and the Bluetooth Gateway are built using Node.js, so they will run on various platforms. In this exercise we will use a **Raspberry Pi 2 model B, with Raspbian Jessie operating system**.

The web front end app NaviBLE could also run on a hosted server, such as AWS EC2 or the equivalent and talk to the Bluetooth Gateway through a secure connection, but networking security and server deployments are out of the scope of this exercise. Instead, we will be building NaviBLE to run on the same host as the Bluetooth Gateway. We will explore various ways of deploying NaviBLE to the internet in a future lab.

This lab steps you through the following tasks:

1. Setting up Gateway server.
2. Setting up our NaviBLE project
3. Using NaviBLE to list services offered by connected devices
4. Using NaviBLE to list characteristics of each service
5. Using NaviBLE to updated characteristics of a service
6. Using JavaScript libraries with NaviBLE

Note: If you are more interested in the final solution and want to have a test run before you go through this step-by-step guide, please refer to the **Rapid Deployment Guide document** in this package.

Note: In the solution package, we have included all of the libraries you will need for the completed solution in the `node_modules` folders, so that you don't need to install them separately. In case you want to understand where and how to get those libraries, we listed those as **[optional]** steps..

1.1 Prerequisites for this lab

To successfully complete this lab, you must have the following:

A [Raspberry Pi](#) with TCP/IP connection and a Bluetooth module installed.

The Bluetooth Gateway is a simple web server running Node.js that implements portions of the GAP and GATT API as RESTful web services. You can find the GAP/GATT Restful API whitepaper here:

<https://developer.bluetooth.org/DevelopmentResources/Pages/White-Papers.aspx>

The code for the gateway is in **/gateway** folder. It is designed to run on nearly any platform that supports Node.js with very small memory and storage footprints. You can learn more about Node.js at nodejs.org.

The gateway works by scanning for Bluetooth devices. When it finds a device, the gateway caches the data structure of the services and characteristics supported by the device. When an HTTP request for the data structure is processed by the gateway, it responds to most of these requests from the cache. When there is a request for the current value of a characteristic from the target device, the gateway will start a connection and get/set the data immediately.

If you want to see how the gateway detects and caches device information, look at **/gateway/index.js**.

If you want to know how the gateway handles HTTP requests, look at the files in **/gateway/routes**. These files define the endpoints supported by the gateway and implement the handling of requests.

NOTE: This gateway does not implement every endpoint specified by the GAP/GATT RESTful API documentation. It also does not currently implement the expiration of device caches.

First, make sure you have **Node.js version 0.12.x** installed on your Pi.

To install Node.js, run the following at a terminal window:

```
sudo apt-get install curl
curl -sL https://deb.nodesource.com/setup_0.12 | sudo bash -
sudo apt-get install --yes nodejs
```

To check your installation, run:

```
node -v
```

The version number will display as a result of the above command.

The gateway depends on the following libraries that the node.js community provides:

- [hapi.js](#): A rich framework for building applications and services, hapi enables developers to focus on writing reusable application logic instead of spending time building infrastructure. This library provides the framework for the gateway's API.
- [crc](#): Module for calculating Cyclic Redundancy Check (CRC). CRCs are used to verify that messages between connected devices don't get corrupted.
- Various Bluetooth libraries: The gateway will use those libraries to call the Bluetooth APIs from the OS to use the Bluetooth radio. Please refer to the package.json for the depending libraries.
- [noble.js](#): A Node.js module for Bluetooth Low Energy central role- this library lets us talk to the OS level libraries for the Bluetooth radio using Node.js.

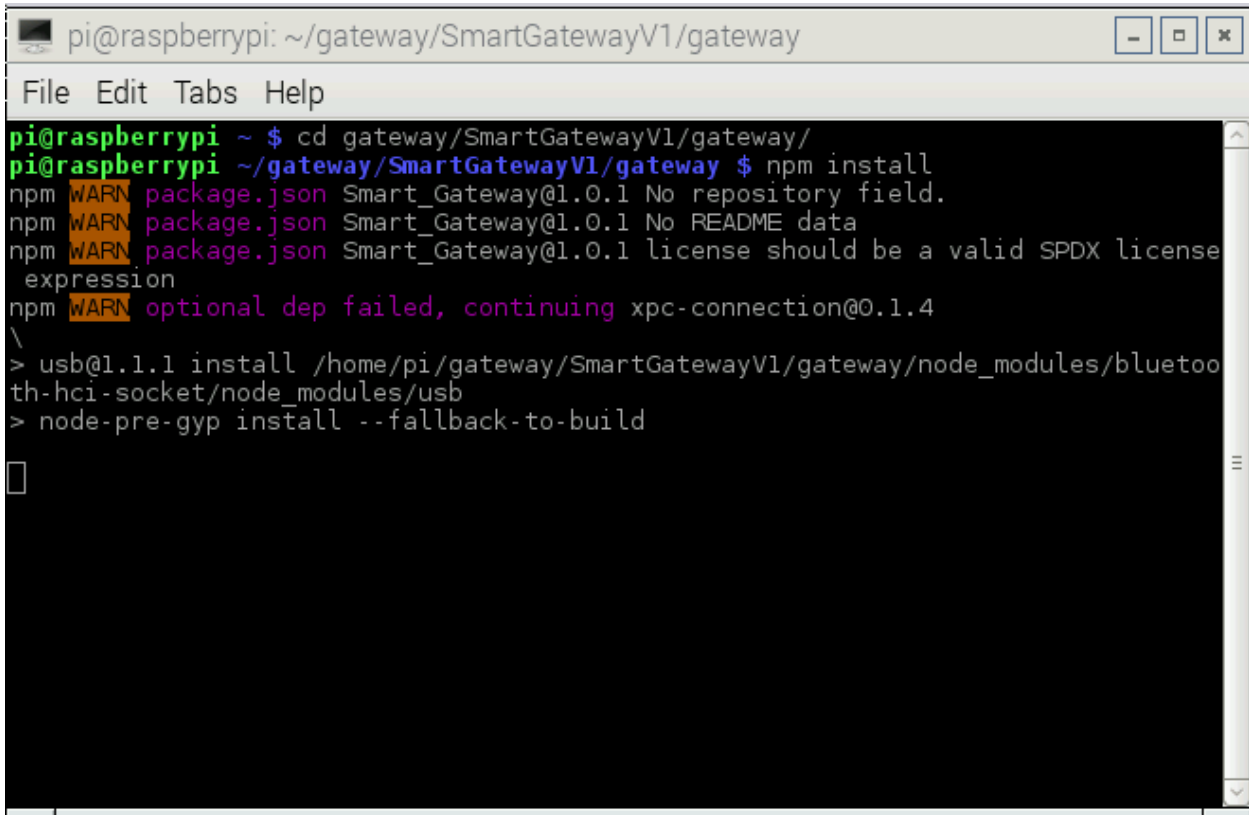
2 Install and start the Gateway server

The node package manager (npm) automatically manages the library dependencies. After installing the libraries, you need to install the gateway.

