ConnectTheDots – Quick Start

Intro

The MS Open Tech ConnectTheDots.io project illustrates how to connect sensors and devices to the Microsoft Azure Cloud, and use Microsoft Azure to analyze and visualize the resulting data streams.

In a typical topology, sensors (here several Arduino Uno R3 boards with Arduino Weather Shields) connect to one or more local IoT Gateways (here Raspberry PI devices), which relay the data to Microsoft Azure Event Hubs. Once in the cloud, the data streams are fed into a web dashboard and to near real-time analytics engines (here Microsoft Azure Stream Analytics, in this case to generate averages and alerts across all devices). The real-time data streams, average, and alerts are then visualized in a Microsoft Azure Website, which can be viewed with any HTML5-capable browser. The high level architecture for the ConnectTheDots.IO project is shown in Figure 1.

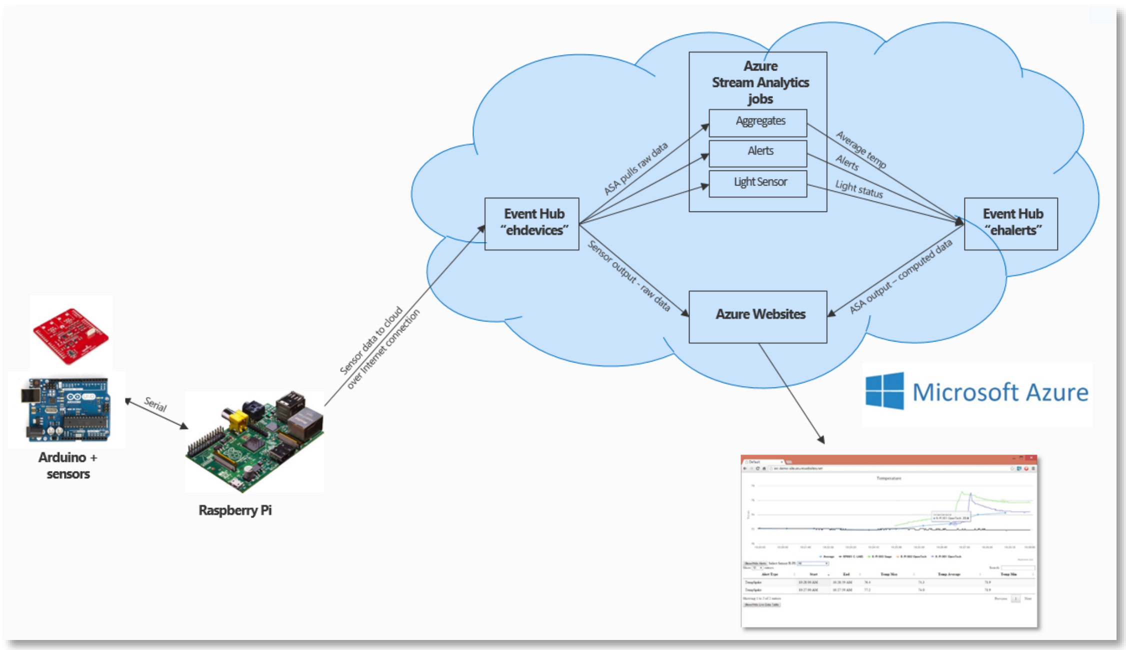


Figure 1: ConnectTheDots.IO architecture

The ConnectTheDots.IO project provides you with all the code and step-by-step instructions to build and deploy such an end-to-end solution.

Prerequisites:

1) Microsoft Azure subscription (free trial subscription is sufficient)
2) Access to the Azure Streaming Analytics Preview
3) Visual Studio 2013 – Community Edition

Hardware:

1) Raspberry PI B/B+ with Internet access
2) Arduino Uno R3

3) SparkFun Weather Shield for Arduino (make sure you also have the required headers etc. as specified on the SparkFun site)

Note: Only the models above have been tested. The Weather Shield sample code from Sparkfun, for example, may need to be modified before it will work reliably on Arduino Due or Arduino Uno R2.

Step-by-step

1) Download and Build "ConnectTheDots":
    a. Download or clone the "ConnectTheDots" solution from https://github.com/MSOpenTech/connectthedots. Save it to a directory near the root of your drive, e.g. c:\software, as the subsequent build might fail if you bury it too far down and the pathnames are too long.
    b. Open ConnectTheDots.sln in Visual Studio and build the solution. This will create three separate sets of programs – one for the Arduino, one for the Raspberry, and one for the Azure website, as well as some additional files to help configure the various components.
    If you get build errors showing that the AMQP.Net Lite NuGet package cannot be found (it has not been published as of this writing), you will need to build it yourself:
        i. Download or clone from http://amqpnetlite.codeplex.com
        ii. In the directory where you saved AMQPLite, open Amqp.Net.nuspec for editing. To avoid errors when you build the ConnectTheDots solution, comment out or remove all <file> elements for platforms for which you don't have tooling, that is, comment or delete all except for "bin\Release\Amqp.Net\*.*" in Amqp.Net.nuspec:

```
<files>
    <file src="bin\Release\Amqp.Net\*.*" target="lib\net" />
    <!--<file src="bin\Release\Amqp.NetCF39\*.*" target="lib\netcf39" />
    <file src="bin\Release\Amqp.NetMF42\*.*" target="lib\netmf42" />
    <file src="bin\Release\Amqp.NetMF42\le\*.p*" target="lib\netmf42\le" />
    <file src="bin\Release\Amqp.NetMF42\be\*.p*" target="lib\netmf42\be" />
    <file src="bin\Release\Amqp.NetMF43\*.*" target="lib\netmf43" />
    <file src="bin\Release\Amqp.NetMF43\le\*.p*" target="lib\netmf43\le" />
    <file src="bin\Release\Amqp.NetMF43\be\*.p*" target="lib\netmf43\be" />
    <file src="bin\Release\Amqp.Store\*.*" target="lib\windows8" />
    <file src="bin\Release\Amqp.WP\*.*" target="lib\windowsphone8" />-->
  </files>
```

Figure 2: Edited Amqp.net.nuspec file

        iii. Open a Developer Command Prompt for VS2013, and run build.cmd from the directory in which you saved AMQPLite. You should now see AMQPNetLite.0.1.0-alpha.nupkg in the build/packages folder.
        iv. Back in Visual Studio, add this folder as a private NuGet location:
            • Open Tools, Options, NuGet Package Manager, Package Sources and add "<youramqpnetlitepath>\Build\Packages" as a new package source.
        v. Build the ConnectTheDots.sln solution again.

