

Internet of Things Workshop

Lab 4

Azure Functions

Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Version | Change Reference |
| 9/16/2016 | Steve Busby | 1.0 | Initial draft |
|  |  |  |  |

Table of Contents

[Change Record 2](#_Toc461829240)

[Introduction 4](#_Toc461829241)

[1.) Create an Azure Function App 5](#_Toc461829242)

[2.) Add reference to the Azure IoTHub Service SDK 11](#_Toc461829243)

Introduction

This lab is focused on configuring an Azure Function to respond to the identification of a high temperature alert, and send a command to the alerting device to allow it to respond to the alert (in this case, by lighting an LED). An alert condition, in the context of this lab, is a high temperature violation of a defined temperature threshold. This lab is the fourth in a series that walks through building an end-to-end Internet of Things prototype for doing temperature monitoring.

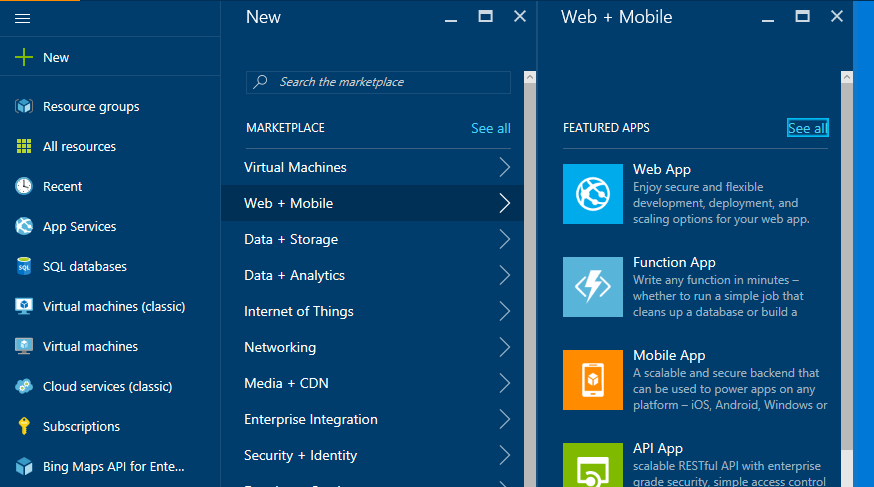
In this series of labs, you will:

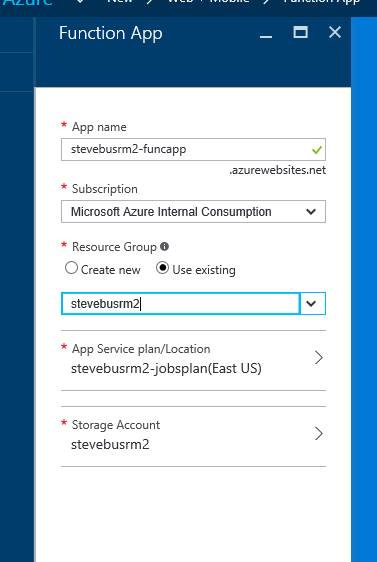
1. Create and navigate the Azure IoT Remote Monitoring Pre-Configured Solution (RM-PCS)
2. Create a device to read a temperature and humidity sensor (and optionally, a light sensor) and send that data to the RM-PCS for display
3. Create a Stream Analytics job that looks for ‘high temperature’ alerts and outputs that alert to a queue for further processing
4. Create an Azure Function that takes that alert, and sends a command to the device to turn on or off an LED depending on the alert condition.