To install the gateway:

1. Download and extract the Gateway.zip file to a code directory in your home directory.
2. This creates a new **Gateway** directory upon extraction.
3. **[Optional]** Navigate to that new directory on your terminal and run the following:

```
npm install
```
4. This installs all the libraries your gateway requires to run.



```
pi@raspberrypi: ~/gateway/SmartGatewayV1/gateway
File Edit Tabs Help
pi@raspberrypi ~ $ cd gateway/SmartGatewayV1/gateway/
pi@raspberrypi ~/gateway/SmartGatewayV1/gateway $ npm install
npm WARN package.json Smart_Gateway@1.0.1 No repository field.
npm WARN package.json Smart_Gateway@1.0.1 No README data
npm WARN package.json Smart_Gateway@1.0.1 license should be a valid SPDX license expression
npm WARN optional dep failed, continuing xpc-connection@0.1.4
> usb@1.1.1 install /home/pi/gateway/SmartGatewayV1/gateway/node_modules/bluetooth-hci-socket/node_modules/usb
> node-pre-gyp install --fallback-to-build
```

You may notice the warnings from the installation, those are generated by NPM and has no impact on the installation result.

After the installation is complete, you can start your Gateway server.

To start the gateway server, run the following at the same terminal in the gateway directory:

```
sudo node .
```

After the gateway starts, it scans for any Bluetooth devices. As it discovers, connects to, and queries devices within range of your Raspberry Pi for the services and characteristics provided by each device, expect output similar to the following:

```
found a node: 10:40:f3:eb:c2:f2
Read characteristic: 8667556c9a374c9184ed54ee27d90049
```

Forcing a read

```
notification on for characteristic 10:40:f3:eb:c2:f2 : 8667556c9a374c9184ed54ee27d90049
```

In our exercise, the gateway scans for 10 seconds and then stops scanning. So if you start your gateway and then turn on a new device some time later, the gateway will identify the device and connect to it when it rescans. The amount of time of scanning can be configured in the `index.js` code.

After the scan is complete, you can use your web browser to access the Gateway's API. Open it up and go to <http://localhost:8001/gap/nodes> and view the device details.

The following is an example of what you can view about your devices:

```
{ "nodes": [ { "self": { "href": "http://localhost:8001/gap/nodes/10:40:f3:eb:c2:f2" }, "advertisement": { "localName": "Apple TV", "manufacturerData": { "type": "Buffer", "data": [76,0,9,6,2,20,10,0,1,39] }, "serviceData": [], "serviceUuids": [] }, "bdaddrs": [ { "bdaddr": "10:40:f3:eb:c2:f2" } ], "rssi": -57, "AD": [ { "ADType": "<type1>", "ADValue": "<value1>" } ] }, { "self": { "href": "http://localhost:8001/gap/nodes/32:00:13:54:03:4b" }, "advertisement": { "localName": "Kuna", "txPowerLevel": 8, "serviceData": [], "serviceUuids": [ "00001c00d10211e19b2300025b00a5a3" ] }, "bdaddrs": [ { "bdaddr": "32:00:13:54:03:4b" } ], "rssi": -88, "AD": [ { "ADType": "<type1>", "ADValue": "<value1>" } ] }, { "self": { "href": "http://localhost:8001/gap/nodes/cb:87:16:31:f6:cd" }, "advertisement": { "localName": "Tile", "serviceData": [], "serviceUuids": [ "feed" ] }, "bdaddrs": [ { "bdaddr": "cb:87:16:31:f6:cd" } ], "rssi": -89, "AD": [ { "ADType": "<type1>", "ADValue": "<value1>" } ] } ] }
```

Note: If you use Google Chrome, you can install a plugin that makes this type of output easier to read, as shown in the following figure. This plugin is available at [this link](#).

```
{
  - nodes: [
    - {
      - self: {
        href: "http://localhost:8001/gap/nodes/10:40:f3:eb:c2:f2"
      },
      - advertisement: {
        localName: "Apple TV",
        - manufacturerData: {
          type: "Buffer",
          - data: [
            76,
            0,
            9,
            6,
            2,
            20,
            10,
            0,
            1,
            39
          ]
        },
        serviceData: [ ],
        serviceUuids: [ ]
      },
      - bdaddrs: [
        - {
          bdaddr: "10:40:f3:eb:c2:f2"
        }
      ],
      rssi: -57,
      - AD: [
        - {
          ADType: "<type1>",
          ADValue: " <value1>"
        }
      ]
    },
    - {
      - self: {
        href: "http://localhost:8001/gap/nodes/22:00:13:54:03:45"
```

Note: If the links in the JavaScript Object Notation (JSON) do not work, it might be because the Bluetooth Gateway specification used to create this JSON specifies URLs that are not yet supported in this implementation. It is not yet a full implementation.

The Gateway implements portions of the RESTful Smart™ Server API. The API supports the endpoints in the following table:

<i>HTTP Action</i>	<i>Endpoint</i>	<i>Description</i>
GET	/gap/nodes	Get information about all discovered devices
GET	/gatt/nodes/{node_id}/services	Get a list of services supported by a device
GET	/gatt/nodes/{node_id}/services/{service_id}/characteristics	Get a list of characteristics supported by a service
GET	/gatt/nodes/{node_id}/services/{service_id}/characteristics/{characteristic_id}/value	Get the value of a characteristic
PUT	/gatt/nodes/{node_id}/services/{service_id}/characteristics/{characteristic_id}/value	Write the value of a characteristic

This concludes the install of the gateway itself. Now, on to NaviBLE!

3 Setting up NaviBLE

The Bluetooth NaviBLE web application uses the gateway to display devices within its range and lets us explore the services they offer and make changes to their characteristics, all from a web browser.

NaviBLE is a web application that you can use to discover and explore Bluetooth devices near your Raspberry Pi. This web app is built using open source tools from the Node.js community.

Following are the features that NaviBLE will support when you complete this lab:

1. Displays a list of devices along with their address and manufacturer
2. Displays a list of services supported by a device
3. Displays a list of characteristics supported by a service
4. Allows a user to read and write characteristic values using hexadecimal

Note: the NaviBLE app and all files are also included with the solution. If you have already installed the solution, then the files and code outlined in the following sections should already exist on your Raspberry Pi in the 'navible' directory. You can use these exercises to review the installed code and get a better understanding of it.

The following are screenshots of each feature you will build in this lab:

Gateway Explorer

Nodes

All Nodes

Address	RSSI	Manufacturer
• 10:40:f3:eb:c2:f2	-59	Apple TV
• cb:87:16:31:f6:cd	-87	Tile
• 32:00:13:54:03:4b	-86	Kuna
• c4:be:84:e5:4b:8a	-79	Bean

Gateway Explorer

cb:87:16:31:f6:cd Services

[All Nodes](#) -> Node: cb:87:16:31:f6:cd

Node Address	UUID	Primary?	
cb:87:16:31:f6:cd	feed	true	Characteristics

Gateway Explorer

Service Characteristics

[All Nodes](#) -> [Node: cb:87:16:31:f6:cd](#) -> Service: feed

UUID	Properties	Values
9d41000435d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> notify 	Get/Set Value
9d41000635d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000735d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000535d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000235d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> write 	Get/Set Value
9d41000835d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> writeWithoutResponse 	Get/Set Value
9d41000d35d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> writeWithoutResponse 	Get/Set Value
9d41000e35d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read writeWithoutResponse 	Get/Set Value

Gateway Explorer

Characteristic 9d41000235d6f4ddba60e7bd8dc491c0

[All Nodes](#) -> [Node: cb:87:16:31:f6:cd \(Tile\)](#) -> [Service: feed](#) -> Characteristic: 9d41000235d6f4ddba60e7bd8dc491c0

Type	Data	Write a Hex Value
		<input type="text"/> <input type="button" value="Write"/>

Bluetooth devices communicate with very small encoded chunks of data that work really well with low energy devices. However, it can make interacting with a Bluetooth device challenging for us humans. That's why our last feature is so important: Creating an interface for a specific device that a human can work with.

