BLUEMETAL
An Insight company

d2c2d

Lab Workbook Three

Device to Cloud to Device - a workshop for learning about Windows 10 Core IoT device development, Azure IoT Hub, Stream Analytics and automating Azure using PowerShell

Workshop Overview

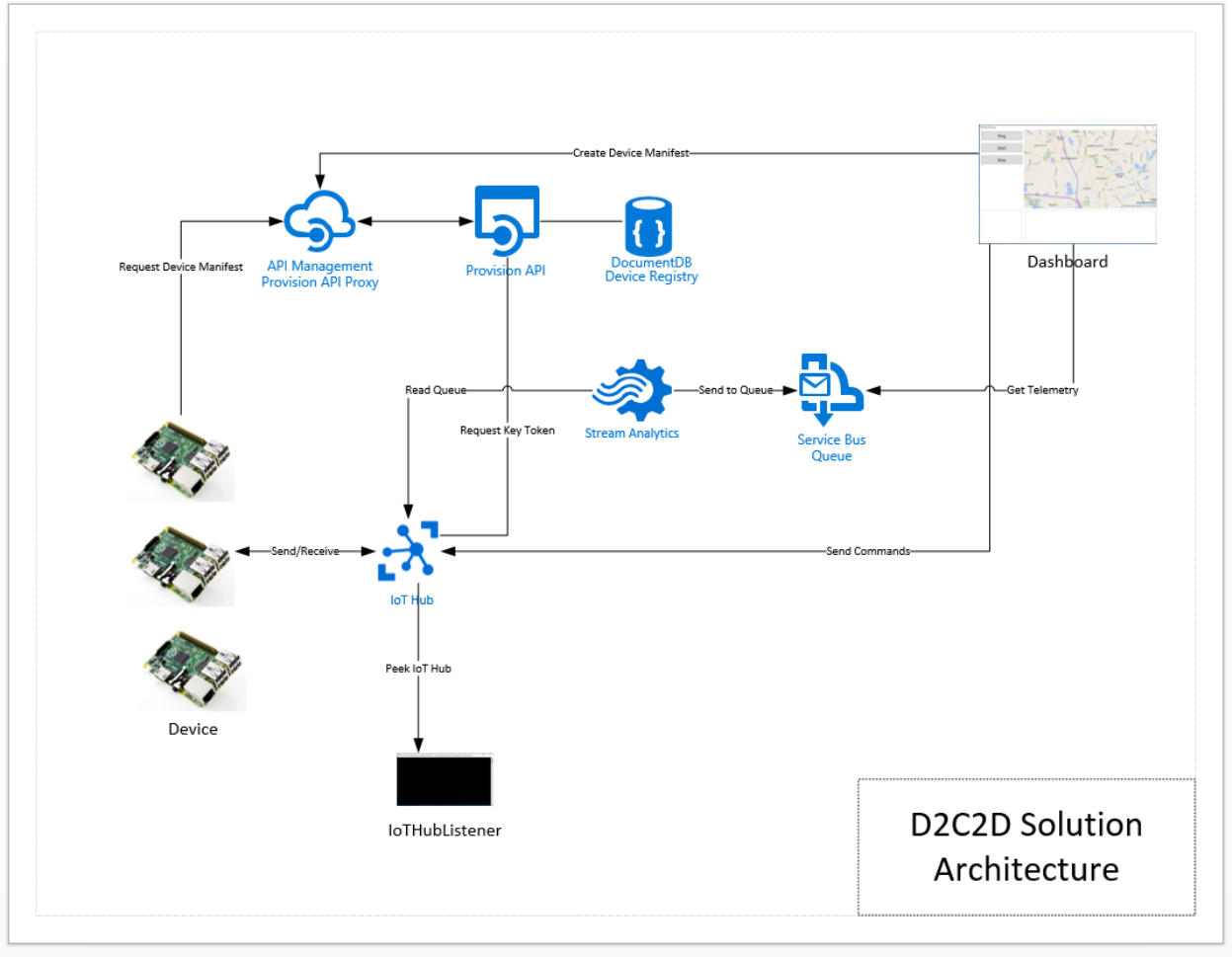
This training program provides foundational knowledge in how to architect and implement an IoT solution using Windows 10 Core IoT hardware devices and Azure IoT Hub and Stream Analytics. Both Device-to-Cloud and Cloud-to-Device communication patterns are discussed, designed, and implemented using best practices.