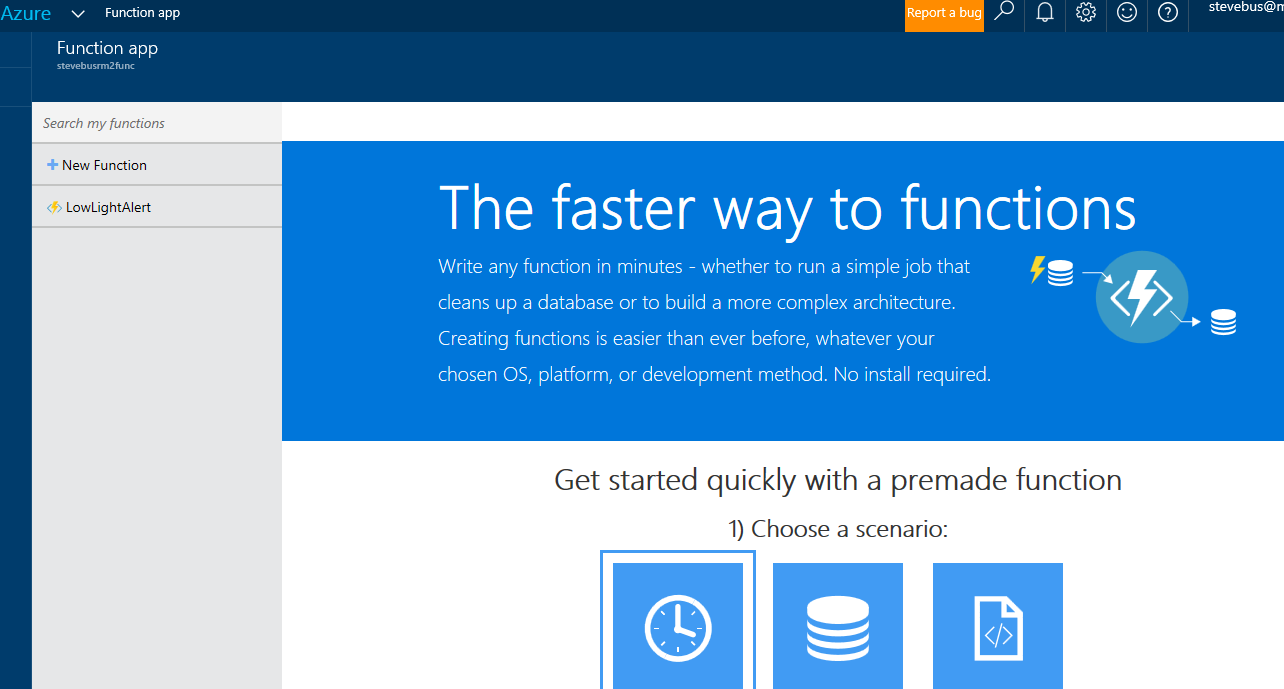
At the end of this lab you will have written the Azure Function to pull the Alert event from the EventHub queue, parse it, and send a command to the device to toggle the “high temp” LED.

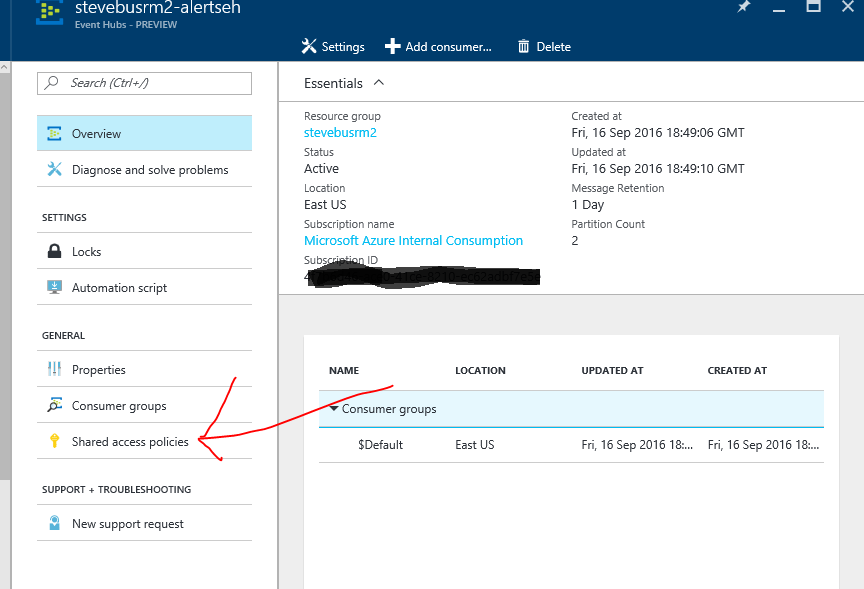
1.) Create an Azure Function App

An Azure Function App is a container for one or more Azure Functions. It sets, among other things, the scalability settings and pricing plans.

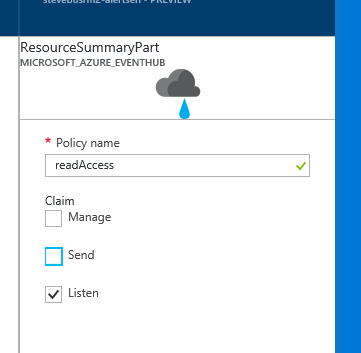
1. Using a web browser navigate to portal.azure.com
2. Create a new Azure Function App from the portal navigation using + New -> Web + Mobile -> Function App.  
   