2) Create Azure resources for Event Hub:
    a. Download publishsetting file from Azure. This will contain information about your current subscription and be used to configure other components of your solution.
        i. Go to https://manage.windowsazure.com/publishsettings/index?client=powershell) and save to local disk <publishsettingsfile> (contains keys to manage all

resources in your subscriptions, so handle with care). Save this to the same folder as the ConnectTheDotsDeploy.exe file is found (Either Azure\ConnectTheDotsCloudDeploy\bin\Release directory (or \Debug directory, depending upon how you built the solution.)

    ii. **If you have access to multiple subscriptions, make sure the file only contains the subscription that you want to use. Otherwise, edit and remove the other XML elements for the other subscriptions.**

b. Run ConnectTheDotsCloudDeploy from an elevated command prompt ("Run as administrator") , passing a name to be used for all cloud resources, and the publishsetting file

    i. Chose a <name> that has only letters and numbers – no spaces, dashes, underlines, etc. (If the publishsettingsfile filename has spaces in it, you will get an error saying the file cannot be found. Surround just the publishsettingsfile with quotation marks and re-run ConnectTheDotsCloudDeploy.exe.)

```
cd ConnectTheDotsCloudDeploy\bin\debug\
ConnectTheDotsCloudDeploy.exe -n <name> -ps <publishsettingsfile>
```

Figure 3: ConnectTheDotsCloudDeploy.exe command line parameters

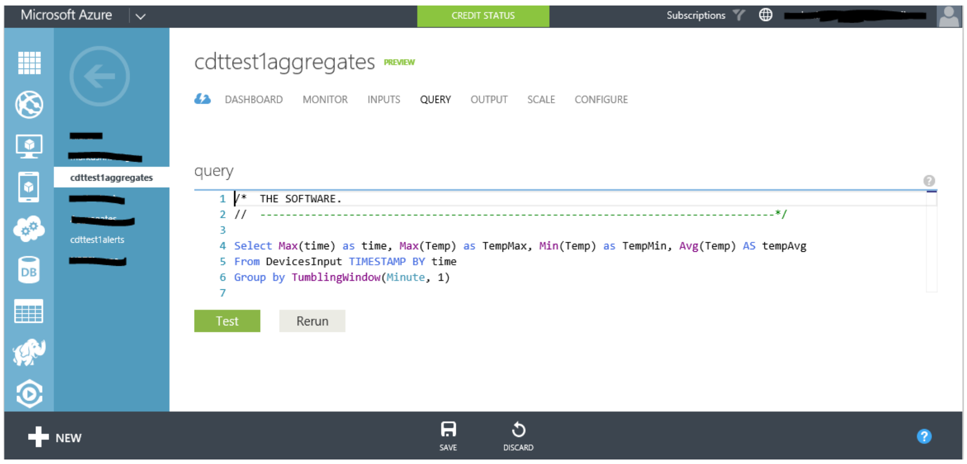
c. Note the device connection strings displayed by the tool, highlighted below, as you will need them to provision the devices later. You might copy and paste into Notepad for easy retrieval.

```
C:\MyProjectLocation\connectthedots\Azure\ConnectTheDotsCloudDeploy\bin\Debug>
ConnectTheDotsCloudDeploy.exe -n ctdtest1 -ps
C:\MyTempFolder\MyAzureSubscription.publishsettings
Creating Service Bus namespace ctdtest1-ns in location Central US
Namespace cdttest1-ns in state Activating. Waiting...
Namespace cdttest1-ns in state Activating. Waiting...
Namespace cdttest1-ns in state Activating. Waiting...
Creating Event Hub EHDevices
Creating Consumer Group WebSite on Event Hub EHDevices
Creating Consumer Group WebSiteLocal on Event Hub EHDevices
Creating Event Hub EHAlerts
Creating Consumer Group WebSite on Event Hub EHAlerts
Creating Consumer Group WebSiteLocal on Event Hub EHAlerts
Creating Storage Account cdttest1storage in location Central US

Service Bus management connection string (i.e. for use in Service Bus Explorer):
Endpoint=sb://ctdtest1-
ns.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAcc
essKey=zzzzzzz

Device AMQP address strings (for Raspberry PI/devices):
amqps://D1:xxxxxxxx@yyyyyyyy.servicebus.windows.net
amqps://D2:xxxxxxxx@yyyyyyyy.servicebus.windows.net
amqps://D3:xxxxxxxx@yyyyyyyy.servicebus.windows.net
amqps://D4:xxxxxxxx@yyyyyyyy.servicebus.windows.net

Web.Config saved to
C:\MyProjectLocation\connectthedots\Azure\ConnectTheDotsWebSite\web.config
```

Figure 4: ConnectTheDotsCloudDeploy.exe command line output

As you can see in Figure 4, the ConnectTheDotsCloudDeploy command created two Event Hubs, EHDevices and EHAlerts, also shown in Figure 1 ConnectTheDots.IO architecture, earlier. It also created four endpoints for AMQP connections from up to four Raspberry Pi devices.