With this lab, you will create a page with buttons that allow you to get temperature, accelerometer, and LED data from a [LightBlue Bean from Punch Through Design](#). You will also create a form that you can use to turn the LED of the Bean On and Off and set its color. Here is an example of the page:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> [Characteristic: a495ff11c5b14b44b5121370f02d74de](#)

Hex Value	Human Readable Value
128,9,0,32,144,245,255,249,255,24,1,2,185,238	{"x":"-0.04305","y":"-0.02740","z":"1.09589"}

[Request LED data](#)

[Request Accelerometer data](#)

[Request Temperature](#)

Red

Green

Blue

[Change LED Color](#)

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

3.1 Create a node app

First you need to create a node app:

1. Open up your terminal and navigate to a convenient location to store your code. For example, create a directory named **code**.
2. Create a new directory named **navible**.
3. At a command prompt, change to that directory with the **cd** command.

Caution: This directory must not be in the gateway directory, because it must be a completely separate node application that can talk to your gateway.

4. Then run the following and **accept all of the defaults**:

```
npm init
```

Note: You need to type **yes** for the very last step.

5. Confirm that the **package.json** file is in your new **code** directory. This file defines your application and its dependencies.

3.2 [Optional] Install hapi.js

Next install **hapi.js**. Hapi.js is an easy to use framework for building web applications. Without it, you would have a lot more work to do teaching our web server how to respond to http requests.

To install hapi.js, run the following:

```
npm install --save hapi@9.0.3
```

This takes just a few seconds at most. If successful, you will see the following output:

```
navible (master)$ npm install --save hapi
npm WARN package.json navible@1.0.0 No description
npm WARN package.json navible@1.0.0 No repository field.
npm WARN package.json navible@1.0.0 No README data
hapi@9.0.3 node_modules/hapi
├── ammo@1.0.1
├── cryptiles@2.0.4
├── topo@1.0.3
├── peekaboo@1.0.0
├── items@1.1.0
├── kilt@1.1.1
├── catbox-memory@1.1.2
├── accept@1.1.0
├── call@2.0.2
├── statehood@2.1.1
├── boom@2.8.0
├── iron@2.1.3
├── shot@1.6.0
├── qs@4.0.0
├── hoek@2.14.0
├── catbox@6.0.0
├── mimos@2.0.2 (mime-db@1.17.0)
├── joi@6.6.1 (isemail@1.1.1, moment@2.10.6)
├── heavy@3.0.0 (joi@5.1.0)
└── subtext@2.0.0 (content@1.0.2, wreck@6.1.0, pez@1.0.0)
```

Great! Hapi.js will make building a web application easy!

3.3 [Optional] Install other tools

After hapi.js is installed, you then need to install the following tools:

- **Handlebars:** Gives you a template language for building dynamic web pages. It's named after its liberal use of the double braces (`{{ }}`), which look like handlebars if you look at them sideways.
- **Inert:** Makes it easy to serve up static files, such as CSS and JavaScript.
- **Vision:** Adds template rendering to the hapi.js application.

To install these tools, run the following:

```
npm install --save handlebars@3.0.3
npm install --save vision@3.0.0
npm install --save bluetooth-hci-socket@0.4.2 (this line not needed on a Macintosh)
npm install --save inert@3.0.1
```

Now look at the contents of the **package.json** file. They should look something like the following:

```
{
  "name": "NaviBLE",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "handlebars": "^3.0.3",
    "inert": "~3.0.1",
    "bluetooth-hci-socket": "~0.4.2"
    "hapi": "^9.0.3",
    "vision": "^3.0.0"
  }
}
```

The **package.json** file makes installing an application's dependencies easy. If you share this code with a friend after you have finished this lab, your friend will not have to install these dependencies one at a time, like you did. They can do it all in one command:

```
npm install
```

3.4 Create a hapi server

After the node modules are installed, you need to create a **hapi** server and configure it to use the modules.

To create and set up a hapi server:

1. Create an **index.js** file.
2. Add the following code to that file:

```
var Path = require('path');
var Hapi = require('hapi');

// Create a server with a host and port
var server = new Hapi.Server();

server.connection({
  host: 'localhost', //to access NaviBLE from another computer, comment out this line
  port: 8000
});

server.register([require('vision'), require('inert')], function (err) {
  if (err) {
    throw err;
  }
  server.views({
    engines: { html: require('handlebars') },
    path: __dirname + '/templates',
    layout: false
  });
});

// Start the server
server.start(function() {
  console.log('Server running at:', server.info.uri);
});
```

This code creates your server, tells it how to render views, and then starts the server.

3. After the server starts, run it:

```
node .
```

You should see output similar to this:

```
Server running at: http://localhost:8000
```

Congratulations! You can now point your browser to <http://localhost:8000> and see if you get the following output:

```
{
  statusCode: 404,
  error: "Not Found"
}
```

This output is expected at this point, because you have not told your server what URLs to support. You can do this using routes.

3.5 Set up routes for the server

Next, you need to set up your first route: <http://localhost:8000>.

This route is going to be your root route, but it will also be accessible at <http://localhost:8000/nodes> and will deliver a list of all nodes that your gateway has discovered.

To set up the root route:

1. Tell your server where to find your routes by adding a new line to `/index.js` after:

```
// Create a server with a host and port
var server = new Hapi.Server();
```

and pasting in the following code:

```
var routes = require('./routes');
```

2. Then tell your server to use the routes variable to determine what URLs to support. Add a new line to `/index.js` after:

```
server.views({
  engines: { html: require('handlebars') },
  path: __dirname + '/templates',
  layout: false
});
```

and paste in the following code:

```
server.route(routes)
```

3. Press Ctrl-C on the terminal to stop your node server and type the following to restart it:

```
node .
```

Note: You need to stop and restart your node server anytime you want to test changes. This can get a little tedious, so check out `nodemon` as an alternative.

An error similar to the following should occur:

```
module.js:338
    throw err;
    ^

Error: Cannot find module './routes'
    at Function.Module._resolveFilename (module.js:336:15)
    at Function.Module._load (module.js:278:25)
    at Module.require (module.js:365:17)
    at require (module.js:384:17)
    at Object.<anonymous> (/Users/dchristiansen/code/ble/student/NaviBLE/index.js:6:14)
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
    at Module.load (module.js:355:32)
    at Function.Module._load (module.js:310:12)
    at Function.Module.runMain (module.js:501:10)
```

This is expected, because you have not actually created any routes yet.