At the conclusion of this workshop you will have provisioned, using PowerShell, an Azure environment that contains IoT Hub, Stream Analytics Jobs that identify telemetry events and alarm states, and a Service Bus Namespace and set of message queues for backend integration.

You will also have developed a Windows 10 Core IoT application ("device") that sends telemetry and receives incoming commands from the cloud as well as a real-time dashboard that can communicate bi-directionally with the device (e.g., displaying telemetry readings and sending commands to the remote device).

Device Provisioning and IoT Hub monitoring and techniques for dynamic business rules are also covered.

.



Solution Architecture

The solution you will build and deploy consists of the following components:

- **Device** – a Windows 10 IoT Core IoT solution that dynamically connects to IoT hub and sends heartbeat and climate telemetry as well as responds to command from a dashboard. The device application can run on your local machine or be deployed to a Windows 10 Core IoT device, such as a Raspberry Pi.
- **Dashboard** – a Windows 10 WPF application that lists registered devices, maps location using Bing Maps, and displays incoming device telemetry and alarms.
- **Provision API** – a ReST API that provides endpoints for device and device manifest lookup via a unique serial number. The Dashboard application registers devices, and the Device application uses the API to retrieve its manifest.
- **IoT Hub Listener –** a debugging utility that provides visibility to messages arriving from the device.

and the following Azure Services

- **Service Bus** – two queues are defined: one is a target for all incoming messages, and the other receives messages representing an alarm state, for instance, an out-of-range value

- **IoT Hub –** IoT Hub provides device registration, incoming telemetry at scale, and cloud-to-device message services
- **DocumentDb** – DocumentDb is a NoSQL database service that is used for managing Device Manifests, i.e., a Device Registry
- **Stream Analytics –** the solution leverages two Stream Analytics jobs, one that handles all incoming messages and another that identifies alarm states and routes those messages to a second queue.

Lab Three Overview

In Lab three you will begin the implementation of your device. The device must be able to connect to the Provision API to retrieve its manifest and then use the key stored there to connect to IoT Hub. Once that is working, the device can start to send a heartbeat message using the Ping Class you defined in Lab 1. Next we will update the Dashboard to receive these Ping Messages.

