Bring IoT home: Connect your car to your home

Use IBM Watson IoT Platform and a Raspberry Pi, Bluetooth device, and programmable power strip to turn on your lights and appliances as your car approaches home

Josef Reisinger

Use simple smart devices (a Raspberry Pi, Bluetooth device, and programmable power outlet strip) and a simple IoT platform (IBM Cloud, IBM Watson IoT Platform, and MQTT) to turn on lights and appliances as your car approaches home.

Connected cars and smart homes are hot topics these days in the IoT domain. If you look closer, you find that connected cars and smart homes seem to exist in silos. Yes, a smart home resident is able to turn on lights and appliances with his mobile phone. But why can't the car, as it is approaching the smart home, do exactly the same thing?

In this tutorial, you'll learn how to use IBM Cloud and Node-RED to connect a car to a smart home, enabling the car to automatically activate household features, such as turning on the lights and adjusting the air conditioning.

We will use a Raspberry Pi to sense the signal strength of a Bluetooth device in the smart home and send that information to the Watson IoT Platform. Then, based on the signal strength, we will turn on an electrical appliance with that Bluetooth device. Also, we will use GPS-based tracking of the car, creating a geo-fence around the smart home, to be able to essentially connect your car to your home.

What you'll need to build this IoT app

This hands-on tutorial is based a Proof of Concept (PoC) that I did on short notice with equipment that I found at home at the time that I did the PoC. You can use the same devices, or you can substitute the devices with what you need for your particular purpose.

You need the following devices, accounts, or skills to build this IoT app as I built it:

- A Linux system with a Bluetooth device, such as a Raspberry Pi with a USB Bluetooth dongle. Even though any Linux device with a C-Compiler configured will work, nobody really wants to run a computer on a 24 x 7 schedule as a device gateway.
• A power outlet that can be turned on from a PC. I used a Gembird Silver Shield (a programmable power outlet strip), but there are others brands that might work better for you.
• Familiarity with the C programming language and C development in a Linux system (such as Fedora, which is a Linux-based, open source operating system).
• An IBM Cloud® account. (You can request a free trial here).

To see IBM Cloud, Mosquitto, and Node-RED in action, watch me in this video walk you through how to connect a Raspberry Pi to the Watson IoT Platform and write a simple "hello world" IoT app to read and display data.

• Familiarity with IBM Cloud. You need to know how to browse the catalog and instantiate one of the IBM Cloud services.
• Familiarity with MQTT, especially the Eclipse Mosquitto library, which is an open source message broker that implements the MQTT protocol and that you can find in all major Linux distributions. Use your Linux platform's package management tool to search for Mosquitto, and install the package on your device.
• Familiarity with Node-RED. You need to understand how to use nodes in Node-RED and how you process a message in Node-RED. Node-RED is a front end to JavaScript, so you also need some experience with JavaScript. You can use the getting started documentation on the Node-RED website to help you gain this familiarity.

Get the code
Run the app

Architecture of our sample IoT app
Figure 1 depicts the first phase of the architecture of the IoT app that receives the strength of a Bluetooth signal.

Figure 1. Architecture diagram of the sample IoT application and the devices

In Figure 1, the numbers in the circles guide you through the steps to build the IoT application:
1. In IBM Cloud, you create an app by using the Internet of Things Platform Starter. The app will read sensor data that is received from a device and send data to a device. To protect the app from arbitrary devices sending data to it, each device needs to authenticate itself by using an API token. Likewise, the app needs to authenticate itself by using an API token to be able to read and process the device data that it receives.

2. On a Raspberry Pi device, you create an MQTT client to read the Bluetooth signal strength and continuously send the values to IBM Watson IoT platform.

3. You create a Node-RED app to process the data and to generate commands to send to the programmable power strip to turn on the lights and air conditioning in the home.

After you successfully implement this initial version of the IoT app, you will learn how to extend it to take the GPS coordinates of the car into account. You will also develop a dashboard to define the geo-fence for the GPS control function and to set the sensitivity of the Bluetooth approximation sensor.

Let's get going!

1. **Create your IoT app and connect your Raspberry Pi to the Watson IoT Platform**

   First, let's get our Raspberry Pi connected to the Watson IoT Platform.