3) Create three Azure Stream Analytics (ASA) jobs
   a. Make sure you have access to the ASA preview> If you don't, sign up at https://account.windowsazure.com/PreviewFeatures
   b. Create the first job
      i. Open the Azure Management Portal, and create a new job "**Aggregates**":
         - "+" in lower left corner -> Data Services -> Stream Analytics -> Quick Create -> Job name "Aggregates".
      ii. Create an input
         - Select the Inputs tab in the Aggregates job.
            a. Inputs tab -> Add an Input -> Data Stream, Event Hub
         - Input Alias: "DevicesInput"
         - Subscription: "Use Event Hub from Current Subscription"
         - Choose the namespace <name>-ns, where <name> is the name you created when running the ConnectTheDotsDeploy in step 2.b above.
         - Event Hub "ehdevices"
         - Policy Name: "StreamingAnalytics"
         - Serialization: JSON, UTF8
      iii. Create a query
         - Select the Query tab in the Aggregates job
         - Copy/paste contents "Aggregates.sql" found in the \Azure\StreamingAnalyticsQueries folder in Windows Explorer
         - Save



```
/*  THE SOFTWARE.
//  ---------------------------------------------------------------------------------*/

Select Max(time) as time, Max(Temp) as TempMax, Min(Temp) as TempMin, Avg(Temp) AS tempAvg
From DevicesInput TIMESTAMP BY time
Group by TumblingWindow(Minute, 1)
```

Figure 5: Creating the Azure Stream Analytics query

      iv. Create an output
         - Select the Output tab in the Aggregates job
            a. Output tab -> Add an Output, Event Hub,
         - Choose the namespace <name>-ns,
         - Event Hub "ehalerts"
         - Policy name "StreamingAnalytics"
         - Serialization "JSON", UTF8
      v. Start the Job
         - Dashboard, Start

   c. Create a second job “**Alerts**”: as above, but use “alert.sql” contents for the query (in step iii/2.)
   d. Create a third job “**LightSensor**”: as above, but use “lightsensor.sql” contents for the query (in step iii/2.)

4) Publish the Azure Website
   a. In VS: Right-click on Azure\ConnectTheDotsWebSite, Publish.
   b. Select Azure Web Sites, create new one.
      i. Site name: <pick something unique>
      ii. Region: <pick same region as you used for Stream Analytics>
      iii. Database server: no database
      iv. Password: <leave suggested password>
   c. Publish [You might need to install WebDeploy extension if you are having an error stating that the Web deployment task failed. You can find WebDeploy here).
   d. Enable WebSockets for the new Azure Web site
      i. Browse to https://manage.windowsazure.com and select your Azure Web Site.
      ii. Click on the Configure tag. Then set WebSockets to On
   e. Open the site in a browser to verify it has deployed correctly.
      i. At the bottom of the page you should see “Connected.”. If you see “ERROR undefined” you likely didn’t enable WebSockets for the Azure Web Site (step d above).

5) Provision the Raspberry PI:
   a. Connect the Raspberry Pi to a power supply, keyboard, mouse, monitor, and Ethernet cable (or Wi-Fi dongle) with and Internet connection.
   b. Get a Raspbian NOOBS SD Card or download a NOOBS image as per the instructions on http://www.raspberrypi.org/downloads/
   c. Boot the NOOBS SD Card and choose Raspbian (see http://www.raspberrypi.org/help/noobs-setup/ for more information).
   d. Connect to the Raspberry PI from your laptop, either via a USB-Serial adapter or via the network via SSH (enable once as per these instructions while booting via a monitor on HDMI and a USB keyboard). To connect using SSH:
      i. For Windows, download PuTTY and PSCP from here.
      ii. Connect to the Pi using the IP address of the Pi.
   e. Once you have connected to the Pi, install on it the Mono runtime and root certs required for a secure SSL connection to Azure:
      i. Run the following from a shell (i.e. via SSH).
      Note: Especially steps 1 and 2 can take a long time to download/un-compress.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install mono-complete
mozroots --import --ask-remove
```

Figure 6: Installing mono on the Raspberry Pi

   f. Create a directory /home/pi/ RaspberryPiGateway. [NOTE: directory names on the Raspberry are CASE SENSITIVE, so raspberrypigateway is not the same as RaspberryPiGateway]:

```
mkdir /home/pi/ RaspberryPiGateway
```
Figure 7: Creating directory on Raspberry Pi

g. Using Visual Studio on your PC, update Devices\RaspberryPiGateway\scripts\autorun.sh with any one of the four amqp address strings returned by ConnectTheDotsCloudDeploy.exe, i.e. amqps://D1:xxxxxxxx@yyyyyyyy.servicebus.windows.net:

```
…
#!/bin/bash
echo "Running CloudDemo..."
#sudo -u pi /usr/bin/mono /home/pi/RaspberryPiGateway/RaspberryPiGateway.exe -
forever -deviceid C8CA5B13-A550-4FF0-B823-46D8A2640880 -devicename "R-PI 001" -
address amqps://D1:xxxxxxxx@yyyyyyyy.servicebus.windows.net -target ehdevices -
tracelevel information >> /home/pi/RaspberryPiGateway/RaspberryPiGateway.log &
…
```
Figure 8: Updating the autorun.sh file on your PC before copying to the Raspberry Pi

You can also change the –devicename from “R-PI 001” if you like. This is the device name that will show up in the website.

Note: Make sure you use an editor that preserves UNIX line end (Visual Studio does by default, Notepad does NOT). Otherwise the autorun.sh file will not execute on the Raspberry PI.

h. Copy the required files from Devices\RaspberryPiGateway\bin\Debug to Raspberry’s /usr/pi/RaspberryPiGateway/:
   - RaspberryPiGateway.exe
   - Amqp.Net.dll
   - Newtonsoft.Json.dll
   - scripts/autorun.sh

   You can copy directly to the SD card, or use the scprip.cmd file found in \devices\RaspberryPiGateway\scripts\scprpi.cmd to copy via SSH. To use the .CMD file, you will need to
   i. Remove the "REM" and update the IP address
   ii. Change the SCPR download location in the .CMD file as necessary
   iii. Change bin\Debug or bin\Release to reflect whether you built the solution to Debug or Release.

i. Log in to the Raspberry Pi via PuTTY, and make /home/pi/RaspberryPiGateway/autorun.sh executable:

```
sudo chmod 755 /home/pi/RaspberryPiGateway/autorun.sh
```
Figure 9: making the autorun.sh file bootable on the Raspberry Pi

j. On the Raspberry PI, modify /etc/**rc.local** by adding one line to start the gateway program on every boot (OK to skip this if you prefer to run manually after every reboot via /autorun.sh from a shell/SSH session):

