Windows 10 IoT – BLE IoT Hub

The use of Bluetooth has made untethered communication over short distances very easy. This lab demo shows how to submit BLE Beacon data to an Azure IoT Hub.

APPROXIMATE TIME (EXCLUDING PREPARATION WORK): 60 Minutes

PREREQUISITES:

- Setup / Install Windows 10 (v 10.0.10240) or higher
- Download Windows 10 IoT Core Dashboard (http://go.microsoft.com/fwlink/?LinkID=708576)
- Setup / Install Visual Studio 2015 Update 1 (You can download Visual Studio Community Edition For Free: http://go.microsoft.com/fwlink/?LinkID=534599)
- Install IoT Core Project Templates (https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ad-82a5-a88ac4b01dec)
- Azure Device Explorer (https://github.com/Azure/azure-iot-sdks/releases)
- Azure Subscription (A 30 day trial subscription is available)

PARTS LIST:

- Raspberry Pi 2
- Supported Bluetooth Dongle (http://ms-iot.github.io/content/en-US/win10/SupportedInterfaces.htm#Bluetooth-Dongles)
- Bluetooth Beacon (this lab is programmed to use XY Find It BLE Beacons http://www.xyfindit.com/)

HARDWARE SETUP:

There is no specific hardware setup in this lab, it is primarily code. The Raspberry Pi should be setup and connected to a monitor, network cable plugged in, speakers plugged in, and Bluetooth dongle plugged into an available USB Port

SOFTWARE SETUP:

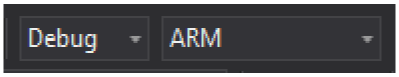
A) Preparing The Device
The device should already have an image of Windows 10 IoT core on it, but if it does not, you can always install it onto the SD Card.
   1. Launch Windows 10 IoT Core Dashboard
   2. Click "Set up a new device" in the left hand menu
   3. Select the appropriate Device Type and OS Image
   4. Insert the Micro SD card into your computer, click "Download and Install" and follow the directions

B) Boot The Device
It may take ~5 minutes for the device to boot up for the first time.

C) Connecting To The Device
When online, the device should appear under the "My Devices" tab on the IoT Core Dashboard.
   1. Note the IP address of the machine and either connect to http://<IP_ADDRESS>:8080 or click the globe under "Open in device Portal" for that specific device
   2. The default user name is Administrator
   3. The default password is p@ssw0rd

D) Deploying To The Device

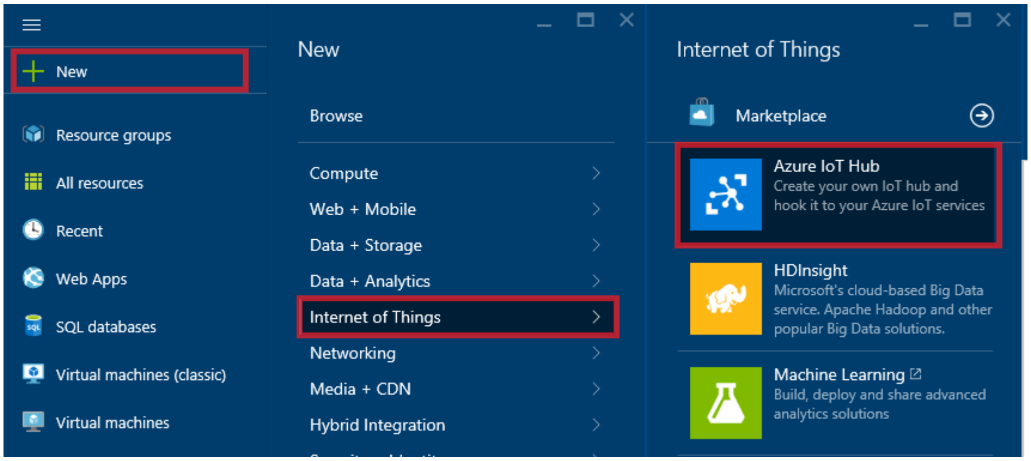
1. In Visual Studio, on the build bar, make sure that the appropriate build / architecture is selected:


Debug ARM

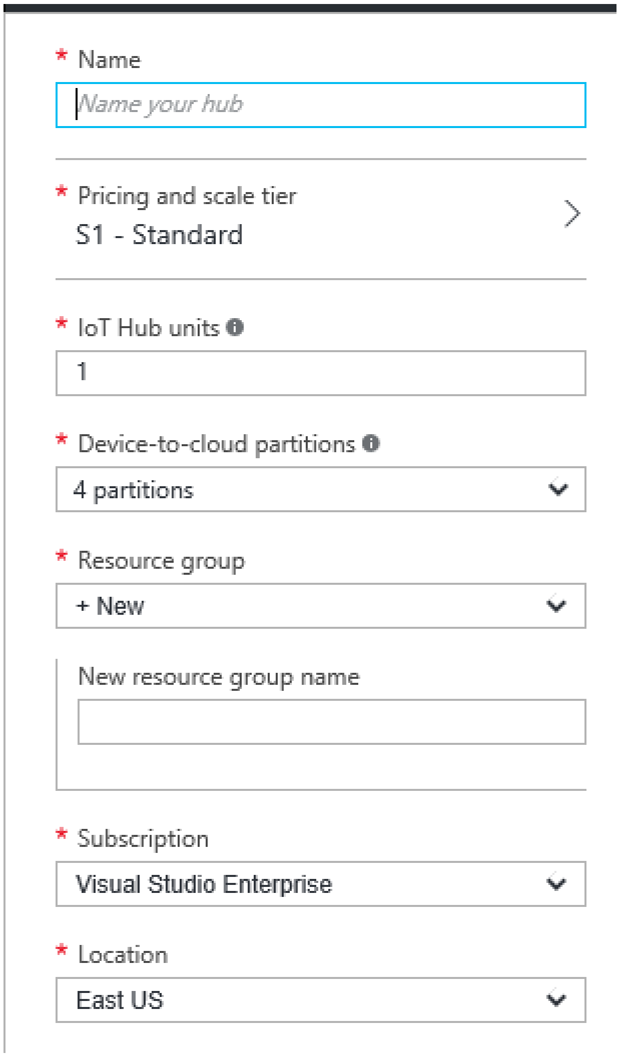
2. Right click the project name and select properties
3. In the settings dialog, click the Debug option
4. Under start options (If the device shows up under IoT Core Dashboard, you should be able to hit “Find” to populate the data
   i. Target Device: Remote Machine
   ii. Remote Machine: <IP ADDRESS>
   iii. Authentication Mode: Universal
5. Build / Deploy the project to the device

IoT Hub Setup

1. Login to Azure Portal (https://portal.azure.com/)
2. New -> Internet of Things -> Azure IoT Hub



3. Specify the configuration for the Hub



* Name

Name your hub

* Pricing and scale tier
S1 - Standard

* IoT Hub units
1

* Device-to-cloud partitions
4 partitions

* Resource group
+ New

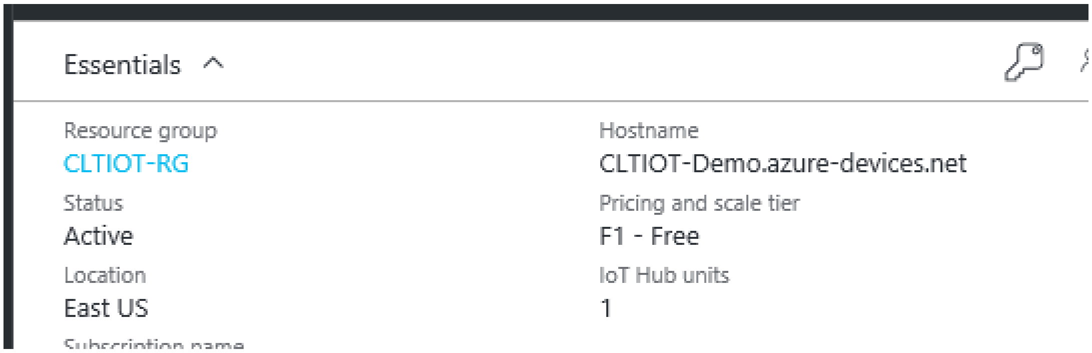
New resource group name

* Subscription
Visual Studio Enterprise

* Location
East US

   a. Name: Name of Hub
   b. Pricing: Select F1 – Free
   c. Resource Group: Resource Group Name
   d. Location: Pick Location

