DOMAIN NAME: CLOUD APPLICATION DEVELOPMENT

PROJECT NAME: TRANSFORM YOUR HOME INTO A SMART LIVING SPACE USING IBM CLOUD FUNCTION FOR IOT DATA PROCESSING.

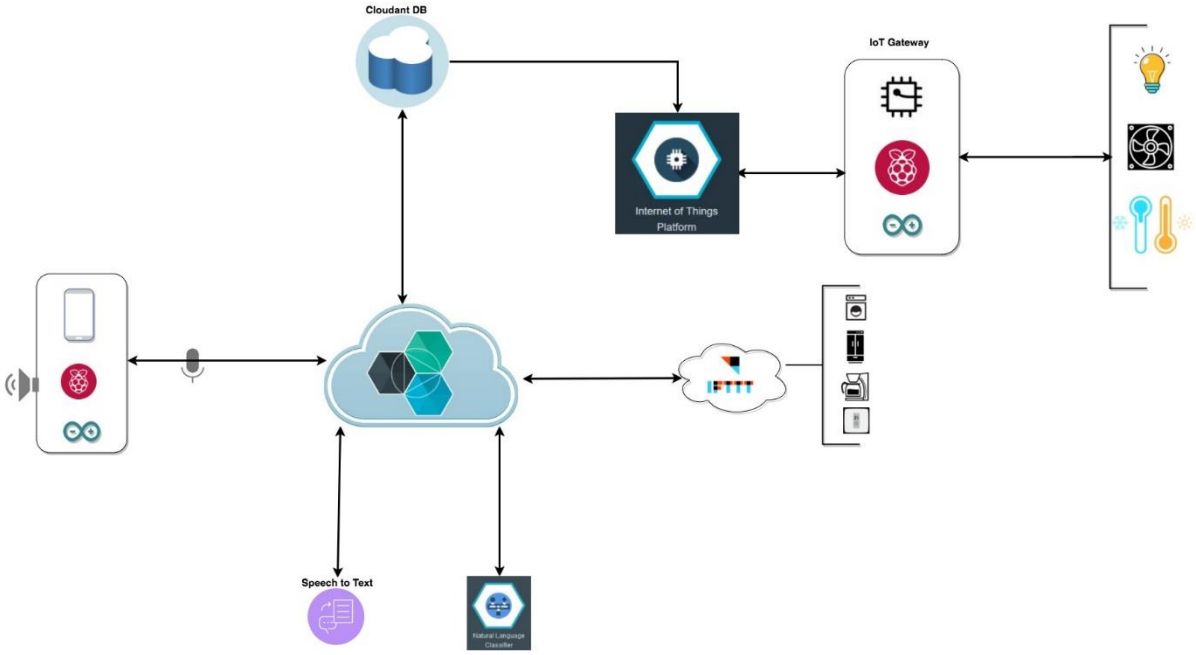
Phase 5 Submission Document

INTRODUCTION:

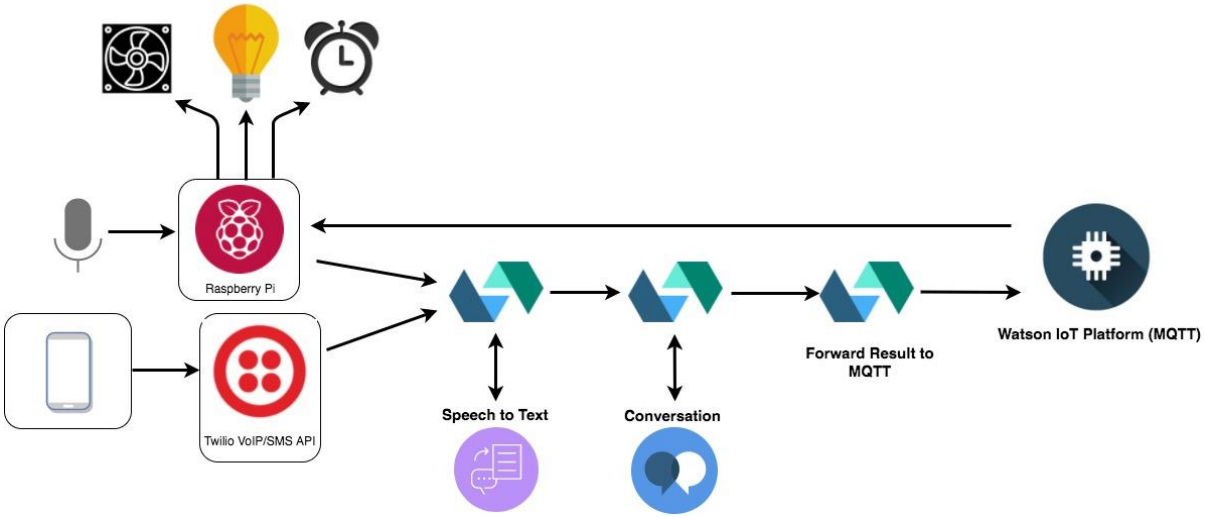
Over the past few years, we've seen a significant rise in popularity for intelligent personal assistants, such as Apple's Siri, Amazon Alexa, and Google Assistant. Though they initially appeared to be little more than a novelty, they've evolved to become rather useful as a convenient interface to interact with service APIs and IoT connected devices.

This series will guide users through setting up their own starter home automation hub using a Raspberry PI. This first blog post provides a step by step tutorial to create a RF circuit that'll enable the Raspberry PI to turn power outlets off and on. Once the circuit and software dependencies are installed and configured properly, users will also be able to leverage Watson's language services to control the power outlets via voice and/or text commands. Furthermore, we'll show how Openwhisk serverless functions can be leveraged to trigger these sockets based on a timed schedule, changes to the weather, motion sensors being activated, etc. We'll assume that the reader has a basic understanding of Linux and electronic circuits.

ARCHITECTURE DIAGRAM:



Architecture



ArchitectureV2

ARCHIYECTURE FLOW:

1. User says a command into the microphone, or sends a text to the Twilio SMS number
2. User input is captured and embedded in an HTTP POST request triggering an IBM Cloud Functions sequence
3. The first IBM Cloud Functions action in the sequence forwards the audio to Speech to Text service, and waits for the response
4. Transcription is forwarded to the second IBM Cloud Functions action
5. IBM Cloud Functions action 2 calls the Conversation service to analyze the user's text input, again waits for the response
6. Conversation service result is forwarded to final IBM Cloud Functions action
7. Final IBM Cloud Functions action publishes a entity/intent pair (fan/turnon for example) to the IoT MQTT broker
8. MQTT client subscribed on Raspberry Pi receives and interprets result
9. Raspberry Pi transmits corresponding RF signal to adjust outlet state

Steps:

- Connect And Configure Hardware
- Assemble RF Circuit
- Install Software Dependencies + Libraries
- Capture RF codes corresponding to wireless sockets

- Provision IBM Cloud Services
- Create Serverless Functions
- Deploy to IBM Cloud

This tutorial requires the following components:

- *Raspberry PI3*
- *Etekcity 433 MHz Outlets*
- *USB Microphone*
- *GPIO Ribbon cable + Breakout Board*
- *433MHz RF transmitter and receiver*
- *Electronic Breadboard*

BASIC COMPONENTS USED IN HOME AUTOMATION:

- Smart lightning.
- Smart thermostat.
- Smart lock.
- Smart security cameras.
- Speaker and voice assistant.
- Smart irrigation system.
- Smart garage door openers.
- Smart water leak detectors.

- Smart Garden system.
- Smart entertainment system.
- Smart Appliances.