3. You’ll be presented with the create pane for a Function App. For the App Name field, enter a name for your app (just adding “-FuncApp” to the end of your solution name works well). For “Resource Group”, choose “use existing”. In the drop down list box, you should see a resource group named the same thing as your chosen solution name from lab 1. Choose that. For the App Service Plan, there should be an existing one that has the name of your solution plus “-JobsPlan” on the end. We can use that one for the lab. For storage account, you can use the Azure Storage Account created for the RM-PCS solution. Choose that. Click Create



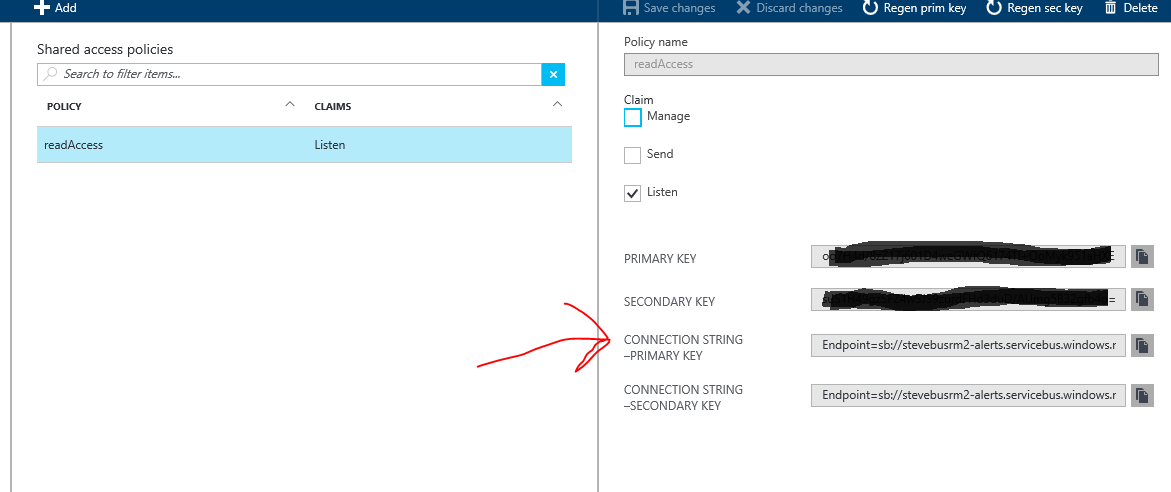
1. Upon completion you’ll be presented the main blade for your Azure Function App. From here we can create a new Azure Function.  
   
2. But before we do, we need to go snag a connection string for our EventHub. In the Azure Portal, on the left nav, click on All Resources, then scroll down and find the EventHub you created as the output in Lab 3. Click on it to open up the blade for it. On that blade, in the “entities” section, expand the “event hubs” section and click on your EventHub you created earlier. On the blade for your eventhub, click on Shared Access Policies



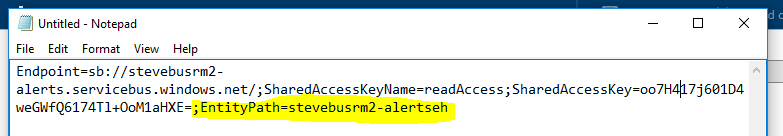
1. Click “Add” to add a shared access policy. Give it a name (any name) and check “Listen”



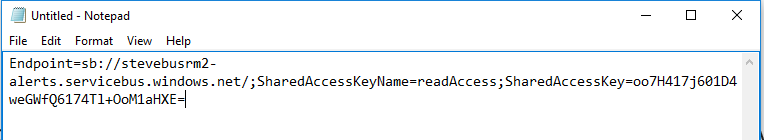
1. Once created, click on the access policy you just created, copy the “Connection String – Primary Key” and paste it into notepad or similar editor.