```
Sudo nano /etc/rc.local
```
Figure 10: editing the rc.local file on the Raspberry Pi

When you are in the nano editor, insert or change the reference to autorun.sh to be the following:

```
/home/pi/RaspberryPiGateway/autorun.sh &
```

Figure 11: making the Raspberry run the autorun.sh file upon boot

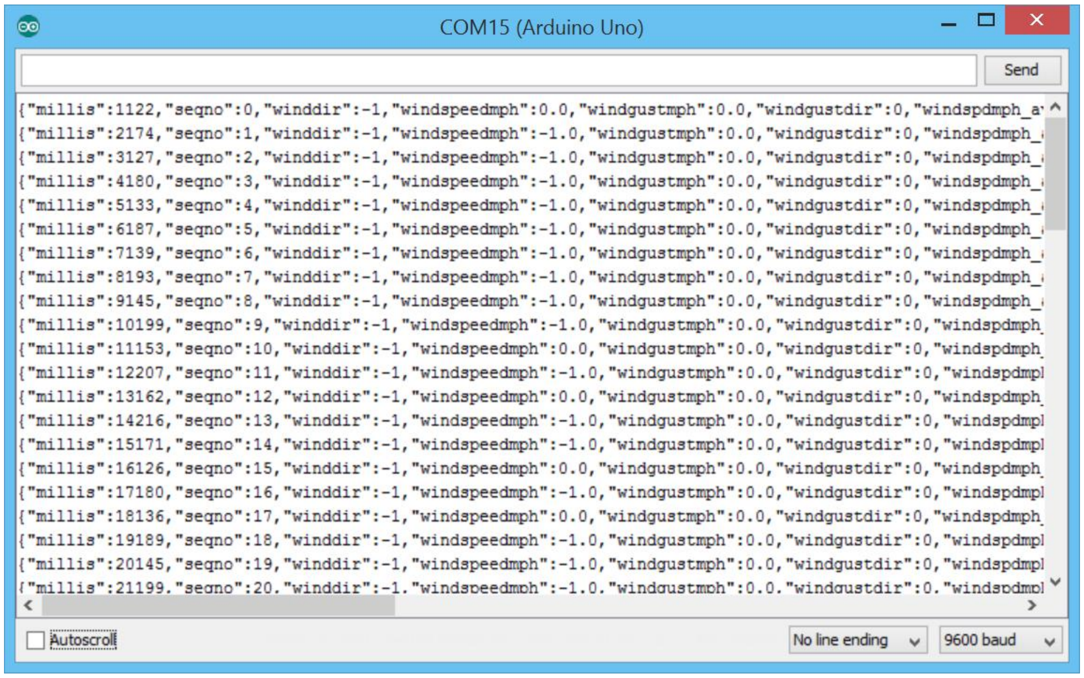
To exit the nano editor use Ctrl-x. To have the new settings take effect, reboot the Raspberry Pi by cycling the power or by issuing the command

```
Sudo reboot
```

Figure 12: rebooting the Raspberry Pi

6) Prepare the Arduino Uno R3:
   a. Connect the Arduino Uno R3 directly to your computer with the USB cable
   b. Install and run the Arduino IDE (we recommend the 1.5.8 version, with the Windows Installer).
   c. If necessary, install the Windows device drivers for the Arduino on your computer, following the instructions here.
   d. Download the Weather Shield libraries from here (as per the instruction in the Weather Shield Hookup guide).
      i. Extract all files from the ZIP file to a temporary location, then import the two folders (HTU21D_Humidity and MPL3115A2_Pressure) in the Arduino IDE by clicking Sketch, Import Library, Add Library and selecting the folder (once for each folder).
   e. In the Arduino IDE open Devices\Arduino\WeathershieldJson.ino (it is modified from the original Spark fun sample to send data in JSON format)
   f. Compile and upload the Weather Shield sketch to the Arduino: File/Upload or Crtl-U or press the Right Arrow.

g. Open the serial monitor (shift-ctrl-m). You should now see temperature and other data on the serial monitor:



```
{"millis":1122,"seqno":0,"winddir":-1,"windspeedmph":0.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_a
{"millis":2174,"seqno":1,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":3127,"seqno":2,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":4180,"seqno":3,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":5133,"seqno":4,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":6187,"seqno":5,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":7139,"seqno":6,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":8193,"seqno":7,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":9145,"seqno":8,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph_
{"millis":10199,"seqno":9,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmph
{"millis":11153,"seqno":10,"winddir":-1,"windspeedmph":0.0,"windgustmph":0.0,"windgustdir":0,"windspdmph
{"millis":12207,"seqno":11,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":13162,"seqno":12,"winddir":-1,"windspeedmph":0.0,"windgustmph":0.0,"windgustdir":0,"windspdmph
{"millis":14216,"seqno":13,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":15171,"seqno":14,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":16126,"seqno":15,"winddir":-1,"windspeedmph":0.0,"windgustmph":0.0,"windgustdir":0,"windspdmph
{"millis":17180,"seqno":16,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":18136,"seqno":17,"winddir":-1,"windspeedmph":0.0,"windgustmph":0.0,"windgustdir":0,"windspdmph
{"millis":19189,"seqno":18,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":20145,"seqno":19,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
{"millis":21199,"seqno":20,"winddir":-1,"windspeedmph":-1.0,"windgustmph":0.0,"windgustdir":0,"windspdmp
```

Figure 13: Viewing the weather shield output in the Serial Monitor

h. Disconnect the Arduino from your computer

7) Run Raspberry PI + Arduino

   a. Plug Arduino's USB cable into one of the Raspberry PI USB ports:

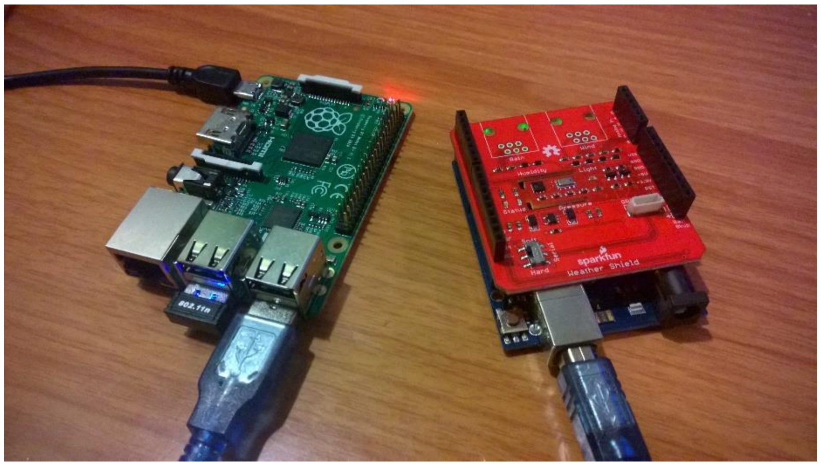
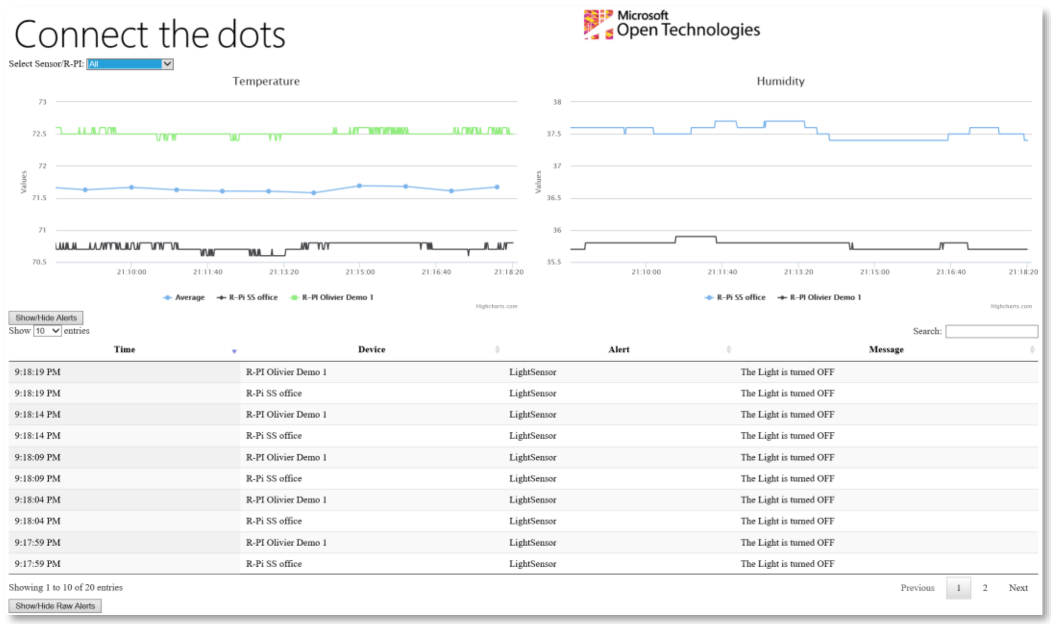


Figure 14: Raspberry Pi and Arduino Uno with weather shield

Note/Tip: The R-PI in the picture uses a WIFI dongle instead of Ethernet. Look [here]() for tips to configure your SSID/keys.

b. (Re-)start Raspberry PI

You should now see average temperature measurements showing up in the web browser every minute.
If you select “All”, you should see raw readings from the device coming in every second.
If the temperature exceeds 75 degrees (F), you should see an alert showing in the alerts table, once every 20 seconds while the temperature on any of the devices exceeds 75 degrees (F).
If you cover the shield, you will see an alert telling you the light is turned off.



Time	Device	Alert	Message
9:18:19 PM	R-PI Olivier Demo 1	LightSensor	The Light is turned OFF
9:18:19 PM	R-Pi SS office	LightSensor	The Light is turned OFF
9:18:14 PM	R-PI Olivier Demo 1	LightSensor	The Light is turned OFF
9:18:14 PM	R-Pi SS office	LightSensor	The Light is turned OFF
9:18:09 PM	R-PI Olivier Demo 1	LightSensor	The Light is turned OFF
9:18:09 PM	R-Pi SS office	LightSensor	The Light is turned OFF
9:18:04 PM	R-PI Olivier Demo 1	LightSensor	The Light is turned OFF
9:18:04 PM	R-Pi SS office	LightSensor	The Light is turned OFF
9:17:59 PM	R-PI Olivier Demo 1	LightSensor	The Light is turned OFF
9:17:59 PM	R-Pi SS office	LightSensor	The Light is turned OFF

Showing 1 to 10 of 20 entries

Figure 15: The Arduino sensor data sent by the Raspberry Pi to Azure as seen from the website

Additional things to try:

To add **more devices**, you can modify the parameters to RaspberryPiGateway.exe in autorun.sh before copying the file to additional Raspberry PI: generate a new GUID as the device id, chose a new device display name and chose one of the other AMQP connection strings (so can turn off rogue devices in the cloud later). [Currently only the device display name really needs to be different between devices.].

Multi-environment deployment (i.e. test, demo, production): Use ASP.Net Web.config transforms to keep separate settings for each of your deployments. You can associate a transform with each publisher profile, and the deployment tool will generate a transform file if you specify the -transform command line parameter, and use the profile name as the -n parameter, i.e.