Install software dependencies:

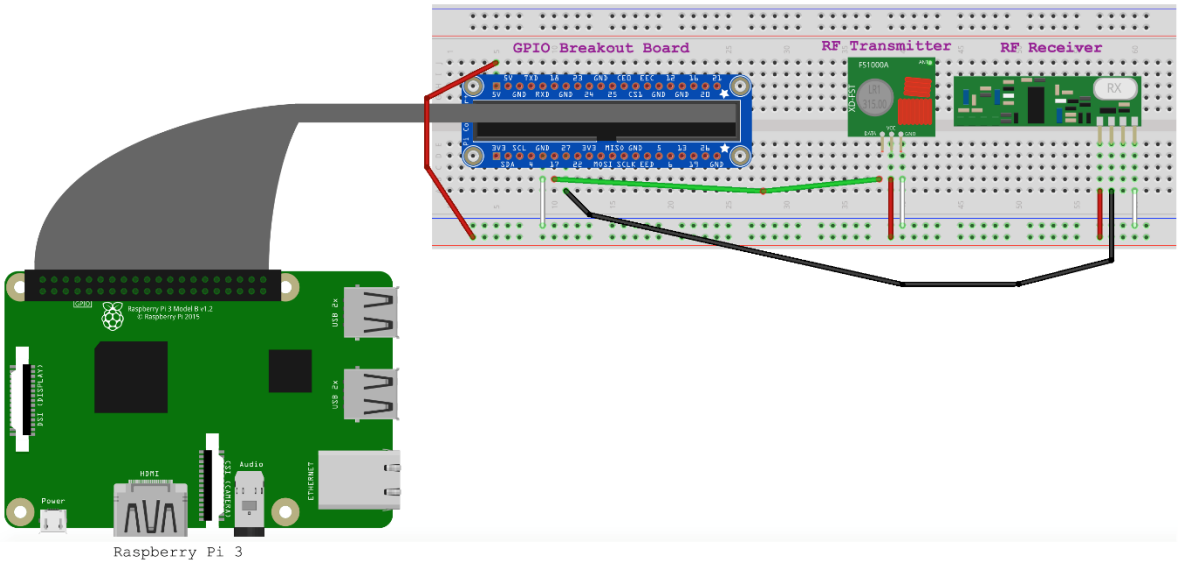
Login to Raspberry PI and install prerequisites for the wiringPi library. This library enables applications to read/control the Raspberry Pi's GPIO pins.

```
sudo apt-get -y update
sudo apt-get -y install git-core
git clone git://git.drogon.net/wiringPi
git pull origin master
./wiringPi/build
```

Ensure the wiringPi library is installed properly by running the following command.

```
gpio readall
```

RF CIRCUIT:



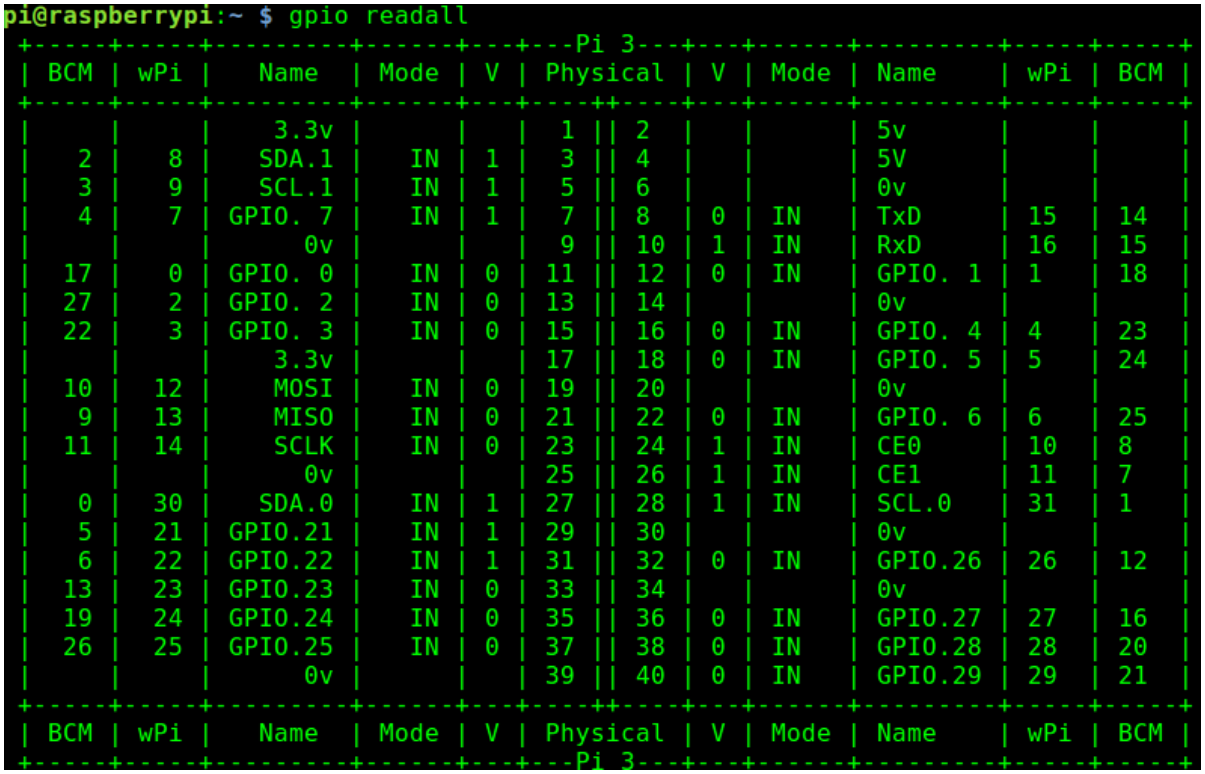
The red wire just left of the breakout board is responsible for bridging 5 volts from the Raspberry Pi to one of the breadboard's power rails. The additional red wires to the bottom right of the diagram supply those 5 volts from the power rail to the RF receiver and transmitter. Similar concept for the white wires, except those provide a negative charge, commonly referenced to as "ground". Next, we have the green wire that connects the Raspberry Pi's GPIO pin 17 to the transmitter's data pin, and the black wire connects the GPIO pin 27 to the receiver's data pin. The reason for this can be seen in the gpio readall output in image below, as the transmitter defaults to wiringPi pin 0 which maps to BCM 17, and the receiver defaults to wiringPi pin 2, which maps to BCM 27. These default pins can be changed by modifying either of the linked files in the 433Utils library, and recompiling the library.

Once the Raspberry Pi is connected to the circuit, we'll need to install dependencies to allow us to interact with the RF transmitter and receiver. This can be accomplished by running the install_deps.sh script.

The open source libraries that are being installed here are wiringPi and 433Utils. wiringPi enables applications to read/control the Raspberry Pi’s GPIO pins. 433Utils calls the wiringPi library to transmit and receive messages via the 433MHz frequency. In our case, each outlet has a unique RF code to turn power on and off. We’ll use one of the wiringPi utilities, titled “RFSniffer” to essentially register each of these unique codes. The 433MHz frequency is standard among

many common devices such as garage door openers, thermostats, window/door sensors, car keys, etc. So this initial setup is not limited to only controlling power outlets.

Once the script completes run gpio readall to ensure that wiringPi installed successfully. The following chart should be displayed.



```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 0 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
```

Pi GPIO output

Next, install 433Utils, which will call the wiringPi library to transmit and receive messages via the 433MHz frequency. In our case, each outlet has a unique RF code to turn power on and off. We'll use one of the wiringPi utilities, titled "RFSniffer" to essentially register each of these unique codes. The 433MHz frequency is standard among many common devices such as garage door openers, thermostats, window/door sensors, car keys, etc. So this initial setup is not limited to only controlling power outlets. This library can be installed by running the following commands on the Raspberry Pi.

```
sudo apt-get install build-essential

git clone git://github.com/ninjablocks/433Utils.git
```

```
cd 433Utils/RPi_utils

make
```

Now we will determine which RF codes correspond with the Etekcity outlets. Start by running.

```
sudo /var/www/rfoutlet/RFSniffer
```

This will listen on the RF receiver for incoming signals, and write them to stdout. As the on/off buttons are pressed on the Etekcity remote, the Raspberry Pi should show the following output if the circuit is wired correctly.

```
pi@raspberrypi:~ $ sudo /var/www/rfoutlet/RFSniffer

Received 5528835

Received pulse 190

Received 5528844
Received pulse 191
```

After determining the on/off signal for the RF sockets, place the captured signals into the /etc/environment file like so.

```
RF_PLUG_ON_1=5528835

RF_PLUG_ON_PULSE_1=190

RF_PLUG_OFF_1=5528844

RF_PLUG_OFF_PULSE_1=191
```

Now, plug in the associated socket, and run the following command to ensure the Raspberry Pi can turn the socket on and off. This command simply sends the RF code at the requested pulse length, which is to be provided as the -l parameter.

```
/var/www/rfoutlet/codesend ${RF_PLUG_ON_1} -l
${RF_PLUG_ON_PULSE_1}

/var/www/rfoutlet/codesend ${RF_PLUG_OFF_1} -l
${RF_PLUG_OFF_PULSE_1}
```

Now that we can control the sockets manually via cli, we'll move forward and experiment with different ways to control them in an automated fashion. Rather than writing and executing pipelines and complex automation logic on the Raspberry Pi, we'll utilize a serverless, event driven platform called Openwhisk. In this implementation, Openwhisk actions communicate with the Raspberry Pi via MQTT messages.