   **1a. Create your IoT app in IBM Cloud**
   
   1. Log in to your IBM Cloud account.
   2. Use the Internet of Things Platform Starter boilerplate to create your app. You can choose your own name for the app; in this tutorial, I use `dw-example-iot`. The application will be started, which might take a while.
   3. Click the overview link in the navigation pane to the left, and wait for the service to be deployed.
   4. Click **ADD A SERVICE OR API**.
   5. Find the Internet of Things Platform service, and click **CREATE**.

   **1b. Register your Raspberry Pi with the Internet of Things Platform service and identify authentication and authorization tokens**

   To send data to the Watson IoT Platform, you need to register your devices to the Internet of Things Platform service. In this tutorial, I use the Raspberry Pi as a gateway device. The authentication token that is generated when you register your device allows the device to send the data to the Watson IoT Platform.

   To identify a device uniquely, it needs a client ID. A client ID is created with this predefined format, where `typeid` is a string that identifies the purpose of devices and where "device-id" is a unique identifier of your device:

   \[
   d:<\text{organization}>:<\text{typeid}>:<\text{device-id}>
   \]

   Issue the following command on your Raspberry Pi device to identify the unique MAC address for the device, which you can use as the unique device-id:
To register your device, complete these steps:

1. In your IBM Cloud app dashboard, click the Internet of Things Platform service.
2. In the left column, at the bottom, click **Launch dashboard**. A new window or tab opens in your browser. Your organization ID is displayed at the top of the dashboard.
3. In the **Devices** section below the Organization ID, click **Add Device**.
4. Specify **proximity** as the device type (**typeid**), and specify the MAC address of your Ethernet device, without the colons, as your device ID (**device-id**).
5. Click **Continue**.
6. Carefully review the next page, and make a note of the device credentials, specifically the **Authentication Token** for your device.

### 1c. Test your connectivity

Now that our device is registered with Watson IoT Platform, you can use the authentication token to send a test message from the Raspberry Pi. Let's use the Mosquitto MQTT library. For example, log in to your Raspberry Pi and issue the following Mosquitto command, providing your own values for the organization and device-id:

```
mosquitto_pub -h e62whi.messaging.internetofthings.ibmcloud.com \
-i d:e62whi:proximity:b872bef27aa8 \
-u use-token-auth -P '<your device credentials here>' \
-t /iot/x -m '{"d":"hello iot"}''
```
In the dashboard for Watson IoT Platform, you’ll see the message "hello iot" in the sensor information for your device:

While we are in the Watson IoT Platform dashboard, let's make sure that our Node-RED application is authorized to consume the device data that is sent to our IoT app. Instead of an authentication token, we need an API authorization token. To authorize an app to consume data from this app, complete these steps:

1. In the Watson IoT Platform dashboard, click **Access** and then **API Keys**.

2. Click **Generate API Key**.

3. Carefully review the next page, and make a note of the API credentials, specifically the **Authentication Token**.
2. Write an MQTT client on your Raspberry Pi to read the Bluetooth signal strength

The MQTT client is a C-program that reads the Bluetooth signal strength and sends the value to the IoT application on IBM Cloud. Likewise, the client receives commands from the IoT application and calls a script to perform the command.