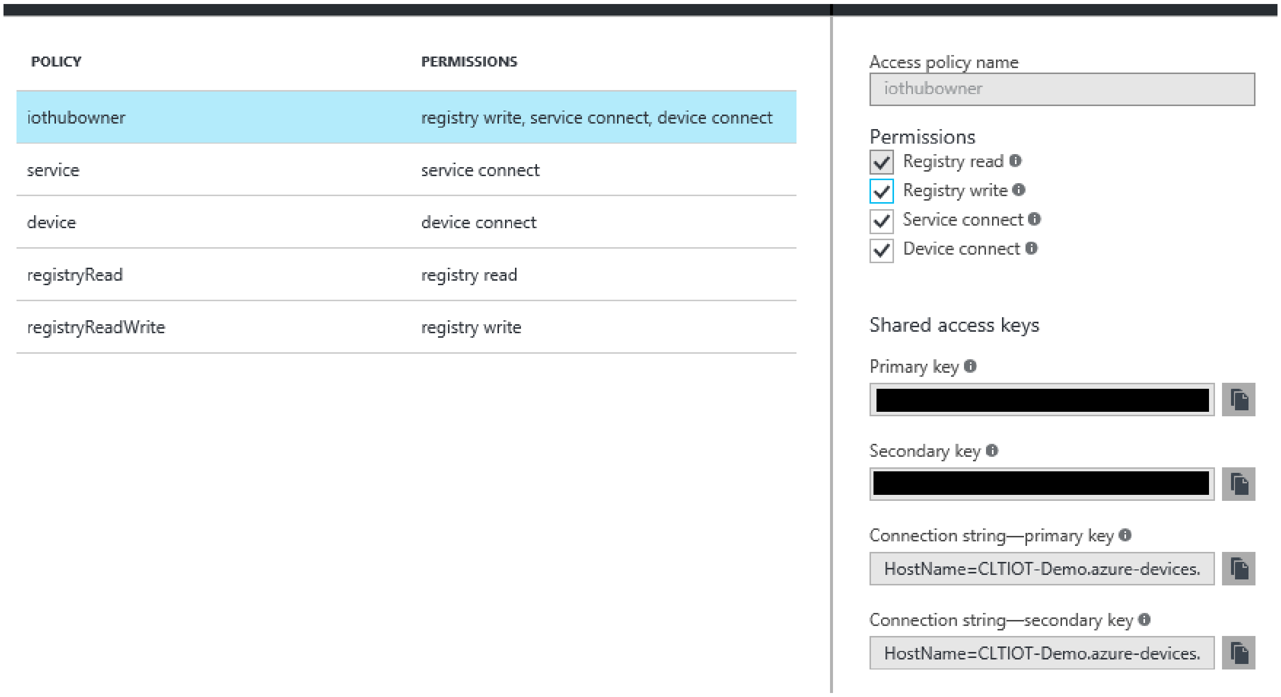
4. Click Create
5. When it is created, take note of the configuration properties and select key icon



Essentials

Resource group CLTIOT-RG	Hostname CLTIOT-Demo.azure-devices.net
Status Active	Pricing and scale tier F1 - Free
Location East US	IoT Hub units 1
Subscription name	

6. Select the Shared access policy of iothubowner and make note of the connection strings and keys. Make sure that you protect the keys as this will allow for devices to connect to your hub



POLICY	PERMISSIONS
iothubowner	registry write, service connect, device connect
service	service connect
device	device connect
registryRead	registry read
registryReadWrite	registry write

Access policy name: iothubowner

Permissions
- ☑ Registry read
- ☑ Registry write
- ☑ Service connect
- ☑ Device connect

Shared access keys

Primary key: [redacted]

Secondary key: [redacted]

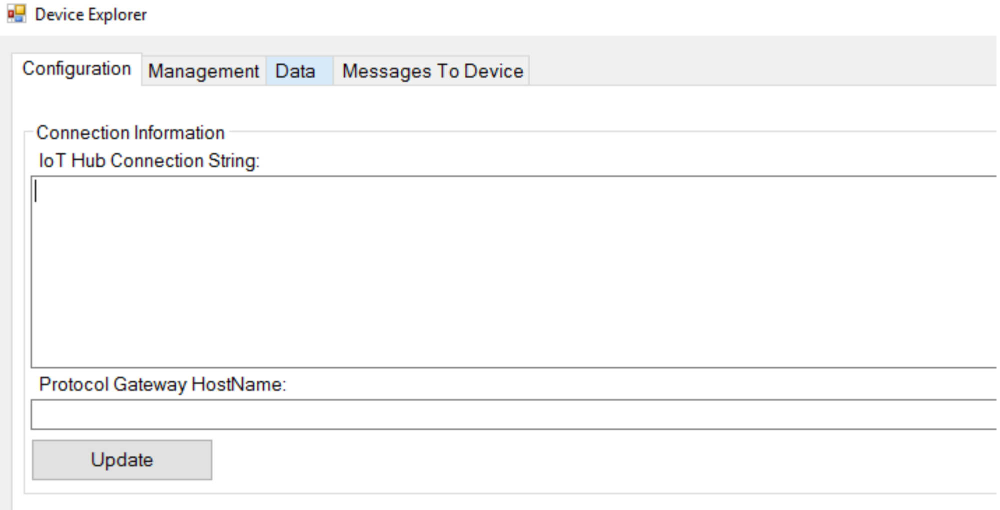
Connection string—primary key: HostName=CLTIOT-Demo.azure-devices.

Connection string—secondary key: HostName=CLTIOT-Demo.azure-devices.