A Bluemix account is required to set up Openwhisk and the accompanying Watson services.

Audio Interface:

Once the Raspberry Pi is setup, we'll need to configure it to recognize audio input from the USB microphone. To ensure that audio is recorded and transcribed only as needed, we'll leverage a "Hotword" detection service named Snowboy, which listens for a specific speech pattern (Hello Watson, in this case), and begins recording once the hotword pattern is detected. The steps required to create a voice model can be found here.

- **Provision and Configure Platform Services**
- Watson Assistant
- Speech to Text
- Watson IoT Platform
- Twilio

A IBM Cloud Account is required to provision these services. After logging in, simply navigate to each of the links above, and select the Create Service button.

Watson Assistant:

The Watson Assistant service is used to analyze natural language and determine which action(s) to take based on the user input. There are two main concepts to understand here. The first are referred to as "Intents", which determine what the user would like the application to do. Next, we have "Entities", which provide context of where the intent should be applied. To keep things simple, we have two intents, one is titled "turnoff", the other "turnon". Next, we have 3 entities, which are household devices that we'd like to turn off and on in this case. This pre-trained data model can be uploaded to the provisioned Watson Assistant service through the UI. To initiate the upload, login to the IBM Cloud console. Next select the Watson Assistant service, and then the button titled Launch Tool.

Watson IoT Platform:

The Watson IoT Platform will be utilized as a MQTT messaging broker. This is a lightweight publish/subscribe messaging protocol that'll allow for various devices such as a Phone, Laptop, and Microphone to communicate with the Raspberry Pi. Once this service has been provisioned, we'll need to generate a set of credentials to securely access the MQTT broker. These steps are listed here