```
cd ConnectTheDotsCloudDeploy\bin\debug\
ConnectTheDotsCloudDeploy.exe -n <name> -ps <publishsettingsfile> -transform
```

Figure 16: Multi-environment deployment

Running without hardware: The RaspberryPiGateway is a C# console application that can also run on Windows. Specify a COM port on the command line (i.e. -serial COM10) and you can receive serial data from an Arduino Uno.

If you do not have an Arduino or Weather Shield, you can compile the RaspberryPiGateway to generate random temperature data by uncommenting the following line in Devices\RaspberryPiGateway\Program.cs:

```
//#define SIMULATEDATA
```

Figure 16: Using simulated data

Change the alert threshold: The threshold of 75 is in the Azure Streaming Analytics query (Alerts.sql).

Code walkthrough

Arduino: Temperature Sensor

The Arduino code (Devices\Arduino\WeathershieldJson.ino) is essentially SparkFun's sample sketch, which reads the sensors every second and writes the data to the serial port. The primary modification is to send the data in JSON format instead of the original CSV-like format. In addition, a sequence number and the current time in "millis" are sent, as well as the secondary temperature reading from the humidity sensor:

```
 MYSERIAL.print("{");
 MYSERIAL.print("\"millis\":");
 MYSERIAL.print(millis());
 MYSERIAL.print(",\"seqno\":");
 MYSERIAL.print(sequenceNumber++);
 MYSERIAL.print(",\"winddir\":");
 MYSERIAL.print(winddir);
…
 MYSERIAL.print(light_lvl, 2);
 MYSERIAL.println("}");
```

This makes parsing of the data on the Raspberry PI Gateway more generic, but fundamentally an unmodified sample could have been used at the cost of adding CSV parsing to the Gateway.

Raspberry Pi: Gateway

The Gateway reads the sensor data from a serial port and sends it to Azure Event Hub using the AMQP protocol. The Gateway could have used HTTP as well, but AMQP is more efficient for higher data volumes, and also allows for commands to be sent back from the Cloud to the device over the same connection (the sample does not implement a backchannel, however).

The Gateway code (Devices\RaspberryPiGateway\Program.cs) is a console application and is written in C#. It uses the Mono runtime on the Raspberry PI, but can also be run on Windows to ease development/debugging. Command line arguments allow configuration of

- Serial port names (-serial),
- Azure Event Hub server names and credentials (-address and –target)
- Device friendly name and device id (-devicename, -deviceid)
- Send frequency (-frequency).

The Gateway reads data from the serial port on a background thread, and parses it into a C# dictionary (error handling and loop removed for brevity), which it places into the global variable (lastDataSample):

```
serialPort = new SerialPort (serialPortName, 9600);
serialPort.DtrEnable = true;
serialPort.Open ();
…
var valuesJson = serialPort.ReadLine ();
var valueDict = JsonConvert.DeserializeObject<Dictionary<string, object>>
(valuesJson);
if (valueDict != null)
{
    Interlocked.Exchange(ref lastDataSample, valueDict);
}
```

The main thread retrieves the sample data from the global variable. It converts it into an AMQP message, and adds additional information like

- A message type (called "Subject" to match the name of the corresponding AMQP header), which identifies the message as a "weather" record and can be used by consumers to differentiate it from other sensor types that may be added in the future,
- a UTC date time stamp (the Arduino doesn't have a clock) and
- Device ID and Device Name (so that the Arduino code can remain the same on all sensors),
- A partition key that is used by Azure Event Hub to ensure that all messages from a gateway for into the same Event Hub partition (preserves order, and can allow for more processing by backend systems that need to aggregate data from specific devices).

```
var sample = Interlocked.Exchange(ref lastDataSample, null);
if (sample == null) return;

Message message = new Message();
message.Properties = new Properties()
{
    Subject = "wthr",              // Message type: Weather
    CreationTime = DateTime.UtcNow, // Time of data sampling
};

message.MessageAnnotations = new MessageAnnotations();
// Event Hub partition key: device id - ensures that all messages from this device go
to the same partition and thus preserve order/co-location at processing time
message.MessageAnnotations[new Symbol("x-opt-partition-key")] = deviceId;

message.ApplicationProperties = new ApplicationProperties();
message.ApplicationProperties["time"] = message.Properties.CreationTime;
message.ApplicationProperties["from"] = deviceId; // Originating device
message.ApplicationProperties["dspl"] = deviceDisplayName;      // Display name for
originating device

if (sample != null && sample.Count > 0)
{
    var outDictionary = new Dictionary<string, object>(sample);
    outDictionary["Subject"] = message.Properties.Subject; // Message Type
    outDictionary["time"] = message.Properties.CreationTime;
    outDictionary["from"] = deviceId; // Originating device
    outDictionary["dspl"] = deviceDisplayName;      // Display name for originating
device

    message.Properties.ContentType = "text/json";
    message.Body = new Data() { Binary =
Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(outDictionary)) };
}
else
{
    message.Properties.Subject = "wthrerr";
}

sender.Send(message, SendOutcome, null); // Send to the cloud asynchronously
```

The AMQP message duplicates key information into the message's ApplicationProperties. This allows for native routing and filtering in AMQP messaging systems like Azure Event Hub (not actually done in this sample). The main message is placed as JSON into the message body (which is opaque to AMQP messaging systems). [Note: an alternate approach that is implemented under #ifdef is to place all properties as ApplicationProperties and not use a message body at all. However, many consumers – like Azure Streaming Analytics) – only process the message body.]

Azure Event Hubs

There is no code required here (other than for creation of the Event Hubs, see deployment tool section below). The sample uses two Event Hubs:

- DeviceData: for the sensor data sent from the Gateways
- Alerts: for alert and aggregate data generated by the Streaming Analytics queries.

While it is possible to use a single Event Hub for both data, using two Event Hubs is simpler and more efficient: it avoids the need for filtering in the queries, and also allows for applications to only consume the reduced throughput alert/aggregate event streams rather than the high-volume raw data stream.

Access to the Event Hubs is secured using SAS (Shared Access Signature) Keys. Given the small number of devices targeted in the sample, each device gets its own SAS key (called D1, … D4 respectively). (Note: Event Hubs only support a limited number of SAS keys. For larger number of devices, SAS Tokens should to be used).
Each consumer of Event Hub data also gets their own SAS key (called WebSite and StramingAnalytics).

The device keys only have Send permission, the consumer key for the WebSite only has Listen permission (Streaming Analytics in the preview requires management permission).

Each Event Hub is created with 8 partitions (the minimum) and devices are implicitly assigned to a partition via the device id (as indicated by the Gateway via the `x-opt-partition-key` annotation, see code sample above).

Azure Streaming Analytics Queries

Azure Streaming Analytics can natively consume data from Azure Event Hubs. Al that is needed it to configure the inputs and output (as described above).

Two very simple queries (Azure\StreamingAnalyticsQueries) are used in this sample: One to generate aggregates across all connected devices (min, max, average temperature) and one to generate alerts if the temperature for any one of the devices exceeds a critical threshold (75 degrees F in this scenario). Most notable is the use of

- the `TIMESTAMP BY time` expression, which indicates which of the fields in the JSON message carries time information and
- the `TumblingWindow(Minute, 1)` group by statement, which aggregates the incoming data into consecutive 1 minute time windows.

Refer to Stream Analytics Query Language Reference for more information on.

Web Site

The Web Site consumes both device data and aggregated data from the Event Hubs. It caches all the data in memory for at least 10 minutes (which is possible given the small number of supported devices), to ensure that browser clients can efficiently reload the data they are displaying.

The Web Site (Azure\ConnectTheDotsWebSite) uses the Event Hub Processor Host (see here for an overview) to consume data from all Event Hub partitions. It uses the Checkpoint feature (PartitionContext.CheckPointAsync), to ensure it can re-read at least 10 minutes of data in the event that it restarts or crashes. It does so by only check pointing data that is at least 10 minutes old.