7. At this point, devices can now be registered to connect to the hub

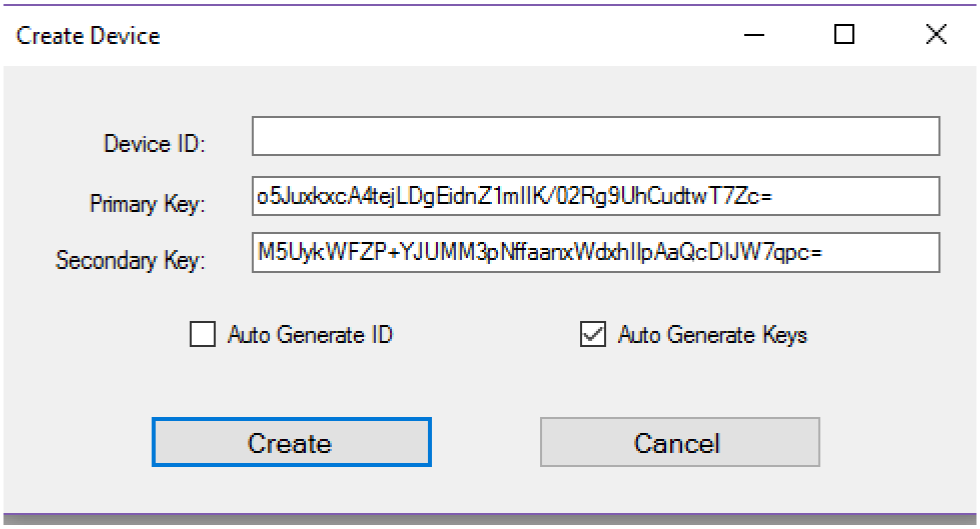
Registering a Device

1. Azure Launch Device Explorer
2. On the Configuration tab, paste the Connection string from step 6 above and then click Update



3. Click the Management Tab to manage all devices connected

4. Click Create to register a new device. The keys will be automatically filled in, just give it a unique device name



Create Device

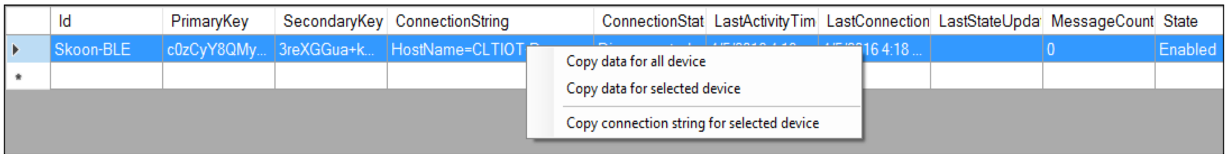
Device ID:

Primary Key: o5JuxkxcA4tejLDgEidnZ1mIIK/02Rg9UhCudtwT7Zc=

Secondary Key: M5UykWFZP+YJUMM3pNffaanxWdxhIlpAaQcDIJW7qpc=

□ Auto Generate ID ☑ Auto Generate Keys

5. Right click on a device to get the connection string for the device



	Id	PrimaryKey	SecondaryKey	ConnectionString	ConnectionStat	LastActivityTim	LastConnection	LastStateUpda	MessageCount	State
▸	Skoon-BLE	c0zCyY8QMy...	3reXGGua+k...	HostName=CLTIOT	[illegible]	[illegible]	16 4:18 ...		0	Enabled
*										

Copy data for all device
Copy data for selected device
Copy connection string for selected device

6. Use this in the appropriate place in the code.
7. Devices can be added / removed / managed via the command line tool as well (https://github.com/Azure/azure-iot-sdks/blob/master/tools/DeviceExplorer/readme.md)

Application

This application uses the DeviceWatcher class to look for and enumerate devices. It constantly looks for the signal strength to determine whether or not to pair the device and speak the text. The sample does have a visual element that binds to a back end collection to show the available devices and their current status.

A) Create A New Project
   A) Launch Visual Studio
   B) Select "New Project"
   C) Under "Templates", select "Visual C#" and then "Blank App (Universal Windows)"
   D) Give the app a name and location. The code sample below assumes you have called your application "BT-BeaconApp"
   E) Open the Nuget Package Manager Console
   F) Add the following packages
      i. Install-package Newtonsoft.Json
      ii. Install-package Microsoft.Azure.Devices.Client
      iii. Install-package pclcrypto
B) Adding The Code

Create Class For Displaying Device Information

1. Right click project name -> Add -> Class
2. Name the file DisplayHelper.cs and click Add

3. Remove the scaffolded code and start with the following blank class file

```
Using Newtonsoft.Json
using System;
using System.Collections.Generic;
using System.ComponentModel;
using Windows.Devices.Enumeration;
using Windows.UI.Xaml.Media.Imaging;

namespace BT_BeaconApp
{

}
```

4. Add a new class to the namespace that will handle the additional properties that we want to get from the endpoint device. In this case, we specifically care about the signal strength

```
public class DeviceInformationDisplay : INotifyPropertyChanged
{
    private DeviceInformation deviceInfo;

    public DeviceInformationDisplay(DeviceInformation deviceInfoIn)
    {
        deviceInfo = deviceInfoIn;
        UpdateGlyphBitmapImage();
    }

}
```