2a. Determine the MAC address of your Bluetooth device

The device client needs to connect to a Bluetooth device by using the MAC address of the device. You can determine the MAC address of your Bluetooth device by issuing the following commands on your device (a Raspberry Pi in my case), which should provide similar results or output:

```
# bluetoothctl
[NEW] Controller 00:15:83:07:CC:97 traumcloud.enet [default]

[Bluetooth]# power on
Changing power on succeeded
[CHG] Controller 00:15:83:07:CC:97 Powered: yes

[Bluetooth]# scan on
Discovery started
[CHG] Controller 00:15:83:07:CC:97 Discovering: yes
[NEW] Device 3C:77:E6:EF:57:F5 3C-77-E6-EF-57-F5
[NEW] Device 00:0A:3A:2B:C6:ED MSI FT200
[CHG] Device 00:0A:3A:2B:C6:ED RSSI: -79
[CHG] Device 00:0A:3A:2B:C6:ED RSSI: -88

[Bluetooth]# agent on
Agent registered

[Bluetooth]# pair 00:0A:3A:2B:C6:ED
Attempting to pair with 00:0A:3A:2B:C6:ED
Request PIN code
[agent] Enter PIN code: 1234
[DEL] Device 3C:77:E6:EF:57:F5 3C-77-E6-EF-57-F5
[CHG] Device 00:0A:3A:2B:C6:ED Connected: yes
[CHG] Device 00:0A:3A:2B:C6:ED UUIDs:
  00001108-0000-1000-8000-00805f9b34fb
  0000111e-0000-1000-8000-00805f9b34fb
[CHG] Device 00:0A:3A:2B:C6:ED Paired: yes
Pairing successful

[Bluetooth]#exit
```
In my example, I know that I am looking for a **MSI FT200 headset**, and I see that its MAC address is **00:0A:3A:2B:C6:ED**. Note the MAC address for your device.

### 2b. Code the MQTT core

After the Mosquitto library is initialized, the program sets the callbacks for the events that it is interested in and connects to the Internet of Things Platform service in IBM Cloud. The initialization code essentially follows the example code from the Mosquito website:

```c
mqtt = mosquitto_new(cfg.deviceid, 1, NULL);
if(!mqtt){
    tprintf("Cannot Initialize MQTT library\n");
    mosquitto_lib_cleanup();
    return 1;
}
mosquitto_connect_callback_set(mqtt, bt_connect_callback);
mosquitto_disconnect_callback_set(mqtt, bt_disconnect_callback);
```

Then, you add Mosquitto callbacks to MQTT events:

```c
mosquitto_log_callback_set(mqtt, bt_log_callback);
mosquitto_message_callback_set(mqtt, bt_message_callback);
mosquitto_subscribe_callback_set(mqtt, bt_subscribe_callback);
dprintf("now setting username and password to %s/*************\n", cfg.username);
sts=mosquitto_username_pw_set(mqtt, cfg.username, cfg.password);
if(sts) {
    tprintf("Error setting username/password\n");
    return sts;
}
dprintf("now connection using host=%s port=%d\n", cfg.host, cfg.port);
sts = mosquitto_connect_bind(mqtt, cfg.host, cfg.port, 60, NULL);
if(sts) {
    tprintf("Error connecting to %s:%d as %s\n", HOST, PORT, IDFMT);
    return sts;
}
```

Then, the program subscribes when the connect callback indicates a proper connection:

```c
void bt_connect_callback(struct mosquitto *mqtt, void *obj, int result) {
    char buf[1024];
dprintf("conn cb %d 's\n", result, mosquitto_connack_string(result));
is_connected=(result==CONN_OK);mosquitto_subscribe(mqtt, NULL, cfg.subscription, 0);
}
```

The main program runs in a loop, which periodically sends the Bluetooth signal strength.

```c
while (is_connected != CONN_ERR) {
    if (is_connected == CONN_OK) publish_bt(mqtt);
    else dprintf("waiting for connection to come up\n", "");
sleep(cfg.sleep);
}
```
2c. Call an external script

The Mosquitto library takes care of receiving commands and calls the appropriate callback function. In our case, we call an external script to do the real work. This script enables us to be flexible with different kinds of actuators to control the appliances.

```c
void bt_message_callback(struct mosquitto *mqtt, void *udata, const struct mosquitto_message *message) {
    dprintf("msg cb topic='%s' payload='%s'\n", message->topic, message->payload);

    /*
    * A execute script btclient.d/command.sh
    */
    int pid,sts;
    pid = fork();
    switch (pid) {
    case 0:         // child, execute shell script
        setenv("TOPIC", message->topic, 1);
        setenv("PAYLOAD", message->payload, 1);
        sts=execl("btclient.d/mqtt-command.sh", "mqtt-command", (const char*)NULL);
        tprintf("something went wrong with execl(), error='%s'\n", strerror(errno));
        break;
    case -1:        // error
        tprintf("something went wrong with fork(), error='%s'\n", strerror(errno));
        break;
    default:        // parent
        wait(&sts);
        break;
    }
}
```