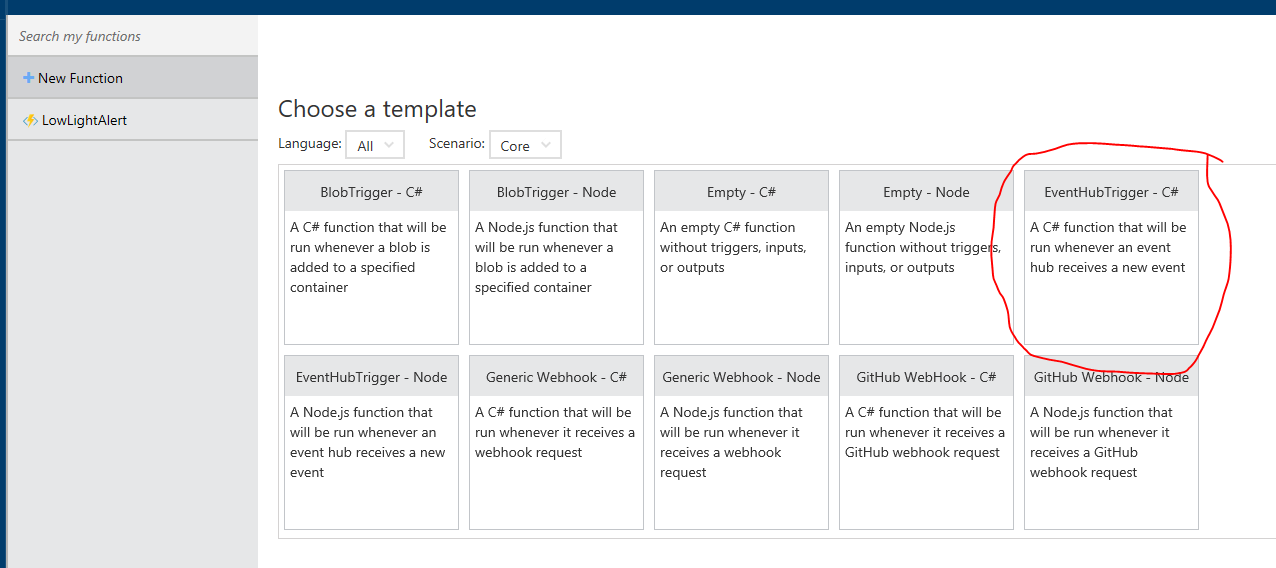
1. In Notepad, delete the section on the end that begins with “EntityPath” (including the ‘;’ in front of it) as shown highlighted below



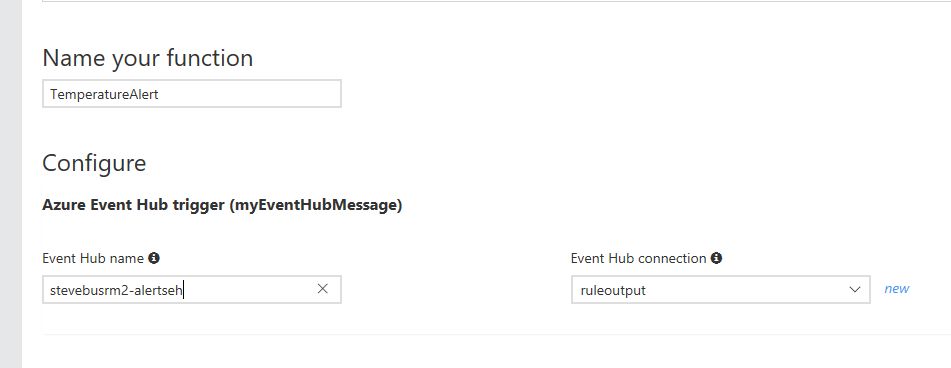
1. After deletion, your connection string should look like the screen shot below (obviously, your name and actual key will be different)



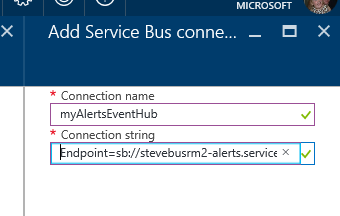
1. Leave notepad open, as we’ll need the connection string in a few steps
2. In the Azure Portal, click on All Resources in the left hand nav, search for your Function App, and re-open it
3. Click “New Function”. We want to ‘trigger’ execution of our function off of messages landing in our EventHub, so choose “EventHubTrigger – C#”



1. At the bottom of the form, give your function a name. Then type the name of your EventHub in the Event Hub name box



1. Under Event Hub connection, click ‘new’, then click “Add a connection string”
2. Give your new connection string a name (any name). In the connection string box, copy and paste the connection string you had previously saved in notepad