5. Add a new class to the namespace that will manage all of the properties that the UI will display. This class inherits from INotifyPropertyChanged to automatically fire an event when the property changes.

```
public static class DeviceProperties
{
    public static List<string> AssociationEndpointProperties
    {
        get
        {
            List<string> properties = new List<string>();
            properties.Add("System.Devices.Aep.SignalStrength");

            return properties;
        }
    }
}
```

6. Add the public properties to the class

```
    public DeviceInformationKind Kind
    {
        get { return deviceInfo.Kind; }
    }

    public string Id
    {
        get { return deviceInfo.Id; }
    }

    public string Name
    {
        get { return deviceInfo.Name; }
    }

    public BitmapImage GlyphBitmapImage
    {
        get; private set;
    }

    public bool IsPairing
    {
        get; set;
    }

    public bool CanPair
    {
        get { return deviceInfo.Pairing.CanPair; }
    }

    public bool IsPaired
    {
        get { return deviceInfo.Pairing.IsPaired; }
    }

    public int SignalStrength
    {
        get
        {
            int val = int.MinValue;
            int.TryParse(deviceInfo.Properties["System.Devices.Aep.SignalStrength"]
                 .ToString(), out val);

            return val;
        }
    }

    public IReadOnlyDictionary<string, object> Properties
    {
        get { return deviceInfo.Properties; }
    }

    public DeviceInformation DeviceInformation
    {
        get { return deviceInfo; }

        private set { deviceInfo = value; }
    }
```

7. Add additional methods and event handlers

```
 public void Update(DeviceInformationUpdate deviceInfoUpdate)
 {
     deviceInfo.Update(deviceInfoUpdate);

     OnPropertyChanged("Kind");
     OnPropertyChanged("Id");
     OnPropertyChanged("Name");
     OnPropertyChanged("DeviceInformation");
     OnPropertyChanged("CanPair");
     OnPropertyChanged("IsPaired");
     OnPropertyChanged("SignalStrength");

     UpdateGlyphBitmapImage();
 }

 private async void UpdateGlyphBitmapImage()
 {
     DeviceThumbnail deviceThumbnail = await deviceInfo.GetGlyphThumbnailAsync();
     BitmapImage glyphBitmapImage = new BitmapImage();
     await glyphBitmapImage.SetSourceAsync(deviceThumbnail);
     GlyphBitmapImage = glyphBitmapImage;
     OnPropertyChanged("GlyphBitmapImage");
 }

 public event PropertyChangedEventHandler PropertyChanged;
 protected void OnPropertyChanged(string name)
 {
     PropertyChangedEventHandler handler = PropertyChanged;
     if (handler != null)
     {
         handler(this, new PropertyChangedEventArgs(name));
     }
 }

public string ToJson()
{
    var obj = new
    {
         time = DateTime.UtcNow.ToString("o"),
         deviceName = this.Name,
         signalStrength = this.SignalStrength,
    };
    return JsonConvert.SerializeObject(obj);
}
```

Add UI Code

1. Open up MainPage.xaml
2. Add the following template to display data about each beacon

```
<Page.Resources>

        <DataTemplate x:Key="ResultsListViewTemplate">
            <Grid Margin="5">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*" MinWidth="100"/>
                </Grid.ColumnDefinitions>
                <Border Grid.Column="0" Height="40" Width="40" Margin="5" VerticalAlignment="Top">
                    <Image Source="{Binding Path=GlyphBitmapImage}"
                           Stretch="UniformToFill"/>
                </Border>
                <Border Grid.Column="1" Margin="5">
                    <StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="Name:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=Name}" FontWeight="Bold" TextWrapping="WrapWholeWords"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="Id:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=Id}" TextWrapping="Wrap"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="SignalStrength:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=SignalStrength}"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="CanPair:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=CanPair}"/>
                        </StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="IsPaired:" Margin="0,0,5,0"/>
                            <TextBlock Text="{Binding Path=IsPaired}"/>
                        </StackPanel>
                    </StackPanel>
                </Border>
            </Grid>
        </DataTemplate>

    </Page.Resources>
```