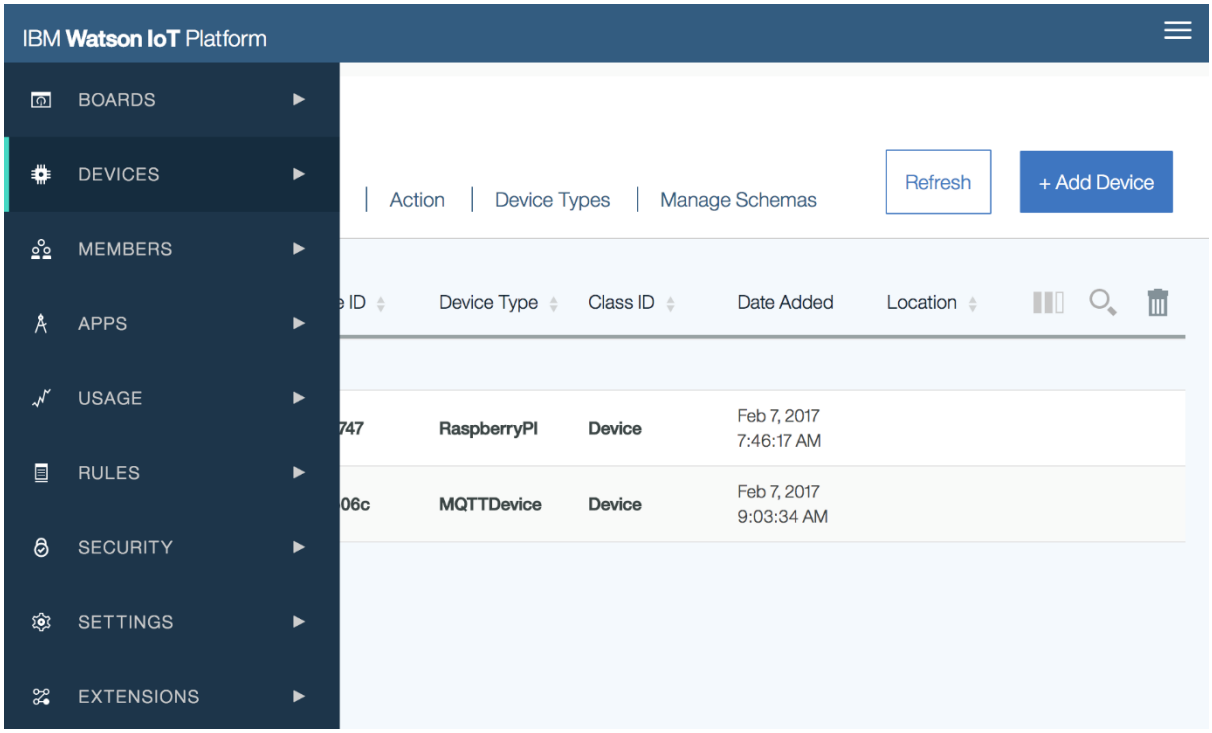
Add Device

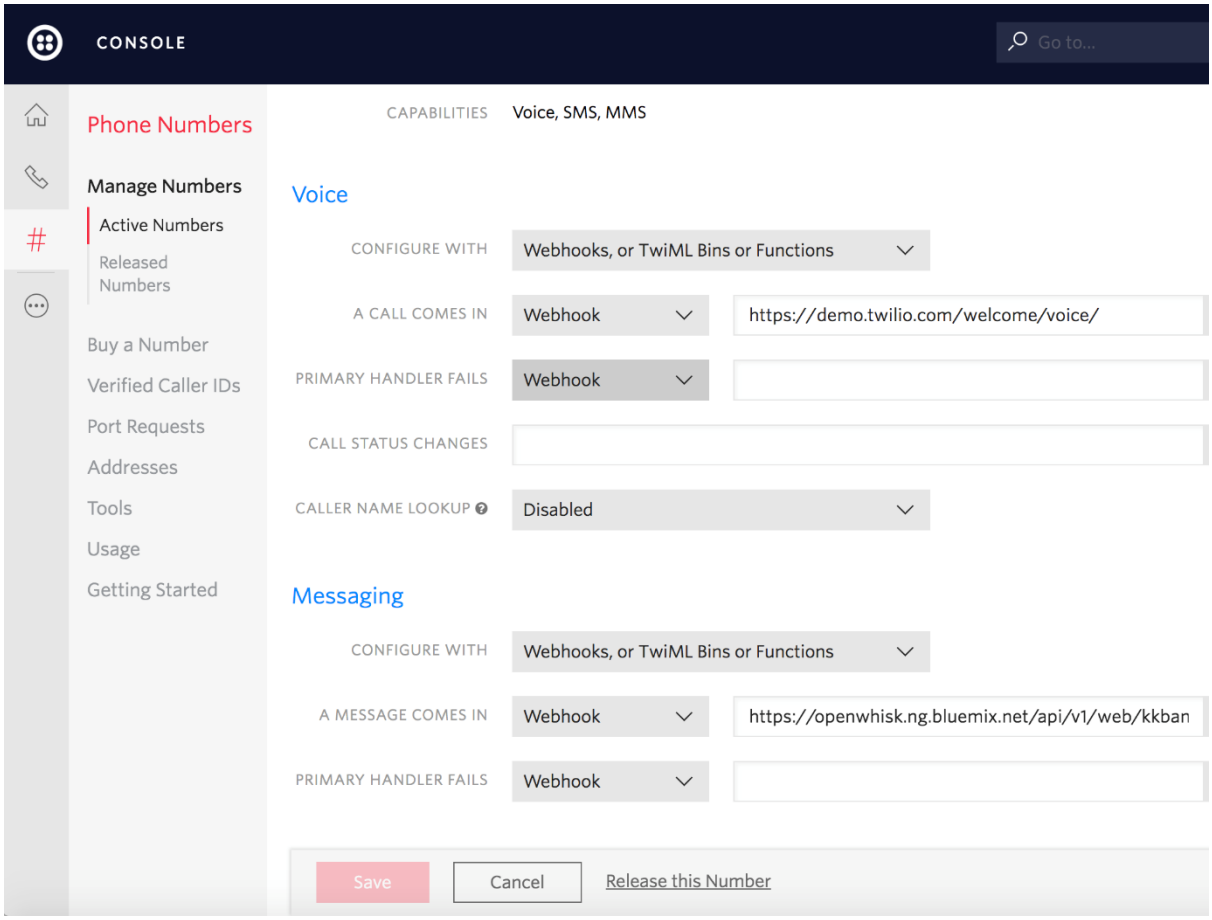
Choose Device Type

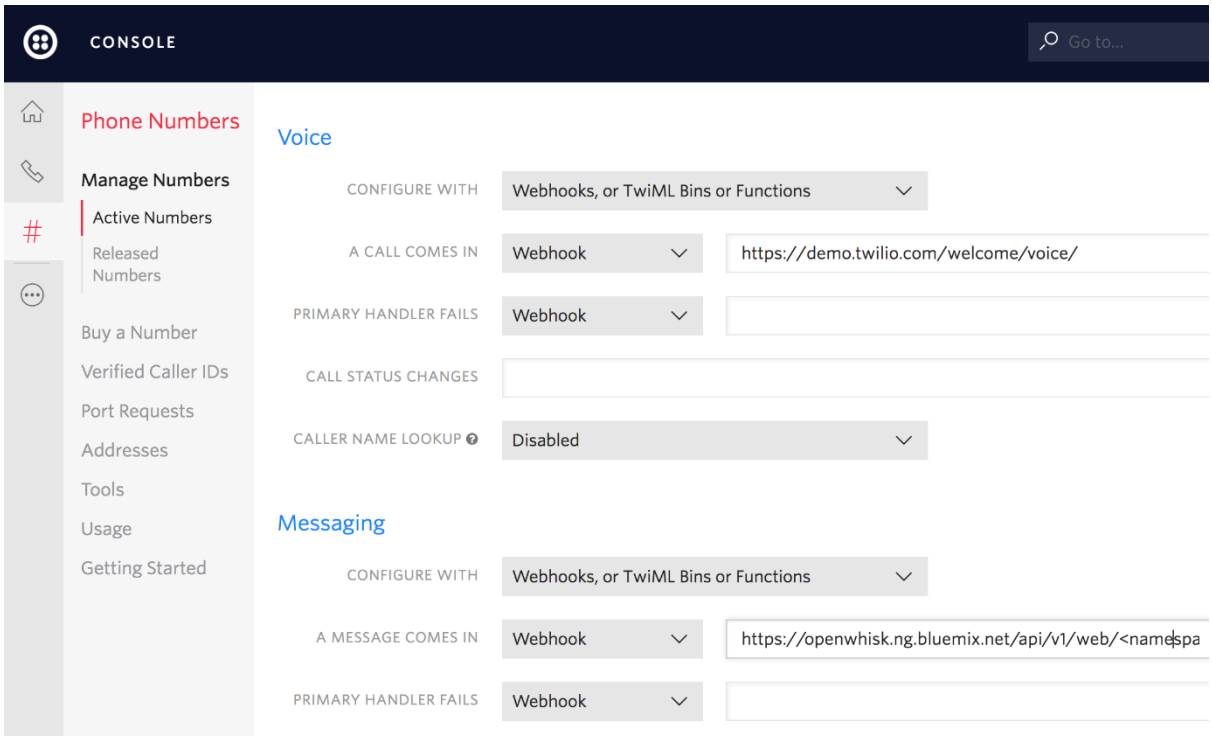
Choose Device Type

Or

Create device type




CONSOLE
Go to...
Phone Numbers
Manage Numbers
Active Numbers
Released Numbers
Buy a Number
Verified Caller IDs
Port Requests
Addresses
Tools
Usage
Getting Started
CAPABILITIES
Voice, SMS, MMS
Voice
CONFIGURE WITH
Webhooks, or TwiML Bins or Functions
A CALL COMES IN
Webhook
https://demo.twilio.com/welcome/voice/
PRIMARY HANDLER FAILS
Webhook
CALL STATUS CHANGES
CALLER NAME LOOKUP
Disabled
Messaging
CONFIGURE WITH
Webhooks, or TwiML Bins or Functions
A MESSAGE COMES IN
Webhook
https://openwhisk.ng.bluemix.net/api/v1/web/kkban
PRIMARY HANDLER FAILS
Webhook
Save
Cancel
Release this Number


CONSOLE
Go to...
Phone Numbers
Manage Numbers
Active Numbers
Released Numbers
Buy a Number
Verified Caller IDs
Port Requests
Addresses
Tools
Usage
Getting Started
Voice
CONFIGURE WITH
Webhooks, or TwiML Bins or Functions
A CALL COMES IN
Webhook
https://demo.twilio.com/welcome/voice/
PRIMARY HANDLER FAILS
Webhook
CALL STATUS CHANGES
CALLER NAME LOOKUP
Disabled
Messaging
CONFIGURE WITH
Webhooks, or TwiML Bins or Functions
A MESSAGE COMES IN
Webhook
https://openwhisk.ng.bluemix.net/api/v1/web/<namespa
PRIMARY HANDLER FAILS
Webhook

Create Device Type

General Information

Name homeAutomation

The device type name is used to identify the device type uniquely, using a restricted set of characters to make it suitable for API use.

Description Enter description

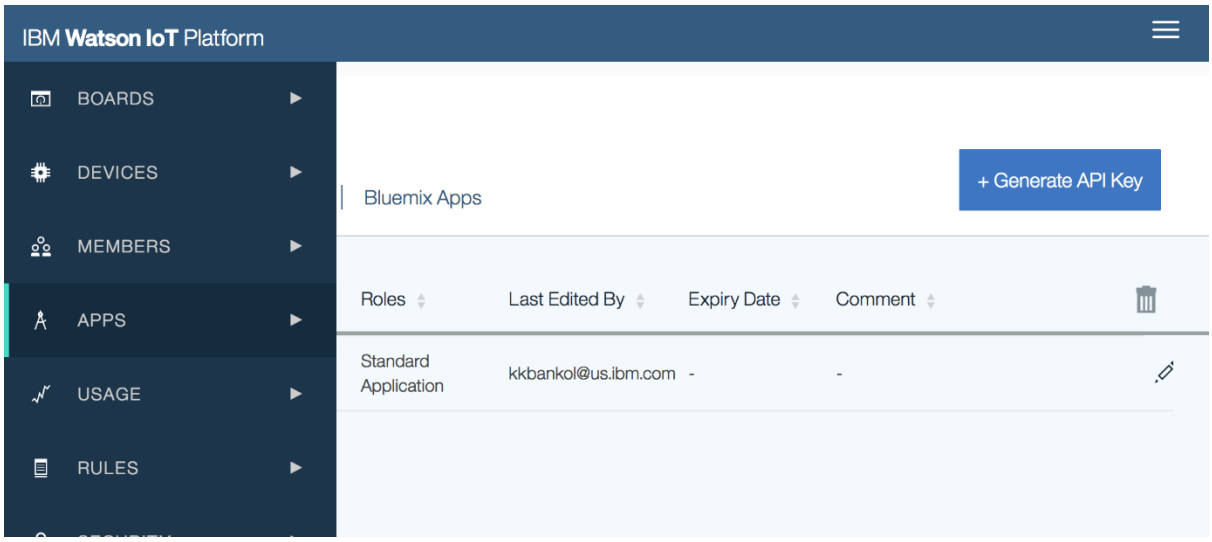
The device type description can be used for a more descriptive way of identifying the device type.