By using a script, the MQTT client can be easily adapted to the hardware or smart devices that you use. I used a Gembird Silver Shield programmable power outlet strip, which lets me control the individual switching of four power sockets by using a USB with a Linux utility called `sispmctl` (which is available as a package in Fedora Linux).

```bash
#!/bin/bash
if [ "$TOPIC" == "iot-2/cmd/switch/fmt/text" ];then
    [ "$PAYLOAD" == "on" ] && arg="-o4"
    [ "$PAYLOAD" == "off" ] && arg="-f4"
    [ "$arg" != "" ] && sispmctl $arg
fi
```

Alternatively, I used a 433 MHz transceiver to turn on wireless power sockets by using the pilight framework. The pilight framework is a framework for home automation on the Raspberry Pi. You can investigate which power socket and project fits your needs.

2d. Modify the MQTT client configuration file

The MQTT client reads a configuration file at startup, which has the same name as the executable file, but appended with `.cfg`. This separate configuration file allows you to run different instances just by using a different name for the executable; for example, after `otherclient` is started, the MQTT client will read `otherclient.cfg` as its configuration file.

The configuration file defines:

- **btmac**: Use the MAC address of the Bluetooth device that you measure the signal strength to (not the built-in device of your Linux system).
• **subscription**: Use the topic that the client needs to subscribe to. In my example, `iot-2/cmd/switch/fmt/+`.

• **host**: The host name of your Internet of Things Platform service instance. Use the name that you used when you sent the "hello iot" message.

• **password**: The password is the *device authentication token*. Use the one that you noted when you registered the device.

• **username**: The user name is the literal string `use-token-auth`

• **typeid**: The typeid is the *device type* that you used when you registered the device. In my example, I used `proximity`.

• **organization**: Use your organization ID.

3. **Write a Node-RED app to sense the Bluetooth signal and turn on an electrical appliance**

The MQTT client sends the Bluetooth signal strength on a regular interval to the Internet of Things Platform service in IBM Cloud. Now, you need to write an application in IBM Cloud to process those values. We use Node-RED as the environment to create that app.

3a. **Create the Node-RED app and connect your device**

First, we will connect our device and make sure it is receiving messages.

1. In your IBM Cloud IoT app, click the **start coding** link in the navigation pane to the left.
2. In the upper part of the screen, find the URL for your running app. Click that URL.
3. Click the large red button to open your Node-RED flow editor.
4. A sample application is displayed in the flow editor. Select all the elements in the flow and delete them.
5. Drag an `ibmiot` input node to the workspace, and configure it by double-clicking it. Use the API Authentication Token that you created earlier to gain access to your device data as shown in the illustration below.
6. Select **All** for *device type*, *device ID*, *event*, and *format*.
7. To test your device, start the device client on your Linux system. Make sure that some messages are sent to the Watson IoT Platform.
8. Drag a debug output node on to your worksheet, and connect it to your ibmiot node.
9. Click Deploy.
10. Click the Debug tab to the right to see debug messages coming in.

11. Move your Bluetooth device back and forth, and verify that the RSSI min/max values change.

3b. Write the Node-RED app to process the Bluetooth device data

After you verify that your device is connected, you can write the Node-RED application.

1. We start with an initializing function. Drag an Inject node, a function node, and a debug output node on to your workspace and connect them in this sequence.

2. Configure the Inject node with payload blank, check inject once at start, and give the node a meaningful name (for example, AppSetup).

3. Configure the function node with the following code, and give it a meaningful name. You need to give rssimin a value that is between the noted min and max values that you discovered previously for your Bluetooth device.
// bluetooth related
context.global.bbtime=0;
context.global.rssimin=-32;
context.global.rssi=0;
context.global.inBeacon=false;

// Timer related
context.global.lastCmd="";
context.global.nextSwitch=0;
context.global.delay=30*1000; // 30 secs

// will be used later with gps
context.global.gpstime=0;
context.global.gps={lat: 0.0, lng:0.0};
context.global.inFence=true;
msg.payload="App initialized";
return msg;

4. Configure the Debug node to output msg.payload and give it a meaningful name.
5. Deploy the code. You should see an "App initialized" message in the Debug tab.
6. Drag an inject node, an ibmiot input node, two function nodes, an ibmiot output node, and a debug node to your workspace.
7. Configure the first function node with this code:

```javascript
context.global.rssi=msg.payload.d.rssi
context.global.inBeacon=context.global.rssi>context.global.rssimin;
return msg;
```