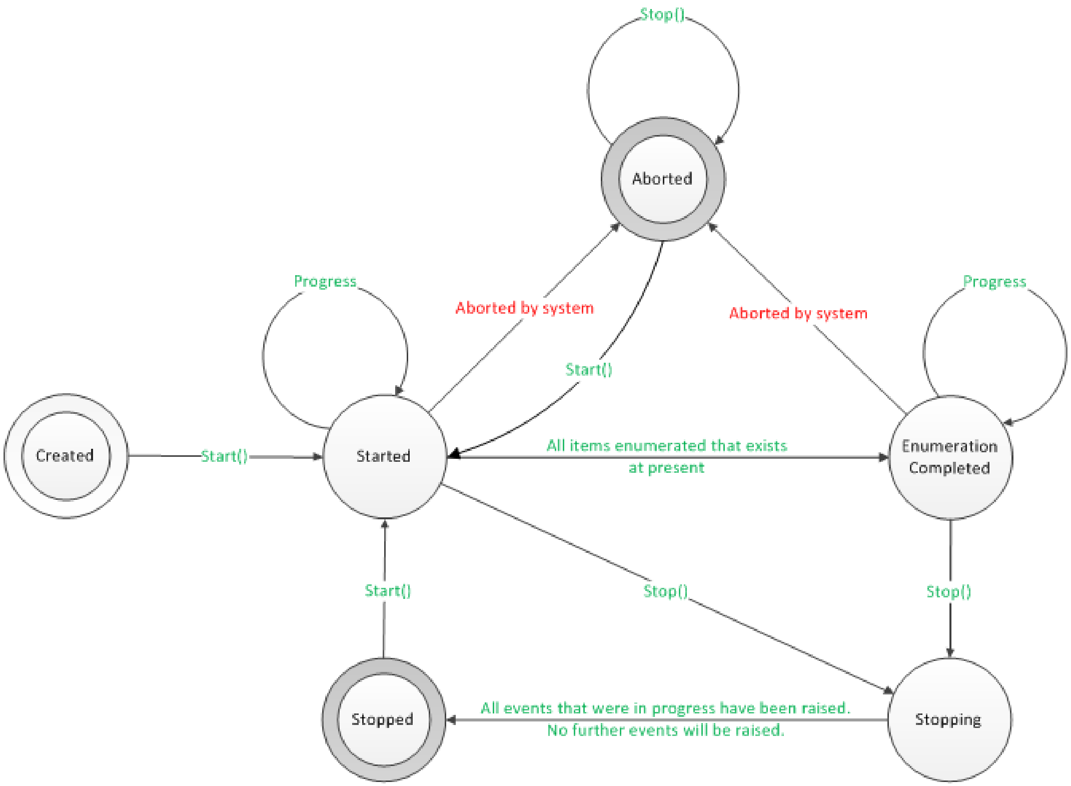
3. Replace the existing Grid code with the following code

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <ScrollViewer Grid.Row="0" VerticalScrollMode="Auto" VerticalScrollBarVisibility="Auto">
        <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top" Margin="8,5,15,0">
            <TextBlock Text="Beacons:" Style="{StaticResource SampleHeaderTextStyle}"/>

            <Border BorderBrush="AntiqueWhite" BorderThickness="1">
                <ListView x:Name="resultsListView"
                          ItemTemplate="{StaticResource ResultsListViewTemplate}"
                          ItemsSource="{Binding Path=ResultCollection}"
                          MaxHeight="450">
                </ListView>
            </Border>
        </StackPanel>
    </ScrollViewer>
</Grid>
```

Add UI Code Behind

The device watcher is the primary process that is running and is alerted to changes within the devices in the range. From MSDN, this is the general behavior that it uses for looking at devices:



1. Open MainPage.xaml.cs

2. Remove the original code and start with the following template. Be sure to replace the <REPLACE> code with the appropriate device connection string from the previous section.

```
using Microsoft.Azure.Devices.Client;
using System;
using System.Collections.ObjectModel;
using System.Diagnostics;
using System.Linq;
using System.Text;
using Windows.Devices.Enumeration;
using Windows.UI.Core;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace BT_BeaconApp
{
    public sealed partial class MainPage : Page
    {
        private static string CXN_STRING = "<REPLACE>";

        private DeviceClient _devClient = null;
        private DeviceWatcher _deviceWatcher = null;        // Device watcher for new BLE devices
        private bool _restartWatcher = false;               // Restart watcher upon completion

        // Collect of devices bound to the UI
        public ObservableCollection<DeviceInformationDisplay> ResultCollection
        {
            get;
            private set;
        }

        public MainPage()
        {
            this.InitializeComponent();
        }

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            // Initialize the objects
            ResultCollection = new ObservableCollection<DeviceInformationDisplay>();
            DataContext = this;

            _devClient = DeviceClient.CreateFromConnectionString(CXN_STRING, TransportType.Http1);

            StartDeviceWatcher();
        }

        protected override void OnNavigatedFrom(NavigationEventArgs e)
        {
            StopDeviceWatcher();
        }
    }
}
```

3. Add code to start / stop the watcher

```
        private void StartDeviceWatcher()
        {
            ResultCollection.Clear();

            // Currently Bluetooth APIs don't provide a selector to get ALL devices that are both paired and non-
paired.  Typically you wouldn't need this for common scenarios,
            // but it's convenient to demonstrate the various sample scenarios.
            string selector = "(System.Devices.Aep.ProtocolId:=\"{bb7bb05e-5972-42b5-94fc-76eaa7084d49}\")" + " AND
(System.Devices.Aep.CanPair:=System.StructuredQueryType.Boolean#True OR
System.Devices.Aep.IsPaired:=System.StructuredQueryType.Boolean#True)";

            // Kind is specified in the selector info
            _deviceWatcher = DeviceInformation.CreateWatcher(
                selector,
                DeviceProperties.AssociationEndpointProperties,
                DeviceInformationKind.AssociationEndpoint);