Your Device Credentials

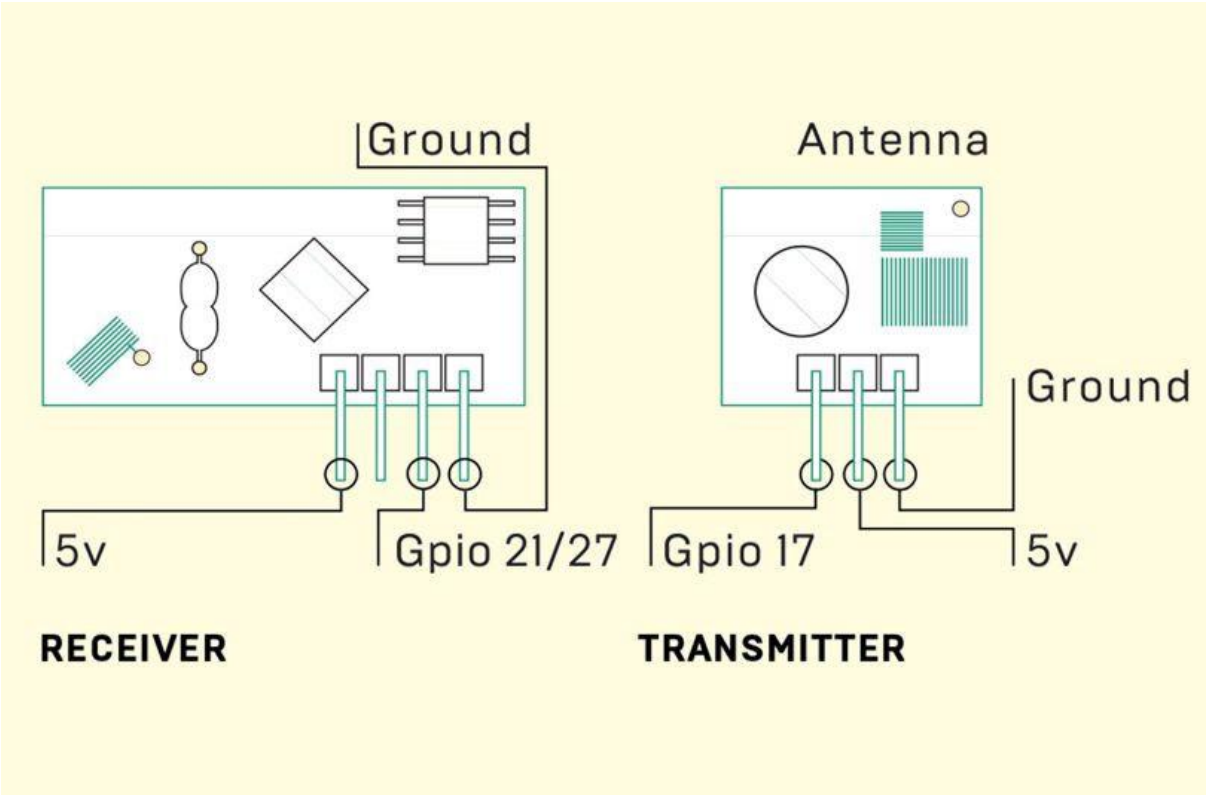
You have registered your device to the organization. To get it connected, you need to add these credentials to your device. Once you've added these, you should see the messages sent from your device in the 'Sensor Information' section on this page.

Organization ID	**uycwmh**
Device Type	**homeAutomation**
Device ID	**FF9C7392**
Authentication Method	**token**
Authentication Token	**(KtqtXfGWl+Kb!*uxe**

Authentication tokens are non-recoverable. If you misplace this token, you will need to re-register the device to generate a new authentication token.




```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 0 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | IN   | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
```


Ground
5v
Gpio 21/27
RECEIVER
Antenna
Ground
Gpio 17
5v
TRANSMITTER

Internet of Things / Internet of Things

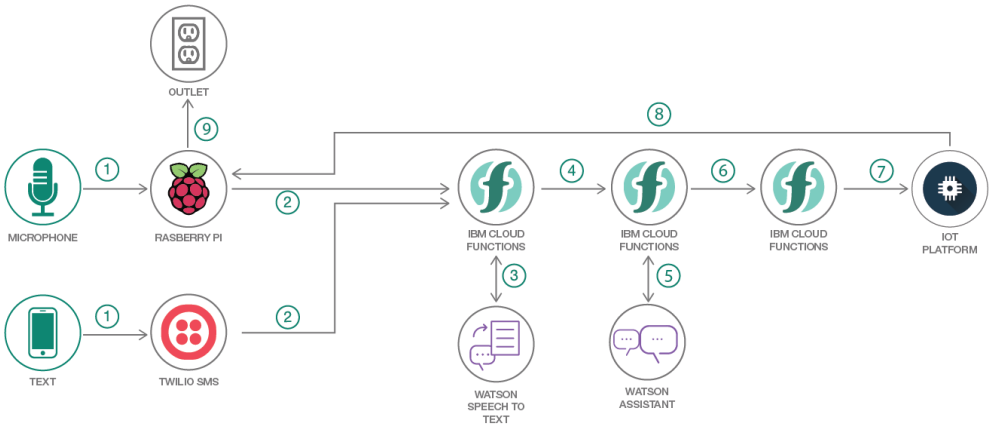
Internet of Things

Details

Welcome to Watson IoT Platform

Securely connect, control, and manage devices. Quickly build IoT applications that analyze data from the physical world.

Launch Docs



Service name:

Speech to Text-xl

Credential name:

Credentials-1

Features

- **Available Languages**

  English (US), English (UK), Japanese, Arabic (MSA, Broadband model only), Mandarin, Portuguese (Brazil), Spanish, French (Broadband model only)

- **Telephony (narrow-band) models**

  There will be an add-on charge of $0.02 USD per minute when used in conjunction with the standard plan.

- **Keyword Spotting (BETA)**

  Optional ability to search for one or more keywords in the audio stream. The returned metadata includes the beginning time, end time and confidence score for each instance of the

- **Metadata**

  Receive a metadata object in the JSON response that includes confidence score (per word), start/end time (per word), and alternate hypotheses / N-Best (per phrase). A new option for returning word alternatives per (sequential) time intervals is now available.

- **Mobile SDKs (BETA)**

  Mobile SDKs are now available to enable native interaction on iOS and Android devices.

- **SoftBank**

  A localized version of this Watson service is available in Japan. Visit the following link for details: http://www.softbank.jp/biz/watson

ate Monthly Cost
Calculator

Create

speech to text

Filter

Services

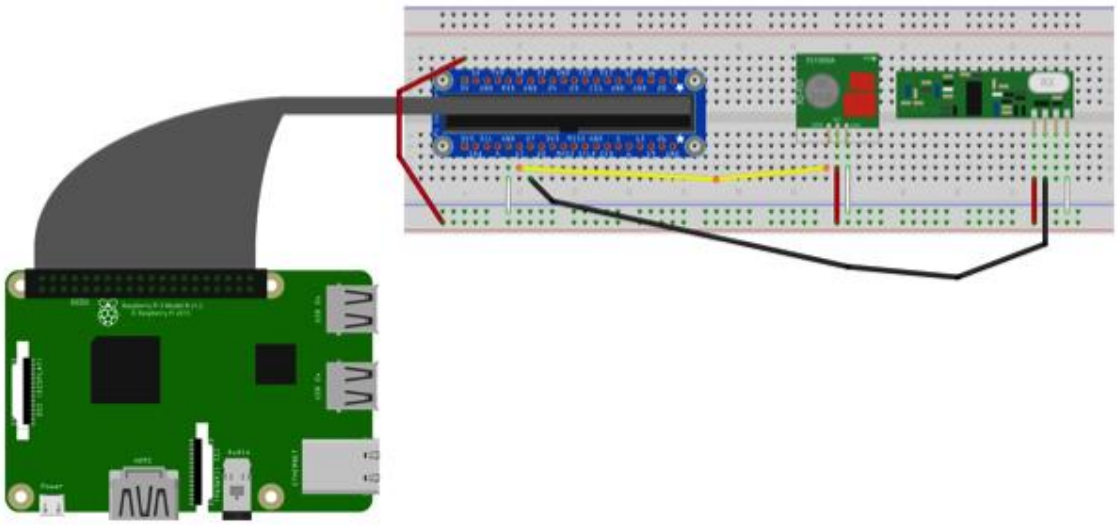
Watson

Build cognitive apps that help enhance, scale, and accelerate human expertise.