8. Configure the second function node with the code below:

```javascript
//
// determine when to send a switch command
// A command is sent when a grace period has expired since last switch (to protect the
// appliance) and if the state of the appliance
// is to be changed

now = Date.now();
dbg={ payload:""};
var cmd=(context.global.inBeacon &&
context.global.inFence)?"on":"off";
if (now > context.global.nextSwitch &&
context.global.lastCmd != cmd) {
// we need to switch now
context.global.nextSwitch+=now+context.global.delay;
context.global.lastCmd=cmd;
msg.payload=cmd;
dbg.payload="switch appliance '+cmd+'";
} else {
  msg=null
dbg.payload="not switching appliance";
}
return [ msg, dbg ];
```
9. Configure the `ibmiot` node as in the picture below.

10. Finally, wire the nodes together:

3c. Test the Node-RED app

1. Deploy your app. You should see the "App initialized" message.
2. To test the flow, click the `timestamp` node to inject a message, if your device is not yet connected. If your device is connected, you should see periodic "not switching appliance" messages.
3. Slowly move the Bluetooth device closer and further away from your Linux system and you'll see that the debug messages that are trying to turn on and off the appliance. At the same time, you should see the on or off commands that are arriving at the client program that is running on your Raspberry Pi and the script that is configured to turn on or off light or air conditioning (A/C) is called. Be aware that there is a rate limit of one switch per 30 seconds; you might want to lower the `context.global.delay` parameter for testing purposes.

4. Add flow to Node-RED app to accept and process GPS data

So far, we created an app that turns on an appliance based on the proximity of a Bluetooth device. Next, we will enhance the app to use a specific geographic location that is provided by a GPS device to determine when to turn on an appliance.

While GPS devices today communicate by using a variety of protocols, for this tutorial and sample app, I assume that the GPS sends HTTP requests with the latitude and longitude pair as a message payload.
4a. Create an HTTP endpoint for the GPS device

In your Node-RED flow editor, drag an HTTP-input node on to your workspace. Double-click it to configure it with these values:

- **Method**: POST
- **URL**: /gps
- **Name**: GPS Input [POST] /gps

This HTTP node accepts POST requests on a URL like this one: `http://<dwexample-iot>.mybluemix.net/gps`. POST data is available as a message object in the Node-RED flow. The latitude and longitude values from the GPS device are further processed against a geo-fence.

The control of appliances by geo location is not bound to one particular location. Usually the control takes place by entering or leaving a defined geo-fence. A fence in our application is represented as an array of latitude and longitude pairs as JSON object, such as this one:

```json
[{
  "lat":43.7074340290373,"lng":7.28219747543335,
  ...
  ...
  ...
  "lat":43.705196002070636,"lng":7.284321784973145},
  "lat":43.706995349373884,"lng":7.284793853759766}
]
```

4b. Retrieve the geo-fence from the Cloudant database

The array of latitude and longitude pairs is stored in a Cloudant database that is part of the Internet of Things Platform Starter that we used to create our app. If a Cloudant database service does not exist in your app, go to the IBM Cloud catalog, find the Cloudant NoSQL DB service, and create a service.

Upon receiving a message with GPS coordinates, our app needs to retrieve the fence definition from the database, and then determine whether the just sent location is inside or outside the fence. The database node overrides the payload element of the message, so we need to temporarily save the HTTP request body. We use the global attributes from Node-RED as a temporary store for this HTTP request.

In your Node-RED flow editor, add a **function** node, and add the following JavaScript code to the node:
context.global.gps=msg.payload;
context.global.gpstime=Date.now();
msg.save=msg.payload;
msg.payload="fence;"
return msg;

The actual database query is done by a Cloudant node from the storage section of the palette. Drag the node with the little cloud to the left, which is the Cloudant query node.

4c. Determine whether the GPS coordinates are in the geo-fence

The output of the Cloudant node is fed in to another function node, where a JavaScript snippet determines whether the GPS coordinates are inside or outside the fence. The algorithm is taken from the point-in-polygon repo from substack on GitHub. The function node records the interface-state in a global structure member.