1. Back on the main form, choose Create to generate a template for your Azure Function
2. After creation, you will have a template for an azure function that looks like this:

public static void Run(string myEventHubMessage, TraceWriter log)

{ log.Info($"C# Event Hub trigger function processed a message: {myEventHubMessage}");

}

1. This “Run” function will get invoked for each message that is dropped in the EventHub. The actual message in the hub is passed in the “myEventHubMessage” string
2. Beneath the Code box is the “Logs” box. This box shows real-time logging from your Azure Function as it executes (and compiles, etc). the “log.Info” call in our code is an example of how to log debug information to this log. At this point, if our ASA job was running, and the temperature transitioned from low to high (or vice versa),we would see the JSON message produced by our ‘alerts’ ASA job written to the log (feel free to try it). At this point, that’s all the function does. In the next section, we will add functionality to parse this message and send a command to the device that generated the alert
3. Below is an example of what the output looks like, at this point, if you hold your fingers over your DHT22 temperature sensor long enough to let the temperature go over 80, and then release it and let it fall back below



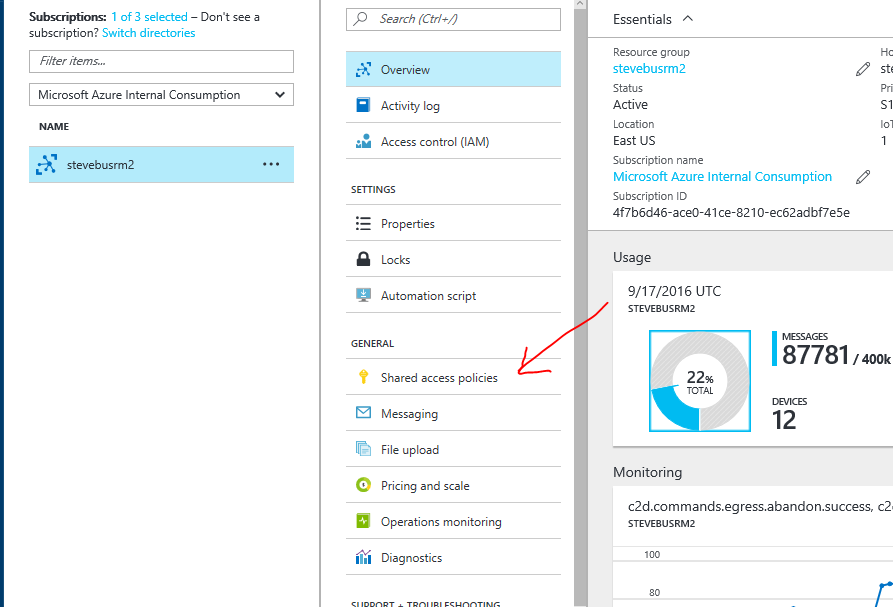
(“rpi-linux” is the Device ID of the raspberry pi used in development of this lab manual)

21.) You now have the basic “wiring” set up to wire our Azure Function to the output of the EventHub queue that contains the temperature alerts. Next we’ll work on developing our code to respond to those alerts

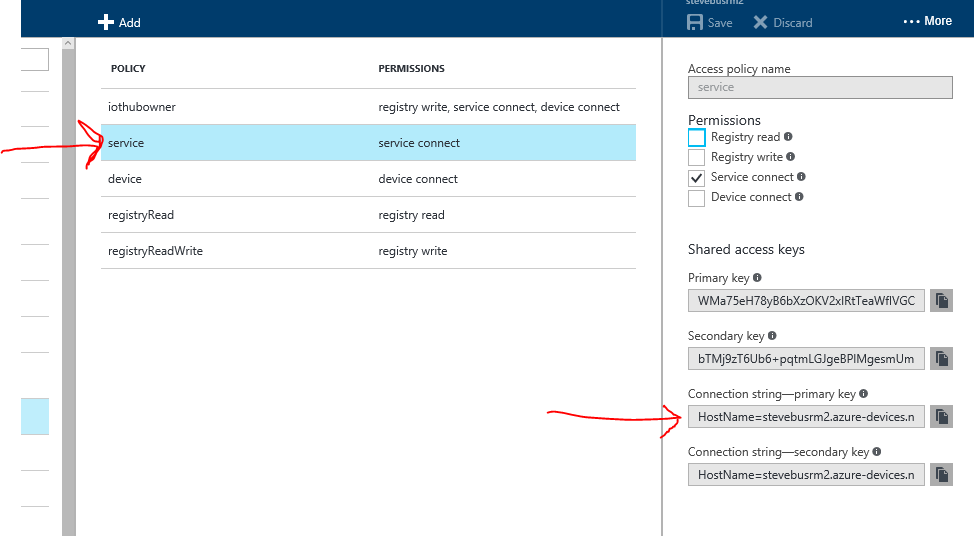
2.) Retrieve IoTHub connection string

To send commands to devices, we need to be able to connect to our IoTHub from our backend function. In order to do so, we need to get a valid connection string to the IoTHub.