To create the routes:

1. Organize your routes into separate files in a **/routes** directory to avoid confusion and long file names.
2. Create a **nodes.js** file in **/routes**.
3. To define your first route, add the following code:

```
module.exports = [  
  { method: 'GET', path: '/', handler: nodes_handler },  
  { method: 'GET', path: '/nodes', handler: nodes_handler }  
]
```

This code simply declares which routes to support. If you were to restart your server and load this route in your browser, you would get a “nodes_handler” undefined error. The nodes_handler is a JavaScript object that does two important things:

- Sends an HTTP GET request to the gateway.
- And gets a JSON list of all the discovered nodes

4. Provide a variable named **nodes** to store the JSON nodes for the template to use with the following the code. Review the code and add it to the very beginning of **/routes/nodes.js**.

```
var nodes_handler = function (request, reply) {  
  var http = require('http');  
  var options = {  
    host: 'localhost',  
    port: 8001,  
    path: '/gap/nodes',  
    method: 'GET'  
  };  
  return http.get(options, function(response) {  
    // Continuously update stream with data  
    var body = '';  
    response.on('data', function(d) {  
      body += d;  
    });  
    response.on('end', function() {  
      // Data reception is done, do whatever with it!  
      var parsed = JSON.parse(body);  
  
      reply.view('nodes/index', {  
        title: 'Nodes',  
        nodes: parsed  
      });  
    });  
  });  
};
```

You will need to write a lot of code very similar to this, so make sure you understand it before you move on.

5. Restart your server and see what happens. You should expect something similar to the following output:

```
NaviBLE (master)$ node .
module.js:338
    throw err;
    ^

Error: Cannot find module './routes'
    at Function.Module._resolveFilename (module.js:336:15)
    at Function.Module._load (module.js:278:25)
    at Module.require (module.js:365:17)
    at require (module.js:384:17)
    at Object.<anonymous> (/Users/dchristiansen/code/ble/student/NaviBLE/index.js:6:14)
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
    at Module.load (module.js:355:32)
    at Function.Module._load (module.js:310:12)
    at Function.Module.runMain (module.js:501:10)
```

The error occurs because your server still does not know about your routes.

3.6 Create the `/routes/index.js` file

Your Hapi server cannot find the `/routes/index.js` file. By defining your routes in various files, you will make the code easier to understand, but you have to tell your server where to look for valid routes.

To create this file, use the following:

```
var nodes = require('./nodes');
module.exports = [].concat(nodes);
```

As you add additional routes, you will need to add them to this file.

Now that you have registered your route, your server is not going to work yet. What have you not done yet?

Take a look at **nodes.js** and make a guess. Then restart your server and refresh your browser to find out if you were right.

3.7 Create the nodes/index view

If you guessed that your server will not be able to find your **nodes/index** view, congratulations, you are a genius.

Caution: You must be running your Bluetooth Gateway server on port 8001 of the computer you are currently using or this will not work!

You are going to need to loop over each service stored in the nodes variable in this view. Handlebars provides a convenient helper method called “each” that makes this easy. Review the following code to get a good understanding of it.

To create the view:

1. Create a file called templates/nodes/index.html.
2. Add the following code to that file:

```
<div>
  <h1>{{title}}</h1>
  <div class='breadcrumbs'>
    All Nodes
  </div>
  <table>
    <thead>
      <tr>
        <th>Address</th>
        <th>RSSI</th>
        <th>Manufacturer</th>
      </tr>
    </thead>
    <tbody>
      {{#each nodes}}
        {{#each this}}
```

```
<tr>
  <td>
    <ul>
      {{#each this.bdaddrs}}
        <li>
          <a href='/nodes/{{this.bdaddr}}'>
            {{this.bdaddr}}
          </a>
        </li>
      {{/each}}
    </ul>
  </td>
  <td>
    {{this.rssi}}
  </td>
  <td>
    {{this.advertisement.localName}}
  </td>
</tr>
{{/each}}
{{/each}}
</tbody>
</table>
</div>
```

3. Now restart your server and try the route. If all went well, you should see something like this:

Nodes

All Nodes

Address	RSSI	Manufacturer
10:40:f3:eb:c2:f2	-59	Apple TV
cb:87:16:31:f6:cd	-87	Tile
32:00:13:54:03:4b	-86	Kuna
c4:be:84:e5:4b:8a	-79	Bean
e3:59:26:d4:ce:49	-81	Tile

A very simple format that works. However, you can create a better view by adding a layout and some CSS.

To improve the view, change the following **bolded** code in `/index.js`:

```
server.views({
  engines: { html: require('handlebars') },
  path: __dirname + '/templates',
  layout: true
});
```

This tells the server to use a layout for each template that it renders. You can use the following simple layout and save it in `/templates/layout.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel="stylesheet" href="/stylesheet.css" type="text/css" media="all">
  </head>
```

```
<body>
  <h1>Gateway Explorer</h1>
  {{{ content }}}
</body>
</html>
```

This code adds a stylesheet in the layout, so you need to create it.

3.8 Create a stylesheet

In this lab, you are going to use a number of static files as part of your app and each one needs a route definition. Instead of making you set these up one at a time for each route, this lab steps you through setting up the routes for every static file for this lab now. This will help speed you along and keep you focused on learning how to interact with the gateway.

To create the stylesheet:

1. Add the following code to your routes directory in the **/routes/assets.js** file. This code tells the server where to get the static files that you are going to need for this lab:

```
module.exports = [
  {
    method: 'GET',
    path: '/stylesheet.css',
    handler: function (request, reply) {
      reply.file('public/stylesheet.css');
    }
  },
  {
    method: 'GET',
    path: '/bean.js',
    handler: function (request, reply) {
      reply.file('public/bean.js');
    }
  },
  {
```

```
    method: 'GET',
    path: '/bean_message.js',
    handler: function (request, reply) {
      reply.file('public/bean_message.js');
    }
  },
  {
    method: 'GET',
    path: '/bean_data.js',
    handler: function (request, reply) {
      reply.file('public/bean_data.js');
    }
  }
]
```

2. You are not going to use all of these routes just yet, but this will save you some steps later. Now that you have defined these routes, you need to add them to **/routes/index.js** with the following code:

```
var nodes = require('./nodes');
var assets = require('./assets');
module.exports = [].concat(nodes, assets);
```

Now our server knows where to look for our static files when we need them. Let's finish this task off. Paste the following css into a file called **/public/stylesheet.css**:

```
html {
  font-family: "Trebuchet MS", Arial, Helvetica, sans-serif;
  width: 80%;
  margin-left: 10%;
}
strong, p {
  text-align: center;
}
```



```
h1 {
    text-align: center;
}

.btn {
    background: #3498db;
    background-image: -webkit-linear-gradient(top, #3498db, #2980b9);
    background-image: -moz-linear-gradient(top, #3498db, #2980b9);
    background-image: -ms-linear-gradient(top, #3498db, #2980b9);
    background-image: -o-linear-gradient(top, #3498db, #2980b9);
    background-image: linear-gradient(to bottom, #3498db, #2980b9);
    -webkit-border-radius: 28;
    -moz-border-radius: 28;
    border-radius: 28px;
    font-family: Arial;
    color: #ffffff;
    font-size: 13px;
    background: #3498db;
    padding: 4px 20px 4px 20px;
    text-decoration: none;
}