            // Hook up events for the watcher
            _deviceWatcher.Added += OnDeviceAdded;
            _deviceWatcher.Updated += OnDeviceUpdated;
            _deviceWatcher.Removed += OnDeviceRemoved;
            _deviceWatcher.EnumerationCompleted += OnEnumerationCompleted;
            _deviceWatcher.Stopped += OnStopped;

            _deviceWatcher.Start();
        }

        private void StopDeviceWatcher()
        {
            if (null != _deviceWatcher)
            {
                // First unhook all event handlers except the stopped handler. This ensures our
                // event handlers don't get called after stop, as stop won't block for any "in flight"
                // event handler calls.  We leave the stopped handler as it's guaranteed to only be called
                // once and we'll use it to know when the query is completely stopped.
                _deviceWatcher.Added -= OnDeviceAdded;
                _deviceWatcher.Updated -= OnDeviceUpdated;
                _deviceWatcher.Removed -= OnDeviceRemoved;
                _deviceWatcher.EnumerationCompleted -= OnEnumerationCompleted;

                if (DeviceWatcherStatus.Started == _deviceWatcher.Status ||
                    DeviceWatcherStatus.EnumerationCompleted == _deviceWatcher.Status)
                {
                    _deviceWatcher.Stop();
                }
            }
        }
```

4. Add code to handle posting data to the Azure IoT Hub.

```
        private async void PostBeaconData(DeviceInformationDisplay deviceInfoDisplay)
        {
            if(deviceInfoDisplay.Name.StartsWith("XY"))
            {
                if(null != _devClient)
                {
                    try
                    {
                        string jsonText = deviceInfoDisplay.ToJson();
                        Message msg = new Message(Encoding.UTF8.GetBytes(jsonText));
                        await _devClient.SendEventAsync(msg);

                        Debug.WriteLine("Message Sent: {0}", jsonText);
                    }
                    catch (Exception ex)
                    {
                        Debug.WriteLine("Exception when sending message:" + ex.Message);
                    }
                }
            }
        }
```

5. Add DeviceAdd event

```
        private async void OnDeviceAdded(DeviceWatcher watcher, DeviceInformation deviceInfoAdded)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                DeviceInformationDisplay deviceInfoDisplay = new DeviceInformationDisplay(deviceInfoAdded);

                if (!ResultCollection.Any(p => p.Name == deviceInfoAdded.Name))
                {
                    ResultCollection.Add(deviceInfoDisplay);
                    Debug.WriteLine("{0} devices found.", ResultCollection.Count);
                }

                PostBeaconData(deviceInfoDisplay);
            });
        }
```

6. Add DeviceUpdated event

```
        private async void OnDeviceUpdated(DeviceWatcher watcher, DeviceInformationUpdate deviceInfoUpdate)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI 
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                // Find the corresponding updated DeviceInformation in the collection and pass the update object
                // to the Update method of the existing DeviceInformation. This automatically updates the object
                // for us.
                foreach (DeviceInformationDisplay deviceInfoDisplay in ResultCollection)
                {
                    if (deviceInfoDisplay.Id == deviceInfoUpdate.Id)
                    {
                        PostBeaconData(deviceInfoDisplay);
                        deviceInfoDisplay.Update(deviceInfoUpdate);
                        break;
                    }
                }
            });
        }
```

7. Add DeviceRemoved event

```
private async void OnDeviceRemoved(DeviceWatcher watcher, DeviceInformationUpdate deviceInfoRemoved)
        {
            // Since we have the collection databound to a UI element, we need to update the collection on the UI 
thread.
            await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
            {
                // Find the corresponding DeviceInformation in the collection and remove it
                foreach (DeviceInformationDisplay deviceInfoDisplay in ResultCollection)
                {
                    if (deviceInfoDisplay.Id == deviceInfoRemoved.Id)
                    {
                        ResultCollection.Remove(deviceInfoDisplay);
                        break;
                    }
                }
                Debug.WriteLine("{0} devices found.", ResultCollection.Count);
            });
        }
```

8. Add EnumComplete Event

```
        private void OnEnumerationCompleted(DeviceWatcher watcher, object args)
        {
            Debug.WriteLine("{0} devices found. Enumeration completed. Watching for updates...", 
ResultCollection.Count);
            _restartWatcher = true;
            _deviceWatcher.Stop();
        }
```

9. Add Stopped Event

```
        private async void OnStopped(DeviceWatcher watcher, object args)
        {
            Debug.WriteLine("{0} devices found. Watcher {1}.",
                ResultCollection.Count,
                DeviceWatcherStatus.Aborted == watcher.Status ? "aborted" : "stopped");

            if (_restartWatcher)
            {
                await Dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
                {
                    _restartWatcher = false;
                    _deviceWatcher.Start();
                });
            }
        }
```

10. Add Pairing Event