Speech to Text

Low-latency, streaming transcription

IBM



IBM Cloud Functions:

Rather than writing and executing pipelines and complex automation logic on the Raspberry Pi, we'll utilize a serverless, event driven platform called IBM Cloud Functions. In this implementation, IBM Cloud Functions actions forward their results to the Raspberry Pi as MQTT messages. IBM Cloud Functions is a serverless framework which has the ability to bind snippets of code to REST API endpoints. Once these have been created, they can be executed directly from any internet connected device, or they can respond to events such as a database change or a message coming in to a specific MQTT channel. Once these snippets, or "Actions" have been created, they may be chained together as a sequence, as seen above in the architecture diagram.

To get started, we will create a sequence that consists of three actions. The first action will transcribe an audio payload to text. The second action will analyze the transcribed text result using the Watson Assistant service. This analysis will extract the intent behind the spoken message, and determine what the user would like the Raspberry Pi to do. So, for example, if the user says something along the line of "Turn on the light" or "Flip the switch", the NLC service will be able to

interpret that. Finally, the third action will send a MQTT message that'll notify the Raspberry Pi to switch the socket on/off.

The speech to text action is already built in to IBM Cloud Functions as a public package, so we'll just need to supply our credentials for that service. Moving forward, we can create the additional actions with the following commands.

cd serverless-home-automation/ibm_cloud_functions

wsk action create conversation conversation.js

wsk action create iot-pub iot-pub.py

Once the actions are successfully created, we can set default service credentials for each of the actions. Otherwise we'd have to pass in the service credentials every time we'd like our actions to call the Watson services. To obtain these credentials, click each provisioned service in the IBM Cloud dashboard, and then select the View credentials dropdown.

Clock.js:

```
/**
 *
 * main() will be invoked when you Run This Action.
 *
 * @param Cloud Functions actions accept a single parameter,
 *        which must be a JSON object.
 *
 * In this case, the params variable will look like:
 *     { "message": "xxxx" }
 *
 * @return which must be a JSON object.
 *         It will be the output of this action.
 *
 */

// Beforehand run both
// wsk trigger create timer
// wsk trigger create alarm
function main(params) {
```

```
  return new Promise(function(resolve, reject) {

    var openwhisk = require('openwhisk');
    const options = {apihost: 'openwhisk.ng.bluemix.net', api_key: '<redacted>'};
    const ow = openwhisk(options);

    // expecting following in params
    // params['sys-time'] = "00:05:00"
    // params['sys-date'] = "2017-11-13"
    // params['type'] = "timer" // (optional, used to tell difference between 5 minute
timer and alarm at 5:00 am, both read as 05:00:00)
    // #  ┌──────────────────────── minute (0 - 59)
    // #  │ ┌──────────────────────── hour (0 - 23)
    // #  │ │ ┌──────────────────────── day of month (1 - 31)
    // #  │ │ │ ┌──────────────────────── month (0 - 11)
    // #  │ │ │ │ ┌──────────────────────── day of week (0 - 6) (Sunday to Saturday)
    // #  │ │ │ │ │
    // #  │ │ │ │ │
    // # 0 0 1 0 *
    // first value is optional, references seconds
    // */20 * * * * * // every 20 seconds

    // var params = {'sys-time': '00:00:05', 'sys-date': '2017-11-13', 'clock': 'timer'}
// 5 second timer, for testing
    var time = inputParams['sys-time'].split(':')
    var date = inputParams['sys-date'].split('-')

    var zero = function(num) {
        return ((Boolean(Number(num))) ? ("*/" + Number(num)) : "*")
    }

    var genCron = function() {
        if (params.clock.includes("timer")) {
            var cron = (Boolean(Number(time[2])) ? (zero(time[2])) : "") + " " +
zero(time[1]) + " " + zero(time[0]) + " " + "* * *"
        }
        else if (params.clock.includes("alarm")) {
            var cron = time[1] + " " + time[0] + " " + date[2] + " " + date[1] + " " + "*
*"
        }
        return cron
    }
```

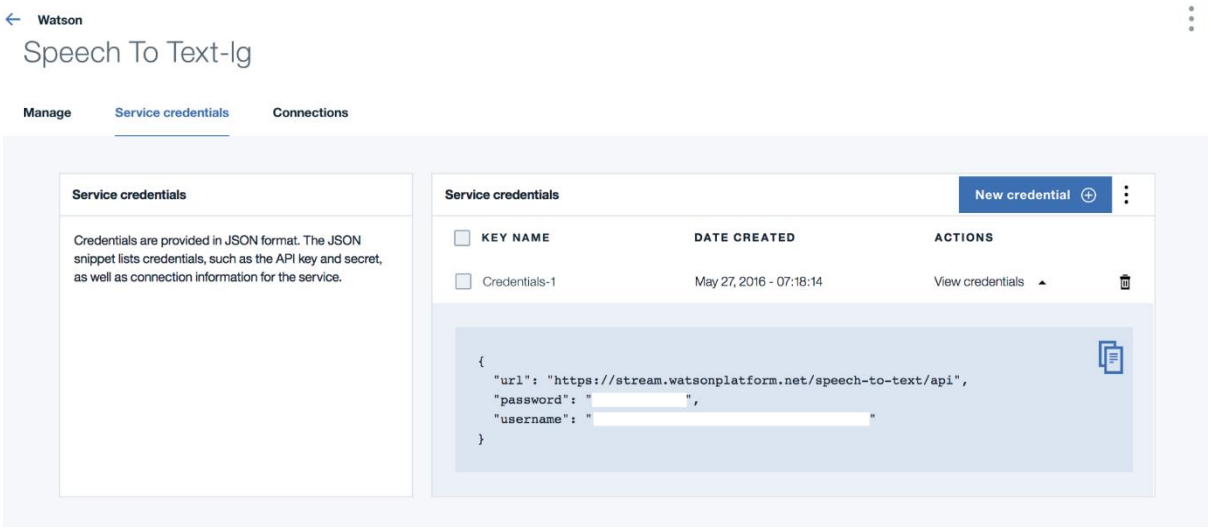
```
   //     ow.actions.invoke({actionName:    params.intents[0]['intent'],    params:
{entites: "params.entities"}}).then(
   //     console.log("alarm set")
   // )
   //var feedParams = {cron: genCron(), maxTriggers: 1, trigger_payload:
{message: params['type'] + " set for " + params['sys-time'] }}

   var feedParams = {
      cron: genCron(),
      trigger_payload: {message: params['clock'] + " set for " + params['sys-time']
},
      maxTriggers: 1
   }
   var name = '/whisk.system/alarms/alarm'
   var trigger = params.clock + '_' + time.join("") + '_' + date.join("")
   ow.triggers.create({name:  trigger, params: {maxTriggers: 1 }})
   ow.feeds.create({name, trigger, params: feedParams }).then(package => {
    console.log('alarm trigger feed created', package)
    return {"payload": "alarm for " + JSON.stringify(params) + " set"}
   }).catch(err => {
    console.error('failed to create alarm trigger', err)
   })

   // ow.triggers.create({
   //     name: "alarm1",
   //     feed: "/whisk.system/alarms/alarm"
   // })


   // return Promise.all(actions).then(function (results) {
   //     console.log(results);
   //     return resolve({payload: "All OK"});
   // });
  });
}
```

Conversation.js:

```
/**

  *

  * main() will be invoked when you Run This Action.
```