```
…
if (eventForNextCheckpoint != null
    && eventForNextCheckpoint.EnqueuedTimeUtc + bufferTimeInterval < now
    && lastCheckPoint + maxCheckpointFrequency < now)
{
    await context.CheckpointAsync(eventForNextCheckpoint);
    …
}
```

[Note that in addition the code checkpoints no more frequently than once every 5 seconds per partition, because checkpoints are relatively costly.]

The Event Hub Processor Host creates a `WebSocketEventProcessor` instance for each of the 8 partitions.

Each `WebSocketEventProcessor` instance maintains its own cache of device messages (each message is a Dictionary of name/value pairs):

```
public List<IDictionary<string, object>> bufferedMessages …
```

Note that the `ProcessEventsAsync` method gets invoked sequentially for each partition, so there is no risk of write contention on this list as data is received. However, access to the cache is protected via a lock as browser client threads will read from the cache when they first connect, i.e.

```
lock (bufferedMessages)
{
    bufferedMessages.Add(messagePayload);
}
```

And on the client thread side:

```
public static IEnumerable<IDictionary<string, object>> GetAllBufferedMessages()
{
    var allMessages = new List<IDictionary<string, object>>();
    lock (g_processors)
    {
        foreach (var processor in g_processors)
        {
            lock (processor.bufferedMessages)
            {
                allMessages.AddRange(processor.bufferedMessages);
            }
        }
    }
    return allMessages;
}
```

The other part of the Web Site handles browser clients, primarily in the `MyWebSocketHandler` class in Azure\ConnectTheDotsWebSite\WebSocketHandler.cs. As clients connect, a `MyWebSocketHandler` instance is created:

```
public class WebSocketConnectController : ApiController
{
    // GET api/<controller>
    public HttpResponseMessage Get(string clientId)
    {
        HttpContext.Current.AcceptWebSocketRequest(new MyWebSocketHandler());
        return Request.CreateResponse(HttpStatusCode.SwitchingProtocols);
    }
}
```

On first connect, the `MyWebSocketHandler` then returns any data from the device data cache:

```
public override void OnOpen()
{
   _clients.Add(this);
    ResendDataToClient();
}
public void ResendDataToClient()
{
    var bufferedMessages = WebSocketEventProcessor.GetAllBufferedMessages();
    this.Send(JsonConvert.SerializeObject(new Dictionary<string, object>
        {
            { "bulkData", true }
        }
    ));
    foreach (var message in bufferedMessages)
    {
        this.SendFiltered(message);
    }
    this.Send(JsonConvert.SerializeObject(new Dictionary<string, object>
        {
            { "bulkData", false}
        }
    ));
}
```

Note that it first indicates to the client that a bulk update is starting, so that clients can avoid refreshing the UI during initialization.

Each client can indicate for which device it wants to receive live data. This selection is kept in `MyWebSocketHandler.DeviceFilter`:

```
public override void OnMessage(string message)
{
    try
    {
        var messageDictionary = (IDictionary<string, object>)
            JsonConvert.DeserializeObject(message, typeof(IDictionary<string,
object>));

        if (messageDictionary.ContainsKey("MessageType"))
        {
            switch (messageDictionary["MessageType"] as string)
            {
                case "LiveDataSelection":
                    DeviceFilter = messageDictionary["DeviceName"] as string;
                    break;
                default:
                    Trace.TraceError("Client message with unknown message type: 
{0} - {1}", messageDictionary["MessageType"], message);
                    break;
            }
        }
        else
        {
            Trace.TraceError("Client message without message type: {0}", message);
        }
    }
    catch (Exception e)
    {
        Trace.TraceError("Error processing client message: {0} - {1}", e.Message, 
message);
    }
    ResendDataToClient();
}
```

The filter is applied here:

```
public void SendFiltered(IDictionary<string, object> message)
{
    if (    !message.ContainsKey("dspl")
        || (this.DeviceFilter != null
            && (
                String.Equals(this.DeviceFilter, "all", 
StringComparison.InvariantCultureIgnoreCase))
                || String.Equals(this.DeviceFilter, message["dspl"])))
    {
        this.Send(JsonConvert.SerializeObject(message));
    }
}
```

On the browser client, a chart and a table control handle most of the UI. Code in Azure\ConnectTheDotsWebSites\js\connectthedots.js receives the real-time data from the web site through the web socket, and applies it to the controls, i.e.