1. In the Azure Portal, navigate to the IoTHub associated with your RM-PCS solution. On the main IoTHub blade, click “shared access policies”



1. In the shared access policies screen, select “service”. This is a policy and connection string that only has the permissions to connect to the backend services (i.e if it gets out, no devices can use it to connect). From this screen, Copy the “Connection String – Primary Key” and paste into Notepad for later use



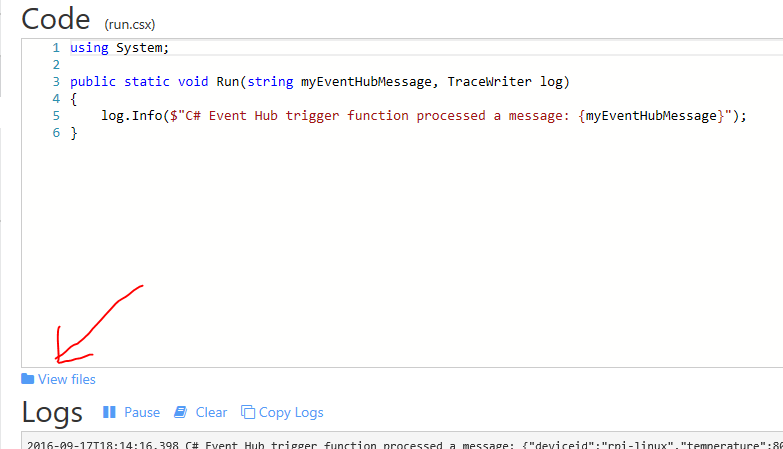
1. Now we are ready to write our Azure Function

3.) Add reference to the Azure IoTHub Service SDK

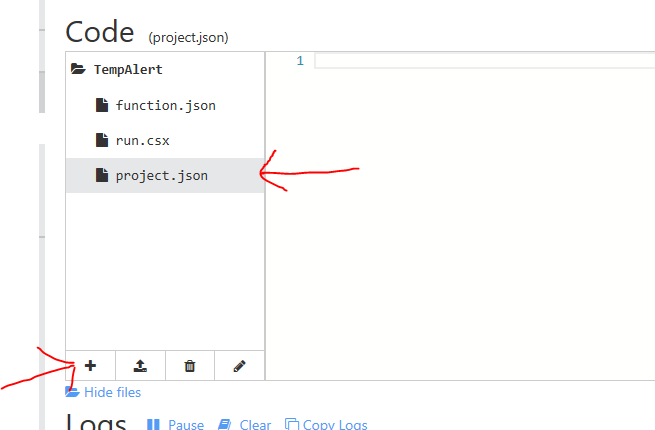
In lab 2, we leveraged the Azure IoTHub \*device\* SDK to allow a device to connection to IoTHub and send and receive messages from it. For a ‘back end’ process like our Azure Function to send a command to a device, it needs to use the \*service\* side SDK for Azure IoT Hub.

To use the SDK, we need to ‘install’ it in our Function App and set a reference to it. We will do so leveraging a Nuget package (the .NET equivalent of apt-get or npm for linux and Node). We will reference the package in a config file, and the Azure Functions framework will automatically download and install it in our Function App.