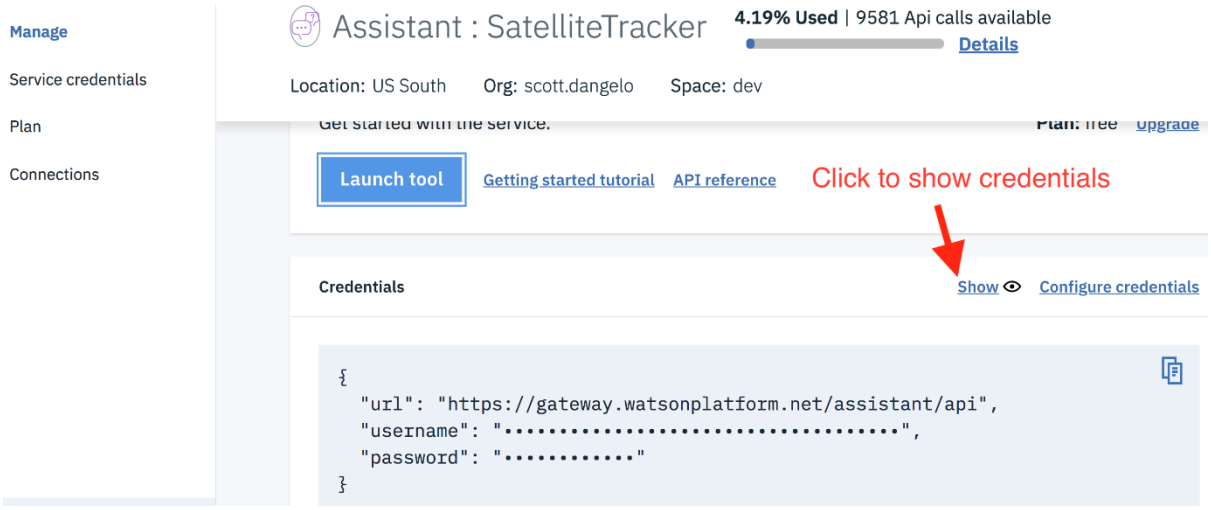
```
 *
 * @param OpenWhisk actions accept a single parameter,
 *        which must be a JSON object.
 *
 * In this case, the params variable will look like:
 *     { "message": "xxxx" }
 *
 * @return which must be a JSON object.
 *         It will be the output of this action.
 *
 */
var request = require('request');

function main(params) {
   var username = params.username
   var password = params.password
   var iamApiKey  = params.iamApiKey
   var workspace_id = params.workspace_id
   var auth = {"user": username,"pass":password}
   var url = "https://gateway.watsonplatform.net/assistant/api/v1/workspaces/" +
workspace_id + "/message?version=2018-07-10"
   if(iamApiKey){
    auth = {"user": "apikey","pass":iamApiKey}
    url  =  "https://gateway-wdc.watsonplatform.net/assistant/api/v1/workspaces/"
+ workspace_id + "/message?version=2018-07-10"
   }
   var input_text = params.data
   var body = {"input": {"text": input_text}}
   return new Promise(function(resolve, reject) {
```

```
        request( {
            url: url,
            method: 'POST',
            auth: auth,
            headers: {
              "content-type": "application/json",
            },
            body: JSON.stringify({"input": {
                "text": input_text
                }})
          },
            function(error, response, body) {
            if (error) {
                reject(error);
            }
            else {
                var output = JSON.parse(body)
                resolve({msg: output});
            }
        });
    }
    );

}
```



Then insert the corresponding credentials when running the commands below.

If the service credentials from IBM Watson Assistant are username/password based as shown in the diagram below, populate the username, password and workspace_id fields and comment out the IAM credentials fields.

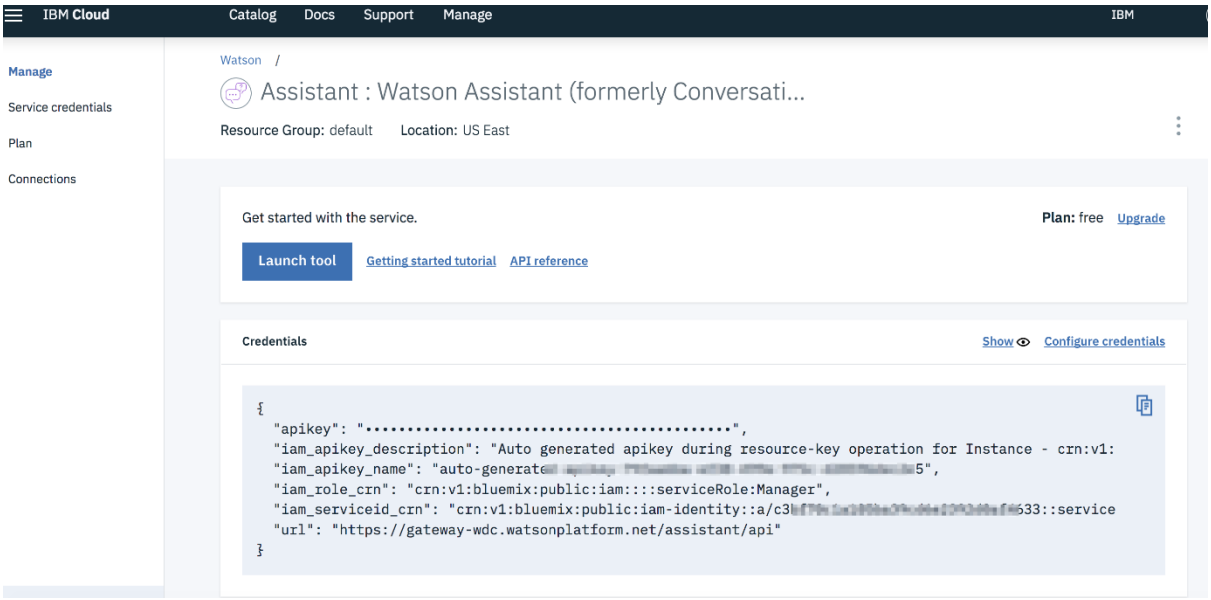


wsk action update conversation -p username ${conversation_username} -p password ${conversation_password} -p workspace_id ${conversation_workspace_id}

wsk action update iot-pub -p iot_org_id ${iot_org_id} -p device_id ${device_id} -p api_token ${api_token} -p device_type ${device_type}

wsk package bind /whisk.system/watson-speechToText myWatsonSpeechToText -p username ${stt_username} -p password ${stt_password}

If the service credentials from IBM Watson Assistant are IAM based as shown below in the diagram, populate the IAM apikey, url, and workspace_id fields and comment out the username/password fields.



wsk action update conversation -p iamApiKey ${apikey} -p workspace_id ${conversation_workspace_id}

wsk action update iot-pub -p iot_org_id ${iot_org_id} -p device_id ${device_id} -p api_token ${api_token} -p device_type ${device_type}

wsk package bind /whisk.system/watson-speechToText myWatsonSpeechToText -p username ${stt_username} -p password ${stt_password}

Next, we can arrange the actions into a sequence

wsk action create homeSequence --sequence myWatsonSpeechToText/speechToText,conversation,iot-pub

For the sequence to be able to return the result to the Raspberry Pi, a MQTT client will need to be listening to the Watson IoT service. If the proper values have been set in the /etc/environment file, you should just have to run the following

commands to create and enable a systemd service, which will automatically start on boot. This will start the node server, which subscribes to the Watson IoT Platform's MQTT broker and listens for intent entity pairs.