.btn:hover {
    background: #3cb0fd;
    background-image: -webkit-linear-gradient(top, #3cb0fd, #3498db);
    background-image: -moz-linear-gradient(top, #3cb0fd, #3498db);
    background-image: -ms-linear-gradient(top, #3cb0fd, #3498db);
    background-image: -o-linear-gradient(top, #3cb0fd, #3498db);
    background-image: linear-gradient(to bottom, #3cb0fd, #3498db);
    text-decoration: none;
}

.breadcrumbs {
    text-align: center;
    font-size: 14px;
    color: darkGray;
}
```

```
margin: 10px 0;

a:link, a:visited, a:hover, a:active {
    color: darkGray;
    font-weight: bold;
    text-decoration: none;
}

table {
    width: 80%;
    border-collapse: collapse;
    margin-left: 10%;
}

table td, table th {
    font-size: 1em;
    border: 1px solid #3cb0fd;
    padding: 3px 7px 2px 7px;
}

table th {
    font-size: 1.1em;
    text-align: left;
    padding-top: 5px;
    padding-bottom: 4px;
    background-color: #3498db;
    color: white;
}

table tr.alt td {
    color: #000000;
    background-color: #EAF2D3;
}
```

This is fairly basic CSS. Review it and make sure you understand it.

3. Then restart your server and reload your browser.

Congratulations!

You have just used the Bluetooth Gateway to create a list of all the devices your gateway has discovered. You are now ready to create a list of all the services for each device.

Warning: Do not click one of these device links just yet or your server will crash, because you have not set that part up yet.

4 Using Gateway server to explore the attributes of the target devices.

The gateway server handles the requests from the NaviBLE web server and interacts with the devices in its range.

In the index.js under /gateway, the code implements the following functions:

1. Initialize the libraries. Noble (<https://github.com/sandeepmistry/noble>) is a node.js module that we will use for BLE central role.
2. Set up the initial scanning. We will use a Timeout function to limit the scanning time to 10 seconds (10,000ms)
3. Service discovery and caching. For each and every devices that are advertising during our scanning, we will save the record in the cache and then initialize connections to explore more about the services and characteristics.
4. Disconnect the devices immediately once we are done.

The second component in the Gateway server is the routing handlers. We have the routing for

1. Nodes
2. Services
3. Characteristics
4. Values.

Basically, the routing handlers for the first 3 of the above are to get the data structure (the metadata) that is saved in the cache and send those out in JSON format. When we get the values of the characteristics, we will quickly start a new connection and read/write to the attribute. This is handled in the value routing handler.

The Gateway server also has an http daemon on port 8000. You can request the endpoints that we defined in the routing component and verify the response in JSON.

5 Using NaviBLE to handle the requests and UX.

You will do the following in this part of the lab:

1. Define a route and add it to the server
2. Implement the route using the Bluetooth Gateway
3. Create a view to display the results.

5.1 Define a route for getting services from a node

In this section you are going to define the following route:

```
/nodes/<node_id>
```

To define the route:

Add the following to the end of **/routes/nodes.js**:

```
module.exports = [  
  { method: 'GET', path: '/', handler: nodes_handler },  
  { method: 'GET', path: '/nodes', handler: nodes_handler },  
  { method: 'GET', path: '/nodes/{node_id}', handler: node_handler }  
]
```

Note: This adds one line of code that declares a new handler named **node_handler** to implement this route.

5.2 Implement the route using the Bluetooth Gateway

Then you need to implement the route.

Add the following code to the beginning of **/routes/nodes.js**:

```
var node_handler = function (request, reply) {  
  var http = require('http');  
  var node_uuid = request.params.node_id;  
  
  var options = {
```

```
    host: 'localhost',
    port: 8001,
    path: '/gatt/nodes/' + node_uuid + '/services',
    method: 'GET'
  };

  return http.get(options, function(response) {
    // Continuously update stream with data
    var body = '';
    response.on('data', function(d) {
      body += d;
    });
    response.on('end', function() {
      // Data reception is done, do whatever with it!
      var parsed = JSON.parse(body);

      reply.view('nodes/show', {
        title: node_uuid + ' Services',
        device: node_uuid,
        services: parsed
      });
    });
  });
};
```

This code is very similar to the **nodes_handler** you defined previously in this lab. The main difference is that you introduced a parameter in the handler that is defined by the **node_uuid** request. You need the address of the device for which you want to get the services. This is provided in the **paramet** request. Notice that this parameter builds the URL for your **HTTP GET** request to the API Gateway and provides it to your view for use in the title.

5.3 Create a view to display the results

Now you need to add the view. Review the following code, particularly noticing the places where this code uses the handlebar (**{{ }}**) syntax to access the variables provided by the node handler.

Previously, the code iterated over each device, this code iterates over each service supported by the device.

To create the view:

1. Add the following code to **/templates/nodes/show.html**:

```
<div>
  <h1>
    {{title}}
  </h1>
  <div class='breadcrumbs'>
    <a href='/nodes'>
      All Nodes
    </a>
    ->
    Node: {{device}}
  </div>
  <table>
    <thead>
      <tr>
        <th>Node Address</th>
        <th>UUID</th>
        <th>Primary?</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      {{#each services}}
        {{#each this}}
          <tr>
            <td>{{../../device}}</td>
            <td>{{this.uuid}}</td>
            <td>{{this.primary}}</td>
            <td>
```

```

    <a href='/nodes/{{node}}'>
      {{this.bdaddr}}
    </a>
    <a href='/nodes/{{../../device}}/services/{{this.uuid}}'>
      Characteristics
    </a>
  </td>
</tr>
{{/each}}
{{/each}}
</tbody>
</ul>
</div>

```

Note: This adds breadcrumbs, so that you can go back to all nodes, as well as links to the characteristics of each service.

- Restart your server, click on a node address link, and then click **All Nodes** to test the breadcrumb feature.

The following screen shows you where to click **All Nodes**:

Caution: Do not click the **Characteristics** link yet or your server will crash. You will make this work in the next section of this lab.

Gateway Explorer

cb:87:16:31:f6:cd Services

[All Nodes](#) -> Node: cb:87:16:31:f6:cd

Node Address	UUID	Primary?	
cb:87:16:31:f6:cd	feed	true	Characteristics

6 Using NaviBLE to list characteristics of each service

This section will seem repetitive, but do not worry, it adds on to what you have just learned.

In this section, you will do the following tasks:

1. Define a route and add it to the server
2. Implement the route using the Bluetooth Gateway
3. Create a view to display the results.

6.1 Define a route for getting characteristics of a service

First, you need to tell your server about the new route.

To define the route:

1. Add the following to **/routes/index.js**:

```
var nodes = require('./nodes');  
var services = require('./services');  
var assets = require('./assets');  
  
module.exports = [].concat(nodes, assets, services);
```

2. The following code defines the route with **two** parameters: a **node id** and a **service id**. You need both to interact with the gateway. Add the following to the new **/routes/services.js** file:

```
module.exports = [  
  { method: 'GET', path: '/nodes/{node_id}/services/{service_id}', handler: service_handler  
  }  
]
```

6.2 Implement the route using the Bluetooth Gateway

Now you need to create the service handler. This follows the same steps as you did previously in this lab:

1. Get the parameters we need from the request
2. Prepare to call the gateway
3. Call the gateway
4. Send the results from the gateway to a view that is rendered to the user

Review the following code and make sure you understand what it does, then proceed.

To implement the route:

Add the following to the beginning of **/routes/services.js**:

```
var service_handler = function (request, reply) {
  var http = require('http');
  var node_uuid = request.params.node_id;
  var service_uuid = request.params.service_id;
  var options = {
    host: 'localhost',
    port: 8001,
    path: '/gatt/nodes/' + node_uuid + '/services/' + service_uuid + '/characteristics',
    method: 'GET'
  };
  return http.get(options, function(response) {
    // Continuously update stream with data
    var body = '';
    response.on('data', function(d) {
      body += d;
    });
    response.on('end', function() {
      // Data reception is done, do whatever with it!
      var parsed = JSON.parse(body);
      reply.view('services/show', {
        title: 'Service Characteristics',
        device: node_uuid,
```

```
        service: service_uuid,  
        characteristics: parsed  
    });  
});  
});  
};
```

6.3 Create a view to display the results

As indicated in the code that you just added, you now need to create a view to display the results to the user.

Review the code, particularly noting how it uses `{{#each}}` twice to iterate over each characteristic and its properties. Also review the additional code for the breadcrumbs. Please take note of how the code is iterating first over characteristics and then over all the properties of each characteristic.

To create the view:

1. In `/templates/services/show.html`, create the **services/show** file.
2. Add the following code to this new file:

```
<div>  
  <h1>  
    {{title}}  
  </h1>  
  
  <div class='breadcrumbs'>  
    <a href='/nodes'>  
      All Nodes  
    </a>  
    ->  
    <a href='/nodes/{{device}}'>  
      Node: {{device}}  
    </a>  
    ->  
    Service: {{service}}  
  </div>
```

```

<table>
  <thead>
    <tr>
      <th>UUID</th>
      <th>Properties</th>
      <th>Values</th>
    </tr>
  </thead>
  <tbody>
    {{#each characteristics}}
      <tr>
        <td>{{this.uuid}}</td>
        <td>
          <ul>
            {{#each this.properties}}
              <li>{{this}}</li>
            {{/each}}
          </ul>
        </td>
        <td>
          <a href='/nodes/{{../device}}/services/{{../service}}/characteristics/{{this.uuid}}'>
            Get/Set Value
          </a>
        </td>
      </tr>
    {{/each}}
  </tbody>
</ul>
</div>

```

- Restart your server, click the **All Nodes** link and then click a characteristic link to see how it works. The following is an example of a list of all the nodes and their service characteristics:

Gateway Explorer		
Service Characteristics		
All Nodes -> Node: cb:87:16:31:f6:cd -> Service: feed		
UUID	Properties	Values
9d41000435d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> notify 	Get/Set Value
9d41000635d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000735d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000535d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read 	Get/Set Value
9d41000235d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> write 	Get/Set Value
9d41000835d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> writeWithoutResponse 	Get/Set Value
9d41000d35d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> writeWithoutResponse 	Get/Set Value
9d41000e35d6f4ddba60e7bd8dc491c0	<ul style="list-style-type: none"> read writeWithoutResponse 	Get/Set Value

7 Getting and setting values for a characteristic

This section adds on to what you have just done in previous sections. However, this time you are going to do the tasks twice: first for a GET to retrieve a value of a characteristic and then again for a POST to write a value to a characteristic.

This section steps you through the following:

1. Define routes and add them to the server
2. Implement the route using the Bluetooth Gateway
3. Create a view to display the results.

7.1 Define routes and add them to the server

Your first task is to define routes to get and set a characteristic value and then add the routes to the server.

To do this:

1. Create a **/routes/characteristics.js** file.
2. Add the following to that file:

```
module.exports = [
  {
    method: 'GET',
    path: '/nodes/{node_id}/services/{service_id}/characteristics/{characteristic_id}',
    handler: handler
  },
  {
    method: 'POST',
    path: '/nodes/{node_id}/services/{service_id}/characteristics/{characteristic_id}',
    handler: postHandler
  }
]
```

Notice that both routes are added at the same time. This saves time later because you already know you will need both routes.

3. Next add this new routes file to **routes/index.js** with the following:

```
var nodes = require('./nodes');
var services = require('./services');
var assets = require('./assets');
var characteristics = require('./characteristics');

module.exports = [].concat(nodes, assets, services, characteristics);
```

7.2 Implement the route using the Bluetooth Gateway

To start, you need to implement the handler for the GET route. Then create a stub for the POST handler so that your server does not crash.

Review the following handler code that gets the values. This code is calling the gateway twice: once to get a characteristic value and again to get the manufacturer info of the device. Why add this to the code? Because it checks if the device is a Bean and if it is, it will add some specific device interactions to the page.

Note how the code loops over all the devices until it find yours. This code gets even more parameters from the request. In addition to the device and service, it is also getting the characteristic ID. Also, take note of the end of the code that stubs out the postHandler.

To implement the handler:

Add the following code to the very beginning of **routes/characteristics.js**, before the code that you added previously:

```
var handler = function (request, reply) {
  var http = require('http');
  var node_uuid = request.params.node_id;
  var service_uuid = request.params.service_id;
  var characteristic_uuid = request.params.characteristic_id;
  var write_url = '/nodes/' + node_uuid + '/services/' +
    service_uuid + '/characteristics/' + characteristic_uuid
  var options = {
    host: 'localhost',
    port: 8001,
```

```
    path: '/gatt/nodes/' + node_uuid + '/services/' +
        service_uuid + '/characteristics/' +
        characteristic_uuid + '/value' ,
    method: 'GET'
};

return http.get(options, function(response) {
    // Continuously update stream with data
    var body = '';
    response.on('data', function(d) {
        body += d;
    });
    response.on('end', function() {
        // Data reception is done, do whatever with it!
        var parsed = JSON.parse(body);

        // Let's figure out if this is a bean
        var options = {
            host: 'localhost',
            port: 8001,
            path: '/gap/nodes',
            method: 'GET'
        };

        return http.get(options, function(nodes_response) {
            // Continuously update stream with data
            var nodes_body = '';
            nodes_response.on('data', function(d) {
                nodes_body += d;
            });
            var manufacturer = ''
            nodes_response.on('end', function() {
                var nodes_parsed = JSON.parse(nodes_body);
                for (var i in nodes_parsed.nodes) {
```