The output of the function node is fed to these nodes:

- An HTTP response node that returns the proper status to the GPS devices
- A node that brings the Bluetooth measurement together with the GPS tracking

This node implements the function turn on the A/C if either GPS is inside the fence or the car is close enough to the Bluetooth sensor, but not if the last switch is less than <n>minutes ago. By adding the last condition, the app avoids turning the A/C on or off at short intervals while the car is driving at the edge of the fence. The code that implements the function is similar to the following code:

//
// determine when to send a switch command
// A command is sent when a grace period has expired since last switch (to protect the appliance) and if the state of the appliance is to be changed
now = Date.now();
var cmd=(context.global.inBeacon || context.global.inFence)?"on":"off";
if (now > context.global.nextSwitch && context.global.lastCmd != cmd) {
    // we need to switch now
    context.global.nextSwitch=now+context.global.delay;
    context.global.lastCmd=cmd;
    msg.payload=cmd;
    msg.format="text";
} else {
    msg=null
}
return msg;

The final Node-RED application looks like the flow in Figure 2. The light gray and dark gray nodes are the nodes that you created in Steps 1 — 4.

**Figure 2. Final Node-RED app**

4d. Verify the geo-fence functions

Let’s do a simple test of the new geo-fence functions with simulated GPS coordinates that we send by a curl command.

1. In the IBM Cloud dashboard, double-click your Cloudant DB service.  
2. Click **Launch** to open the dashboard for your Cloudant database:  
3. In your database dashboard, insert a new document:
4. Click **Save** to store the document.

5. Test the Node-RED flow with two simulated GPS coordinates (a lat/lng pair that is within the fence and a lat/lng pair that is clearly outside the fence) by issuing this curl command (and specifying your sample app name):

   ```bash
   curl -XPOST -d 'lat=1&lng=26' dwexample-iot.mybluemix.net/gps
   {
   "bttime": 0,
   "rssimin": -32,
   "rssi": 0,
   "inBeacon": false,
   "lastCmd": "on",
   "nextSwitch": 1447771815039,
   "delay": 5000,
   "gpstime": 1447771815841,
   "gps": {
   "lat": "1",
   "lng": "2"
   },
   "inFence": false,
   }
   ``

   Review the messages in the **debug** tab in your Node-RED flow editor.
5. Add a dashboard to edit the GPS fence and set a threshold for the beacon

So far we have controlled the A/C by two different ways of getting a car's location, by either:

- Sensing a Bluetooth beacon at a parking lot or in a garage
- Tracking the car by GPS coordinates

Now, in our sample app, we still need to manage these two approximation devices. We need to set the threshold for the beacon and also define the fence more realistically. To manage these devices, we will add a web page that is an administration UI and add two new flows to our back-end app that return current operating values or set new operating values. See Figure 3.
Figure 3. A dashboard for the Bluetooth sensitivity and GPS visualization

The sequence diagram in Figure 4 depicts the components and their interactions that are needed to create the dashboard. The dashboard supports these use cases:

- Continuous monitoring of the GPS coordinates and Bluetooth sensitivity
- Editing and changing the Bluetooth sensitivity
- Editing and changing the geo-fence

The dashboard is implemented as an HTML page and uses an Ajax-request to get data from the back-end application. The left lifelines in Figure 4 are implemented as flows in Node-RED.
Figure 4. A dashboard for the Bluetooth sensitivity and GPS visualization

5a. Create the dashboard web page

You can download the files that you need to build the dashboard from my car-meets-home repo on GitHub. You will import these files into your IBM Cloud IoT app.

1. In your IBM Cloud dashboard, select your IoT application, and click Edit Code. IBM Cloud DevOps services open. It might take a while the first time you click it, as it generates a new project for you.

2. In the navigation bar to the left, identify the public folder. Right-click on the public folder and select Import.
3. In the open dialog, navigate to the *dash.html* file, and click **Open**. The file *dash.html* appears under the public directory.

4. Right-click again on the public folder, and select **New > Folder**. Name the new folder *js*.

5. Right-click the *js* folder, and import the *dash.js* file.

6. Right-click the *css* folder, and import the *dash.css* file.

7. Right-click the *images* folder, and import the .png files.

These files comprise the dashboard app. You can open your dashboard by appending /*dash.html* to your application route (for example, <your name>.mybluemix.net/dash.html).

### 5b. Add APIs to the flow to serve the Ajax requests

To make the dashboard work, we need to add some APIs to our Node-RED flow. The dashboard gets the status of the back-end application with a **HTTP-GET** request to */status*. Likewise, the new *rssimin* value (the sensitivity) is sent with a **HTTP-POST** request to */status*. The geo-fence is retrieved by a **GET** to */fence*, while the fence is stored with a **POST** to */fence*.

Create the flows, and add the code to the functions:

1. Add flows for status retrieval and update like in the picture below:

   ![Flow diagram](image1.png)

2. Add the following code to the node named **Add global as payload**:

   ```javascript
   msg.payload = context.global;
   msg.payload.jt = Date.now();
   return msg;
   ```

3. The following JavaScript code in the web page takes the Bluetooth and GPS values and displays them in text form, graphically as a gauge meter for Bluetooth values, and as a position on a map for GPS values. The core lines of code to display the data are:

   ```javascript
   //
   // get status and display methods
   //
   this.getData = function () {
   $.ajax({
   url: "status",
   success: thiz.getDataOK,
   error: thiz.getDataErr,
   dataType: "json"
   ```
4. The dashboard contains a few more lines of code that cause changed values to be displayed in red between data polls.

The button named **Enter new value for RSSI Min** takes the entered values for RSSI and sends the data to the Node-RED API.
5. The Node-RED API sets the RSSI value (Bluetooth sensitivity) in the global variable structure.