```
websocket.onmessage = function (event)
{
    try
    {
        var eventObject = JSON.parse(event.data);
    }
    catch (e)
    {
        $('#messages').prepend('<div>Malformed message: '+event.data+"</div>");
    }

    try
    {
        if (eventObject.alerttype != null)
        {
            var table = $('#alertTable').DataTable();

…
                table.row.add([
                    eventObject.alerttype+' '+ eventObject.dsplalert,
                    startTime,
                    endTime,
                    eventObject.tempmax,
                    eventObject.tempavg,
                    eventObject.tempmin,
                ]).draw();
            }
        }
        else
        {
            if (eventObject.tempavg != null)
            {
                AddPointToChartSeries(myChart, 0, eventObject.time,
eventObject.tempavg);
            }
…
        }
    }
    catch (e)
    {
        $('#messages').prepend('<div>Error processing message: ' + e.message +
"</div>");
    }
}
```

Deployment Tool

The deployment tool (Azure\ConnectTheDotsDeploy\Program.cs) creates (some of) the Azure Resources that are needed for ConnectTheDots. To do so, it needs

a) Management credentials to an Azure subscription (in the form of a management cert embedded in a .publishsettings file).
b) A unique name (prefix) to be used for the Azure resources.

It uses the Azure Management SDK and underlying REST APIs to create a Service Bus namespace, Event Hubs as well as SAS keys:

```
// Obtain management via .publishsettings file from
https://manage.windowsazure.com/publishsettings/index?schemaversion=2.0
var creds = new CertificateCloudCredentials(SubscriptionId,
ManagementCertificate);

// Create Namespace
var sbMgmt = new ServiceBusManagementClient(creds);
…
// Get the namespace connection string
var nsDescription = sbMgmt.Namespaces.GetNamespaceDescription(SBNamespace);
var nsConnectionString = nsDescription.NamespaceDescriptions.First(
    (d) => String.Equals(d.AuthorizationType, "SharedAccessAuthorization")
    ).ConnectionString;
// Create EHs + device keys + consumer groups (WebSite) + consumer keys (WebSite*)
var nsManager = NamespaceManager.CreateFromConnectionString(nsConnectionString);
var ehDescriptionDevices = new EventHubDescription(EventHubNameDevices)
{
    PartitionCount = 8,
};
ehDescriptionDevices.Authorization.Add(new SharedAccessAuthorizationRule("D1", new
List<AccessRights> { AccessRights.Send }));
…
ehDescriptionDevices.Authorization.Add(new
SharedAccessAuthorizationRule("WebSite", new List<AccessRights> {
AccessRights.Listen }));
ehDescriptionDevices.Authorization.Add(new
SharedAccessAuthorizationRule("StreamingAnalytics", new List<AccessRights> {
AccessRights.Manage, AccessRights.Listen, AccessRights.Send }));

EventHubDescription ehDevices = null;
…
        ehDevices = nsManager.CreateEventHubIfNotExists(ehDescriptionDevices);
…
nsManager.CreateConsumerGroupIfNotExists(EventHubNameDevices, "WebSite");
nsManager.CreateConsumerGroupIfNotExists(EventHubNameDevices, "WebSiteLocal");

…
// Create Storage Account for Event Hub Processor
var stgMgmt = new StorageManagementClient(creds);
    var resultStg = stgMgmt.StorageAccounts.Create(
        new StorageAccountCreateParameters { Name =
StorageAccountName.ToLowerInvariant(), Location = Location, AccountType =
"Standard_LRS" });

…
var keyResponse =
stgMgmt.StorageAccounts.GetKeys(StorageAccountName.ToLowerInvariant());
…
var storageKey = keyResponse.PrimaryKey;
```

The tool displays the most important credentials, and also adds them to the web.config file for the Web Site. The tool can be re-run in case of failure (it detects which resources already exist) as well as to retrieve credentials for an already existing set of resources.

Troubleshooting tips:

Raspberry Pi:

- Diagnostics logs: `tail /home/RaspberryPiGateway/RaspberryPiGateway.log`
- You can run the gateway exe manually, i.e.

```
cd /home/pi/RaspberryPiGateway
mono RaspberryPiGateway.exe -forever -deviceid C8CA5B13-A550-4FF0-B823-46D8A2640880 -
devicename "R-PI 001" -address amqps://D1:xxxxxxxx@yyyyyyyy.servicebus.windows.net -
target ehdevices -tracelevel information
```

  Additional options:
  - **-tracelevel verbose**: logs/shows more details
  - **-tracelevel error**: logs/shows only errors (use this if you plan to run your Gateway for a long time, otherwise your SD Card will fill up)
  - **-serial <port>**: use a different port than the default /dev/ttyACM0, i.e. `-serial /dev/ttyACM1` (or `-serial COM10` for use on Windows).
  - **-frequency <milliseconds>**: send data to the cloud at a different time interval. Default: 1000 (1 second).

    If you want to "stress test" your Raspberry PI/network connection, uncomment "`#define LOG_MESSAGE_RATE`" Devices\RaspberryPIGateway\Program.cs and use –frequency 0 (you may also want to un-define the TRACE symbol in VS build options, or use -tracelevel error, or you may be measuring the speed of your SD Card). [ad hoc observations: expect ~200-400 msg/s on a R-PI B+. EventHub will throttle at 1000 msg/s per partition.].

- Make sure autorun.sh does not have Windows line endings (CR/LF). It will only run with Unix line endings (LF). See here for some tips.
- To stop the gateway program: `killall mono`
  - To see if it's running: `ps -C mono`

Azure:
- Failure while running ConnectTheDotsCloudDeploy.exe or forgot to save device or management connection strings: Just rerun the tool. It will only create missing resources, and obtain connection strings.
- No data coming into browser: make sure WebSockets are enabled in Azure Web Services (see above).
- Verify if data comes into Event Hub either from the devices or Azure Streaming Analytics: Use Service Bus Explorer, connect with the management connection string returned by ConnectTheDotsCloudDeploy.exe:

```
D:\Source\Repos\connectthedots\Azure\ConnectTheDotsCloudDeploy\bin\Debug>
ConnectTheDotsCloudDeploy.exe -n ctdtest1 -ps d:\temporary\MarkusHMSDN.publishsettings
Creating Service Bus namespace ctdtest1-ns in location Central US
…
Service Bus management connection string (i.e. for use in Service Bus Explorer):
Endpoint=sb://ctdtest1-
ns.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=
zzzzzzz

…
```

- View web site trace logs: Server Explorer, Azure, Web Site, right-click on the site and click “View streaming logs”). You should see error and informational traces that can help you pinpoint issues.
- Run the web site locally (just hit “F5” in VS) and observe the debug output window for any error or informational traces [You can safely run locally even while you are running a site in the cloud. The local site uses different Event Hub consumer groups.]