```
        console.log(nodes_parsed.nodes[i]);
        if (nodes_parsed.nodes[i].bdaddrs[0].bdaddr == node_uuid) {
            manufacturer = nodes_parsed.nodes[i].advertisement.localName
        }
    }
    reply.view('characteristics/show', {
        title: 'Characteristic ' + characteristic_uuid,
        device: node_uuid,
        service: service_uuid,
        characteristic: characteristic_uuid,
        value: parsed,
        write_url: write_url,
        manufacturer: manufacturer,
        bean: manufacturer == 'Bean'
    });
});
});
});
});
});

var postHandler = function (request, reply) {} //This is a STUB function that we will implement later
```

Note: There are other ways of calling the gateway twice in the node. You could use promises, or the async library to manage the calls. For example, JavaScript Promises are a better way to do this, but they are beyond the scope of this lab.

7.3 Create a view to display the results

Now you need to build the view. This view will be similar to previous ones in this lab, except that this one adds an `{{#if bean}}` statement that renders the view differently with a bean. This paves the way to use some special logic for the bean because you know how to communicate with it.

Note: You could use the same logic for any device if you knew the specific device's communication standards.

To do this:

1. Create the **/templates/characteristics/show.html** file.
2. Add the following to that file:

```
<div>
  <h1>
    {{title}}
  </h1>
  <div class='breadcrumbs'>
    <a href='/nodes'>
      All Nodes
    </a>
    ->
    <a href='/nodes/{{device}}'>
      Node: {{device}} ({{manufacturer}})
    </a>
    ->
    <a href='/nodes/{{device}}/services/{{service}}'>
      Service: {{service}}
    </a>
    ->
    Characteristic: {{characteristic}}
  </div>
  {{#if bean}}
  <table>
    <thead>
      <tr>
        <th>Hex Value</th>
        <th>Human Readable Value</th>
      </tr>
```

```
</thead>
<tbody>
  <tr>
    <td id='buffer_value'>{{value.value.data}}</td>
    <td id='human_readable_value'></td>
  </tr>
  <tr>
    <td>

    </td>
    <td>

    </td>
  </tr>
</tbody>
</table>
<p>
  <strong> Important Notes about the Bean </strong>
</br>
<ul>
  <li>
    For some reason you have to write a value TWICE to the LED to get it
    to change color.
  </li>
  <li>
    After you write the LED, request the LED values
    to confirm that the color changed.

    The buffer will contain the RGB values in positions 6, 7, 8.
  </li>
</ul>
</p>
{{else}}
<table>
```

```

<thead>
  <tr>
    <th>Type</th>
    <th>Data</th>
    <th>Write a Hex Value</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>{{value.value.type}}</td>
    <td>{{value.value.data}}</td>
    <td>

    </td>
  </tr>
</tbody>
</table>
{{/if}}
</div>

```

- Restart your server and navigate to a characteristic for a non-bean device, as shown in the following screen:

Gateway Explorer

Characteristic 9d41000635d6f4ddba60e7bd8dc491c0

[All Nodes](#) -> [Node: cb:87:16:31:f6:cd \(Tile\)](#) -> [Service: feed](#) -> Characteristic: 9d41000635d6f4ddba60e7bd8dc491c0

Type	Data	Write a Hex Value
Buffer	74,69,51,52,49,48,52,52,51,84,65,0,0,0,0,0,0,0,0	

- Now try the same thing with a lone Bean characteristic, as shown in the following screen:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Hex Value	Human Readable Value
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0	

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

7.3.1 Implement the postHandler

In each case, room is available for the feature to write a value to a characteristic. To do that, start with implementing the postHandler.

As compared to the Get handler, the postHandler is simple. It looks similar to most handlers that you have used and requires the same steps:

1. Get the needed parameters from the request
2. Prepare to call the gateway
3. Call the gateway
4. Send the results from the gateway to a view that is rendered to the user

However, these steps include two important differences:

- This code uses a PUT instead of a GET.
- And instead of rendering a view, this code redirects it to the GET action.

Review the following code and see how it calls the API Gateway in a different way.

Then replace this line in **/routes/characteristics.js**:

```
var postHandler = function (request, reply) {}
```

with the following code:

```
var postHandler = function (request, reply) {  
  var http = require('http');  
  var node_uuid = request.params.node_id;  
  var service_uuid = request.params.service_id;  
  var characteristic_uuid = request.params.characteristic_id;  
  var value = request.payload.new_value  
  var url = '/nodes/' + node_uuid + '/services/' +  
            service_uuid + '/characteristics/' + characteristic_uuid  
  
  var options = {  
    host: 'localhost',  
    port: 8001,  
    path: '/gatt/nodes/' + node_uuid +  
          '/services/' + service_uuid + '/characteristics/' +  
          characteristic_uuid + '/value/' + value,  
    method: 'PUT'  
  };  
  
  var req = http.request(options, function(res) {  
    res.on('data', function(d) {  
      reply.redirect(url);  
    });  
  });  
  
  req.end();  
  req.on('error', function(e) {  
    console.log(e)  
  });  
};
```

By redirecting to the GET action after it posts, the code reads the new value of the characteristic and displays it.

Next, you need to add a way to use this new route in the UI.

To do this:

1. Add a form to the non-Bean portion of **/templates/characteristics/show.html**.
2. Find the very last **<td>** tag in the view and add the following form that enters a hexadecimal value and posts it:

```
<tbody>
  <tr>
    <td>{{value.value.type}}</td>
    <td>{{value.value.data}}</td>
    <td>
      <form accept-charset="UTF-8" method="post" novalidate="novalidate">
        <input type='text' name="new_value">
        <br/>
        <input type="submit" value="Write" class: 'btn'>
      </form>
    </td>
  </tr>
</tbody>
```

This form is not particularly special. It posts directly to the page you are on, which is what you want, and then it adds a **new_value** parameter.


To try it, type something random, such as **catfish** in the input field and submit it, as shown in the following screen.

Gateway Explorer

Characteristic 9d41000535d6f4ddba60e7bd8dc491c0

[All Nodes](#) -> [Node: e3:59:26:d4:ce:49 \(Tile\)](#) -> [Service: feed](#) -> Characteristic: 9d41000535d6f4ddba60e7bd8dc491c0

Type	Data	Write a Hex Value
Buffer	174	<div>catfish</div> <div>Write</div>



CATFISH???

It seems like nothing happens. However, if you look through the terminal output from the API Gateway, you should see the following output:

```
TypeError: Uncaught error: Invalid hex string
```

8 (Optional) Using JavaScript libraries with NaviBLE

Bluetooth devices communicate using short messages in hexadecimal format. This is convenient for things with electronic brains that are fluent in hexadecimal. However, human brains do not. So this code translates hexadecimal values into something you can read.

The tricky part is that each device encodes their data in different ways, so before you can get data that is meaningful to you, you must understand how that data is encoded. Fortunately, you now understand the way beans encode their data.

This section adds a couple of nice JavaScript libraries to translate the data back and forth between you and your bean. Explaining how they work is beyond the scope of this lab, however you can use them to interact with a Bean in a meaningful way.

To add these libraries to your project:

1. Locate the **bean_data.js** and **bean_message.js** files in **utilities.zip** and add them to your **/public** directory.
2. You added the routes for these files earlier, but you need to include them in the **layout.html**:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel="stylesheet" href="/stylesheet.css" type="text/css" media="all">
    <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <script src="/bean.js" type="text/javascript"></script>
    <script src="/bean_message.js" type="text/javascript"></script>
    <script src="/bean_data.js" type="text/javascript"></script>
  </head>
  <body>
    <h1>Gateway Explorer</h1>
    {{{ content }}}
  </body>
</html>
```

This code also adds jQuery, a popular JavaScript framework and a new **bean.js** file. However, first it needs to update the Bean portion of the view, so it can write to characteristics.