```javascript
if (msg.payload.rssimin) context.global.rssimin=msg.payload.rssimin;
msg.payload=context.global;
return msg;
```

6. In the same manner, you can add UI elements for the delay time between switching operations, and set them for the flows similar to `rssimin`. Add flows for retrieving the geo-fence data as shown in the following picture:

```javascript
var storeFence=function() {
  var ll=fence.getLatLngs();
  var data={ fence: JSON.stringify(ll) };
  if (fencers._id !== undefined) data._id=fencers._id;
  if (fencers._rev !== undefined) data._rev=fencers._rev;
  // upload to bluemix
  $.ajax({
    url: "/fence2",
    data: data,
    contenttype: "application/json",
    method: "POST",
    /**
     * @callback
     */
    success: function( data ) {
      $('#upload').show();
      window.setTimeout(function() {
        $('#upload').fadeOut(2000);
      }, 2000);
      loadFence(); // to update _rev
    }
  });
};
```

7. Loading the geo-fence is implemented as two methods, the first to call the Node-RED API asynchronously and the second as a callback function. The geo-fence is loaded when the application starts or when the user pressed the `load` button.

```javascript
var loadFence=function() {
  // load the fence definition from bluemix
  // $('#download').show();
  $.ajax({
    url: "/fence",
    contenttype: "application/json",
    method: "GET",
    success: function( data ) {
      $('#download').fadeOut(1000);
      $('#received').fadeIn(1000);
      window.setTimeout(function() {
        $('#received').fadeOut(1000);
      }, 2000);
      loadFenceOK(data);
    },
    error: function(a, b) {
      thiz.error("Cannot load fence data, reason given: "+a+":"+b+"\n";
    }
  });
  loadFenceOK=function(data) {
```
var ll=[];
var i;
for (fencers=data; i<fencers.length; i++) {
  L.marker([fencers[i].lat, fencers[i].lng], {icon: reddot}).addTo(map).on('click', removeDot);
}
fence=L.polygon(ll, {color: 'red'}).addTo(map);

// position map to fit car & fence in browser window
ll=fence.getLatLngs();
var b=L.latLngBounds(ll);
b.extend(car.getLatLng());
map.fitBounds(b, {animate: true, duration: 2.0});
$('#save').prop( "disabled", ll.length <= 2 );
$('#load').prop( "disabled", false);
};

8. In addition, the dashboard contains a basic geo-fence editor. If you click a point on the map, the point is added to the polygon and defines the geo-fence area. If you click one of the red dots, the point is removed from the geo-fence polygon.

a. To remove the point from the polygon, a loop compares the coordinates of the coordinates in the polygon against the coordinates of the icon created with Leaflet. If they match, the point is removed from the polygon.