1. The first step is to create a config file to hold our reference to the Azure IoTHub Service SDK (which is in the Microsoft.Azure.Devices namespace). The name of our file should be project.json and it is located in the folder of our Azure Function
2. In the “develop” tab of your Azure Function, click on “View Files” right below the code editing box



1. This displays the files associated with our function. At this point, we have only two (function.json and run.csx).
2. Click on the “+” to add a file, and name your file ‘project.json’



1. The project.json file will then be open in the editor. Copy and paste the following into the file and hit save

{

"frameworks": {

"net46":{

"dependencies":

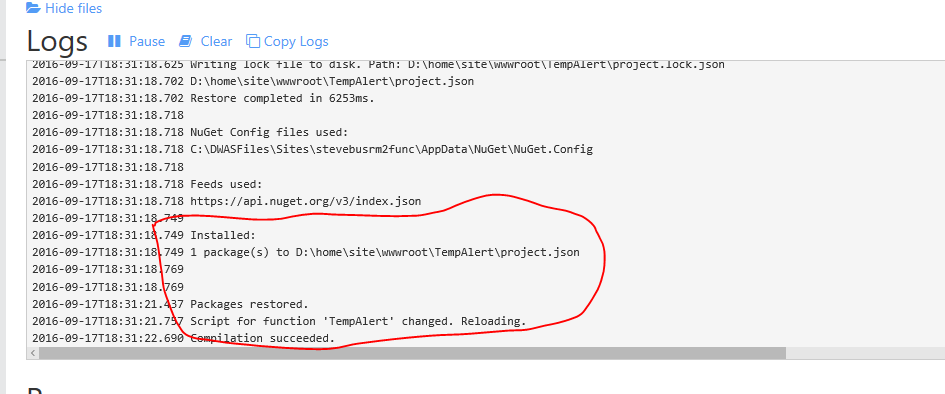
{"Microsoft.Azure.Devices": "1.0.15"}

}

}

}

1. This tells the Azure Functions framework that we want to use version 1.0.15 of the Microsoft.Azure.Devices nuget package (the latest version). The framework will download and install the package. You should see information in the log window similar to the below which indicates that the package was successfully installed.



1. Click back on the run.csx file (which is where our ‘code’ is) to bring back up the code in the editor. (Feel free to click the “hide files” button below the editor at this point to reclaim screen real estate --- the rest of our work will be in run.csx)
2. At the top of the file, right below “using System;”, add these two lines. This tells the framework that we will be using objects out of those two namespaces (for compiler name resolution purposes). Because the vast majority of data from ASA will be either JSON or CSV (and because the Azure Function internals use it anyway ☺), the open source JSON parser from Newtonsoft is already referenced automatically by the framework and we don’t have to explicitly install it like we did with the IoTHub Service SDK. System.Text is included so we can encode our command as an array of bytes

using Microsoft.Azure.Devices;

using Newtonsoft.Json.Linq;

using System.Text;

1. Now for our code. Within the Run function, underneath the first ‘logging’ line, enter the code below

// Service Client is the object used to talk to the IoTHub command & control APIs

ServiceClient serviceClient;

// normally, the connection string would be stored securely in configuration.. but, it's a demo :-)

string connectionString = "HostName=stevebusrm2.azure-devices.net;SharedAccessKeyName=service;SharedAccessKey=WMa75eH78yB6bXzOKV2xIRtTeaWflVGCRr8rUX+nPMk=";

serviceClient = ServiceClient.CreateFromConnectionString(connectionString);

// parse the JSON object from the queue

dynamic jsonData = JObject.Parse(myEventHubMessage);

// pull out the device ID and whether it's a high or low temp state

string deviceID = jsonData["deviceid"].ToString();

string alertTempState = jsonData["tempstate"].ToString();

// if low, turn off, otherwise turn on

string command = "OFF";

if(alertTempState == "OVER")

{

command = "ON";

log.Info("High threshold violated, sending 'ON' command");

}

else

log.Info("Temp now below threshold, sending 'OFF' command");

// create a message object to wrap the command and send as an array of bytes

var commandMessage = new Message(Encoding.ASCII.GetBytes(command));

// send the command to the device

serviceClient.SendAsync(deviceID, commandMessage);

1. The comments should be self explanatory, but essentially we create a connection to IoTHub via the ServiceClient object, we then parse the JSON to determine the target deviceID and what kind of message we need to send (on or off), and then create and send the message
2. On your device, hold your fingers around the DHT22 temperature sensor and drive the temperature over 80. You should see the appropriate debug output in the Azure Function log, you should see the command received in the console for the Raspberry PI, and the LED should come on and stay on. Once done, release your finger and let the temperature drop back below 80 and you should see the opposite occur.
3. Congratulations – you’ve “round tripped” telemetry data from your device, through the RM-PCS (visualized in the portal), wrote a stream analytics job to look for high temperature alerts, and send a command back to the device to respond to that alert.