sudo cp serverless-home-automation/iot-gateway/node-mqtt.service /etc/systemd/system/

sudo systemctl enable node-mqtt

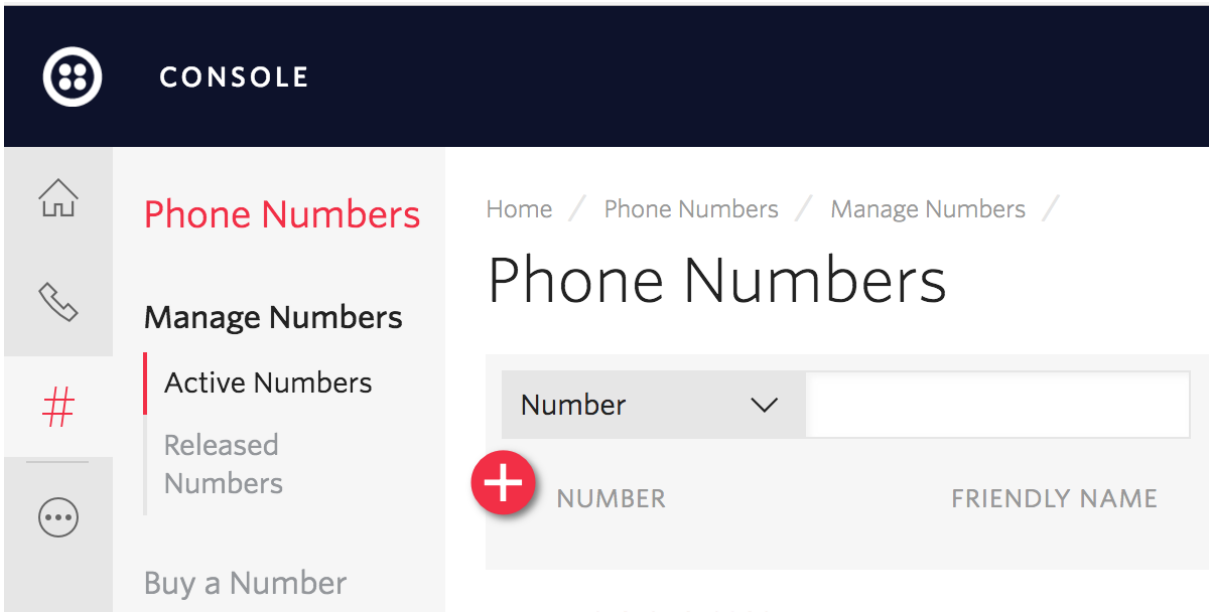
sudo systemctl start node-mqtt

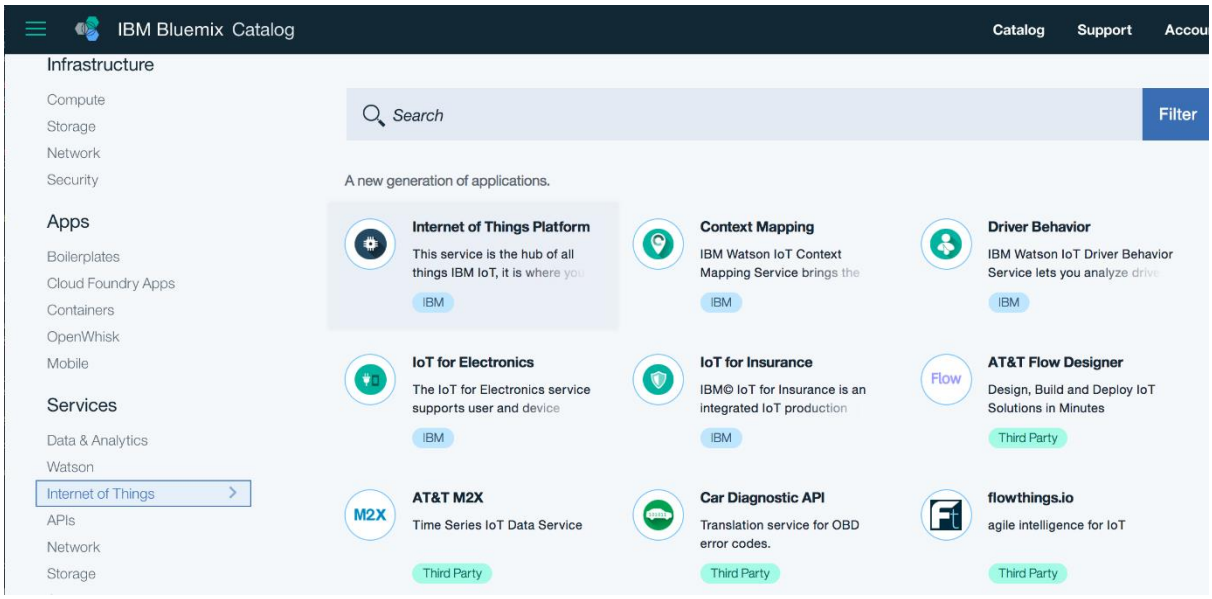
sudo systemctl status node-mqtt

Twilio:

Twilio is a service that enables developers to integrate VoIP and SMS capabilities into their platform. This works by allowing developers to choose a phone number to register. Once registered, Twilio exposes an API endpoint to allow calls and texts to be made programmatically from the number. Also, the number can be configured to respond to incoming calls/texts by either triggering a webhook or following a Twiml document. In this case, we'll configure the Twilio number to respond to incoming texts by triggering a webhook bound to the "homeSequence" IBM Cloud Functions action we created in the previous step. We can find the url to the webhook by navigating to the IBM Cloud Functions console, selecting the homeSequence sequence, and then selecting the View Action Details button. Finally, check the Enable as Web Action button, and copy the generated Web Action URL.

To get started, please visit Twilio's registration page. After signing up, log in and select the # icon in the menu, which will direct the browser to the Phone Numbers configuration. Now, select the circular + button to select and register a number. After registration, click the number to configure it. Scrolling down will reveal a Messaging section. In the form titled A Message Comes in, paste the webhook associated with the "homeSequence" IBM Cloud Functions action, as seen below.

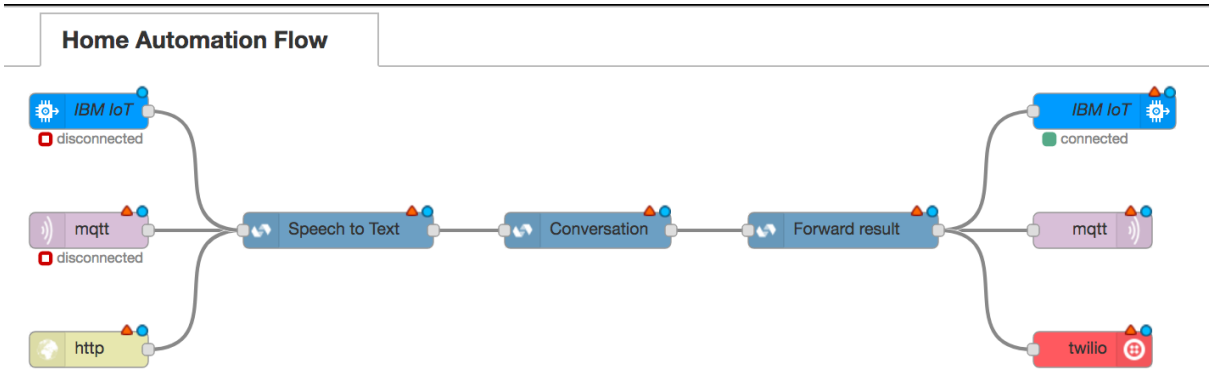




Node Red:

As an alternative to creating sequences in IBM Cloud Functions, the home automation logic can be arranged using Node Red. Node Red is a visual editor capable of assembling "flows", which is done by allowing users to drag, drop and connect "blocks" of code or service calls. It's worth noting that this deployment scheme won't follow a fully serverless model, as it'll be running constantly as a node server. Since the backend logic is all in the IBM Cloud Functions serverless action pool, the devices should be able to be controlled via SMS or voice without having to set up a long running server. However, in use cases where it's preferable to use node red, we can do so by installing the package via npm install node-red,

booting up the editor via node-red, and creating a flow like what we have in the diagram below. After assembling the flow, be sure to populate the authentication credentials and endpoint for each block.



Troubleshooting

RF Circuit: After checking each of the wires to ensure they are lined up correctly, use a multimeter to check each of the connection nodes starting from the power source. For example, to ensure that RF components are being powered properly, touch the negative/grounded end of the multimeter to the grounded power rail, and touch the positive end of the multimeter to the RF components 5V pin.

Audio: Jack Server

jack server is not running or cannot be started

DISPLAY=:0 jack_control start

pulseaudio –start

IBM Cloud Services: Whenever any of the IBM Cloud components (Speech to Text, Assistant, etc) seem to be unresponsive, check the IBM Cloud Status page to see if the service is down or under maintenance. If not, try running a sample request using curl and ensure that a 200 HTTP response is returned. A sample request against the speech-to-text service would look like so.

```
curl -v -u ${username}:${password} https://stream.watsonplatform.net/speech-to-text/api/v1/models
```