```javascript
// fence editor (very basic)

var removeDot=function(me) {
  var ll=fence.getLatLngs();
  var i, l=ll.length;
  for (i=0; i<l; i++) {
    var p=ll[i];
    var dx=p.lat-me.latlng.lat; // distance of a dot in fence to position clicked on map
    var dy=p.lng-me.latlng.lng;
    if (dy === 0 || dx === 0) {
      fence.spliceLatLngs(i, 1);
      l--;
      break;
    }
  }
  map.removeLayer(me.target); // delete dot clicked on;
  $('#save').prop( "disabled", l <= 2 );
};
```

b. Adding a point to the polygon needs a bit more effort, as the code needs to find the two polygon points that are the best to add the new point between them. The dashboard app runs the following steps:

- Find the point closest to the coordinates of the point that was clicked on the map.
- From the two neighbors, determine the one closest to the point on the map.
- Insert the new point between the two points found.

The JavaScript code, which takes care of (nearly) empty polygons and the state of the save button, looks similar to the following code:

```javascript
var addDot=function(me) {
  // see which existing dot is close to the mouse event me as p1
  // find the adjacent nodes of p1 closest to me
// insert me between them
var ll=fence.getLatLngs();
var i, l=ll.length;
var p0=me.latlng, p1=null, p2=null;
if (l < 2) {
    // there is no or only one dot in the fence - just add anew one
    fence.addLatLng([ok, good, me.latlng.lng]);
} else {
    var dist=0x07ffffff; // max 32bit signed int
    var p, dlat, dlng;
    var d=Math.sqrt(dlat*dlat+dlng*dlng);
    for (i=0; i<l; i++) {
        p=ll[i];
        dlat=p0.lat-p.lat;
        dlng=p0.lng-p.lng;
        d=Math.sqrt(dlat*dlat+dlng*dlng);
        if (d < dist) {
            p1=i;
            dist=d;
        }
    }
    // find closest adjacent
    p2=(p1+l-1)%l;
    p=ll[p2];
    dlat=p0.lat-p.lat;
    dlng=p0.lng-p.lng;
    dist=Math.sqrt(dlat*dlat+dlng*dlng);
    p=ll[(p1+l+1)%l];
    dlat=p0.lat-p.lat;
    dlng=p0.lng-p.lng;
    dist=Math.sqrt(dlat*dlat+dlng*dlng);
    if (d < dist) p2=(p1+l+1)%l;
    // special case: p1 and p2 are beginning and end of array
    if (p1 === 0 && p2 === l-1) {
        fence.addLatLng(p0);
    } else {
        fence.spliceLatLngs(p2, 0, p0);
    }
}
L.marker(me.latlng, {icon: reddot}).addTo(map).on('click', removeDot);
$('#save').prop( "disabled", l+1 <= 2 );

Note that a point in the polygon is implemented as a Leaflet Icon with a callback that is triggered when the user clicks on the icon. As a consequence, the point is removed from the polygon and the associated icon is removed from the map. This code allows you to determine easily whether a user click is to add or to remove a point to or from the geo-fence polygon.

You can see the demo dashboard running on IBM Cloud:
http://car-meets-home.mybluemix.net/.

Conclusion

In this tutorial, you learned how to create a IBM Cloud app, register a Raspberry Pi to Watson IoT Platform, get access to device data, and write a Node-RED app to process data that is coming in by using the MQTT protocol.
On the Raspberry Pi, you created an MQTT client that can report Bluetooth signal strength and receive commands from your IBM Cloud application.

We quickly and easily extended the basic application to add new functions like receiving GPS coordinates from an external device, deciding when to turn on appliances based on a combination of Bluetooth signal strength and a geo location inside or outside a defined geo-fence. Lastly, we even created a dashboard to define the geo-fence for the GPS control function and set the sensitivity of the Bluetooth sensor. The functions were added as APIs with only a few extra nodes in the Node-RED app.

Now, go show off your smarts and connect your car to your home.