8.1.1 Request the Bean's temperature data

This code adds a button to request the Bean's temperature data. Requesting data from the Bean is done by first WRITING a message to its characteristic, then reading that characteristic. This is a little counter intuitive, but that is how it works.

It also creates a form with a hard-coded hex value that "gives you the temperature" in a hidden field. You already know this value so the code does not have to translate it.

To implement the code:

1. Add the code shown in bold below between the first `<td>` and `</td>` after `{{#if bean}}` in **/templates/characteristics/show.html**.

Note: the hexadecimal value in the hidden field and that **80020020116d5e** equals "What is the temperature?"

```
<tbody>
  <tr>
    <td id='buffer_value'>{{value.value.data}}</td>
    <td id='human_readable_value'></td>
  </tr>
  <tr>
    <td>
      <form accept-charset="UTF-8" method="post" novalidate="novalidate">
        <input type='hidden' name="new_value" value='80020020116d5e'>
        <input type="submit" value="Request Temperature" class='btn'>
      </form>
    </td>
    <td>
  </td>
</tr>
</tbody>
```

- Now navigate to your Bean's characteristic page. It should look like the following example:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Hex Value	Human Readable Value
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0	
<button>Request Temperature</button>	

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

- Click **Request Temperature** and you should see a screen similar to the following example:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Buffer Value	Human Readable Value
192,3,0,32,145,22,53,32	
<button>Request Temperature</button>	

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

The **Buffer Value** column has an encoded value of the response from the Bean. This is not very useful at first glance, until you learn what the value represents:

The sixth value in the list is the temperature in Celsius. As shown in this screen, it is 22 degrees Celsius or 71.6 degrees Fahrenheit.

8.1.2 Request the Bean's accelerometer and LED data

You can also request the accelerometer and LED values the same way.

To do this:

1. Update the code as follows:

```
<td>

<form accept-charset="UTF-8" method="post" novalidate="novalidate">
  <input type='hidden' name="new_value" value='a0020020024f0c'>
  <input type="submit" value="Request LED data" class='btn'>
</form>

<br/>

<form accept-charset="UTF-8" method="post" novalidate="novalidate">
  <input type='hidden' name="new_value" value='c0020020107d7f'>
  <input type="submit" value="Request Accelerometer data" class='btn'>
</form>

<br/>

<form accept-charset="UTF-8" method="post" novalidate="novalidate">
  <input type='hidden' name="new_value" value='80020020116d5e'>
  <input type="submit" value="Request Temperature" class='btn'>
</form>

</td>
```

2. Restart your server, refresh your browser, and click the **Request LED data** button.

For the LED data, the sixth, seventh, and eighth values are red, green, and blue, respectively. Unless you turn your Bean's LED on with the Bean Loader or some other app, it's just going to be 0,0,0 for now.

The following is an example of how the screen shows these buttons:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Buffer Value	Human Readable Value
192,3,0,32,145,22,53,32	
<div>Request LED data</div>	
<div>Request Accelerometer data</div>	
<div>Request Temperature</div>	

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

8.1.3 To interpret the accelerometer data

Unfortunately, getting the accelerometer data is a bit more complicated than just counting digits. To see the accelerometer, you can add a feature to turn the buffer value into something meaningful.

To do this:

1. Create a new **public/bean.js** file.
2. Then use the libraries that talk to the bean to handle the buffer values returned by the API Gateway. The **bean_data.js** library (that you added earlier) makes it easy to decode the bean data. All you need to do is provide the buffer value to it and it will return JSON.

3. Add the following in **bean.js**:

```
$(document).ready(function() {
    var buffer_string = JSON.parse("(" + $('#buffer_value').html() + ")");
    human_readable_value = new BeanData(buffer_string)['data'];
    $('#human_readable_value').html(JSON.stringify(human_readable_value));
})
```

This JavaScript retrieves the buffer value from your page. And because it is retrieved as a string, the code converts it to a buffer. Then passes the buffer to BeanData, grabs the result, and updates your page, as shown in the following screenshot:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Buffer Value	Human Readable Value
192,3,0,32,145,22,53,32	{"temp":{"C":22,"F":71.6}}

Request LED data

Request Accelerometer data

Request Temperature

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

8.2 Turn on LED

One last thing to do is turn on the LED. To start, create a form with three input fields: red, green, and blue. Then create a hidden field that stores the hexadecimal value that the red, green, and blue values translate to.

To do this:

1. On the bean, click **Change LED Color** and provide RGB (red, green, blue) values. Each value should be an integer between 0 and 255. You can learn more about RGB values [here](#).
2. Add the following code in the **td** tag after the **td** tag that you put the other forms in:

```
<form accept-charset="UTF-8" method="post" id="rgb" novalidate="novalidate">
  <label>Red</label>
  <input type="text" name="red" value="0">
  <br/>
  <label>Green</label>
  <input type="text" name="green" value="0">
  <br/>
  <label>Blue</label>
  <input type="text" name="blue" value="0">
  <input type='hidden' name="new_value" value='8005002001000000f669'>
  <br/>
  <br/>
  <input type="submit" value="Change LED Color" class='btn'>
</form>
```


The following shows an example of the new data in the right column:

Gateway Explorer

Characteristic a495ff11c5b14b44b5121370f02d74de

[All Nodes](#) -> [Node: c4:be:84:e5:4b:8a \(Bean\)](#) -> [Service: a495ff10c5b14b44b5121370f02d74de](#) -> Characteristic: a495ff11c5b14b44b5121370f02d74de

Buffer Value	Human Readable Value
192,3,0,32,145,22,53,32	{"temp":{"C":22,"F":71.6}}
<div>Request LED data</div> <div>Request Accelerometer data</div> <div>Request Temperature</div>	<div>Red <input type="text"/></div> <div>Green <input type="text"/></div> <div>Blue <input type="text"/></div> <div>Change LED Color</div>

Important Notes about the Bean

- For some reason you have to write a value TWICE to the LED to get it to change color.
- After you write the LED, request the LED values to confirm that the color changed. The buffer will contain the RGB values in positions 6, 7, 8.

Now you need to add some JavaScript that triggers each time one of the RGB values change. The following JavaScript grabs the value of each field, sends it to the BeanMessage library, and updates the hidden field with the result.

3. Add the following code to the end of **/public/bean.js**:

```
$(document).on('change', '#rgb input', function() {  
    setLED();  
})  
  
function setLED() {  
    var r = $('#rgb input[name=red]')[0].value;  
    var g = $('#rgb input[name=green]')[0].value;  
    var b = $('#rgb input[name=blue]')[0].value;  
    var messageBuilder = new BeanMessage();  
    var hex_value = messageBuilder.setLED(r, g, b).toString('hex');  
    $('#rgb input[name=new_value]').val(hex_value)  
}
```

And then try it. If all goes as planned, you can turn the LED off and on and see how the data reflects the status change.

Congratulations! You have implemented a Bluetooth Explorer!