```
        private async void OnPairingRequested(DeviceInformationCustomPairing sender, DevicePairingRequestedEventArgs
args)
        {
            switch (args.PairingKind)
            {
                case DevicePairingKinds.ConfirmOnly:
                    // Windows itself will pop the confirmation dialog as part of "consent" if this is running on
Desktop or Mobile
                    // If this is an App for 'Windows IoT Core' where there is no Windows Consent UX, you may want to
provide your own confirmation.
                    args.Accept();
                    break;
            }
        }
```

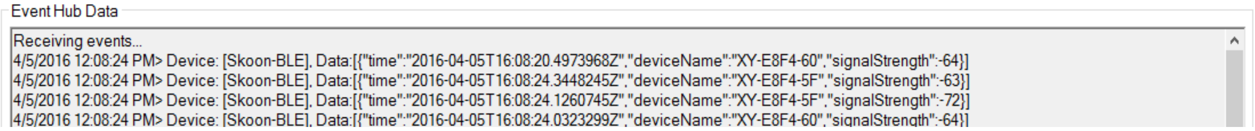
C) Understanding The Code

You should now be able to build and deploy your code. Once it is up and running, the code will start a watcher and look for BLE beacons. If it finds one that starts with the name “XY” then it will send signal strength data to the Azure IoT Hub.

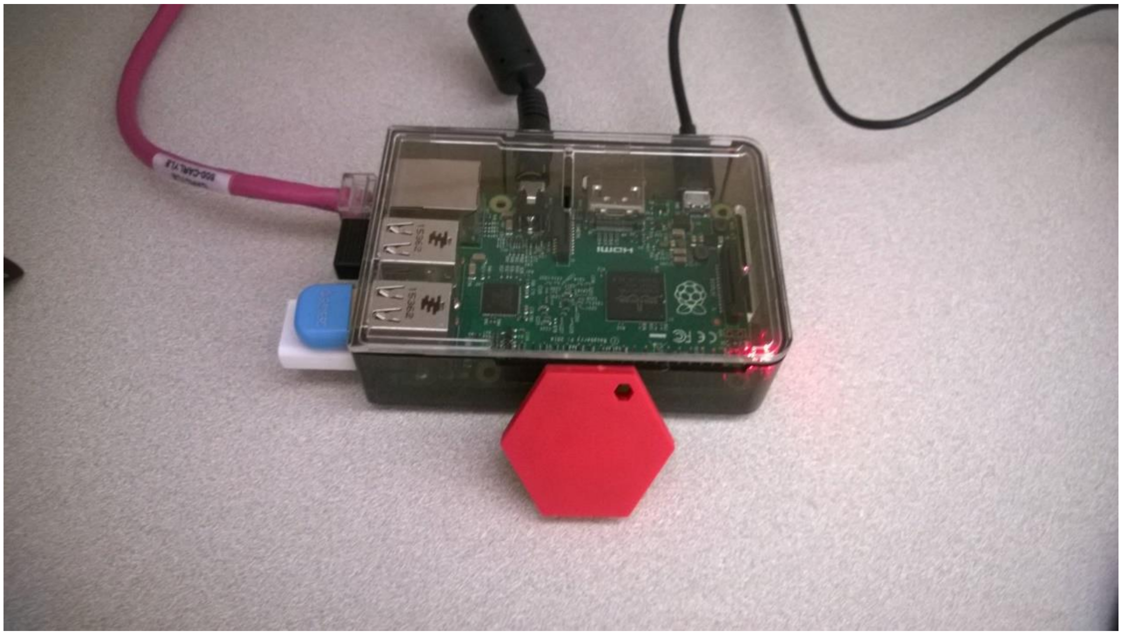
D) Watching Azure Data

You can use the Device Watcher to display data being sent by the device

1. Click the Data tab and select the Device ID you want to monitor
2. Click the Monitor button to start monitoring the device data
3. Run the IoT application
4. When an event is captured, you should see it in the output pane



```
Receiving events...
4/5/2016 12:08:24 PM> Device: [Skoon-BLE], Data:[{"time":"2016-04-05T16:08:20.4973968Z","deviceName":"XY-E8F4-60","signalStrength":-64}]
4/5/2016 12:08:24 PM> Device: [Skoon-BLE], Data:[{"time":"2016-04-05T16:08:24.3448245Z","deviceName":"XY-E8F4-5F","signalStrength":-63}]
4/5/2016 12:08:24 PM> Device: [Skoon-BLE], Data:[{"time":"2016-04-05T16:08:24.1260745Z","deviceName":"XY-E8F4-5F","signalStrength":-72}]
4/5/2016 12:08:24 PM> Device: [Skoon-BLE], Data:[{"time":"2016-04-05T16:08:24.0323299Z","deviceName":"XY-E8F4-60","signalStrength":-64}]
```



http://www.skoonstudios.com

Last Revised: April 5, 2016