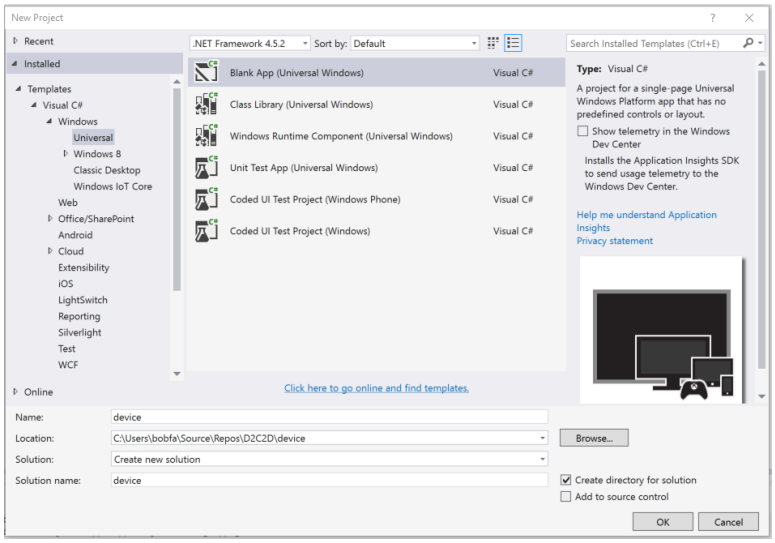
Lab

Step	*Details*
1	**Create the Device Solution**

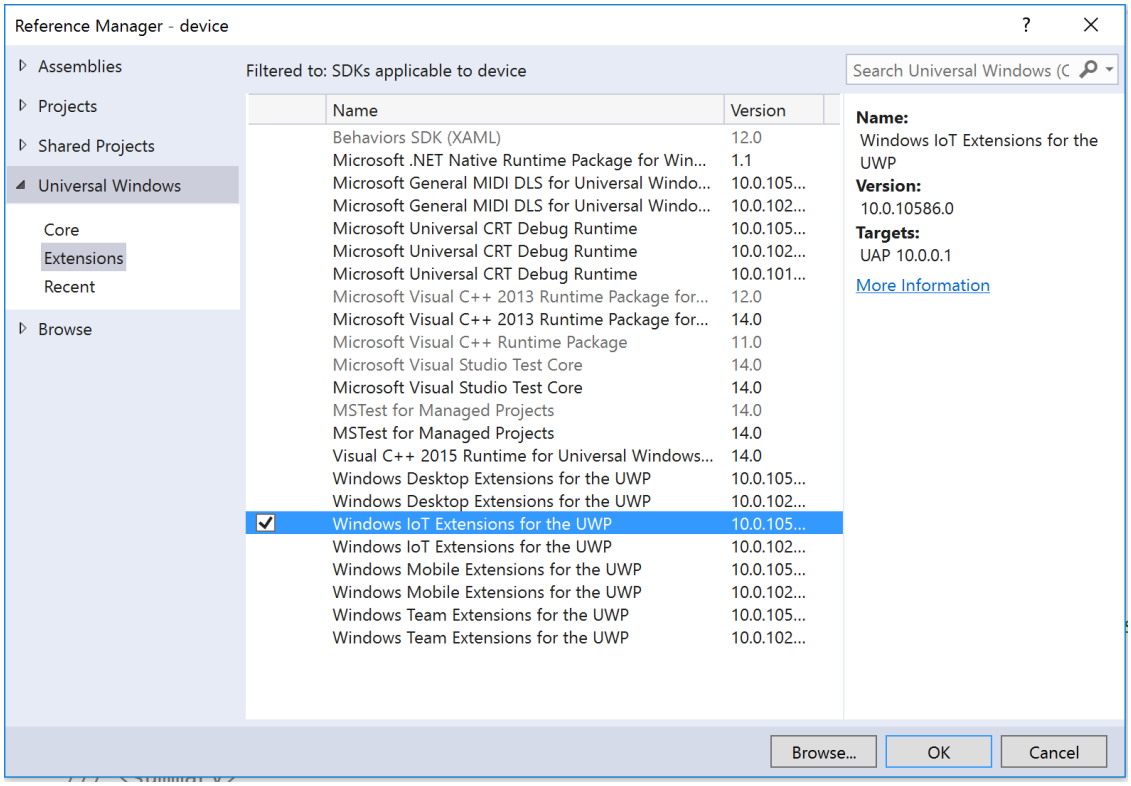
Note: A completed solution can be found in the repo's *device/pre-baked* folder. You will need to modify the MainPage.xaml.cs file to add your device ID and API link. See the comments in the code for further instructions, or scroll below to the paragraph beginning with "*To retrieve the Device Serial…*"

The Device solution will be built on a .NET Core Universal App foundation. We will add in the Windows 10 Core IoT libraries and then reference the IoT and Message Model libraries needed for our implementation.

- Create a solution call 'device' in the device folder of the repo using the Windows 10 Universal Blank App template



- If prompted for the target and minimum platform versions, accept the defaults and continue.
- Add a Reference to the Windows IoT Extensions for UWP 10.0.10586.0



- Replace the entire contents of MainPage.xaml file with the markup below

```
<Page
      x:Class="device.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="using:device"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      mc:Ignorable="d"
      Loaded="MainPage_OnLoaded">

      <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
          <Grid.RowDefinitions>
              <RowDefinition Height="50*"/>
              <RowDefinition Height="50*"/>
          </Grid.RowDefinitions>

          <TextBlock Grid.Row="0"
                     HorizontalAlignment="Center"
                     VerticalAlignment="Center"
                     FontSize="24">Windows 10 IoT Device</TextBlock>
          <StackPanel Grid.Row="1" Margin="10,10,10,10">
              <TextBox x:Name="Status" Margin="10" IsReadOnly="True" TextAlignment="Center" />
          </StackPanel>
      </Grid>
</Page>
```

The device solution uses the Azure Devices Client as well as the Message Model NuGet packages.

- Right-click on the device solution in Solution Explorer and select Manage NuGet packages…
- With the source set to NuGet.Org
  - search for Microsoft Azure Devices Client (be sure to select the Browse ‘tab’ at the top of the NuGet window), and install that package.
  - search for Newtonsoft JSON and install that package.
- With the source set to the d2c2d NuGets folder location, add a reference to the MessageModelsNet5 NuGet package

Open the MainPage.xaml.cs file and add the following using statements:

```
using Looksfamiliar.d2c2d.MessageModels;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
using System.Text;
using System.Net.Http;
using System.Threading.Tasks;
```

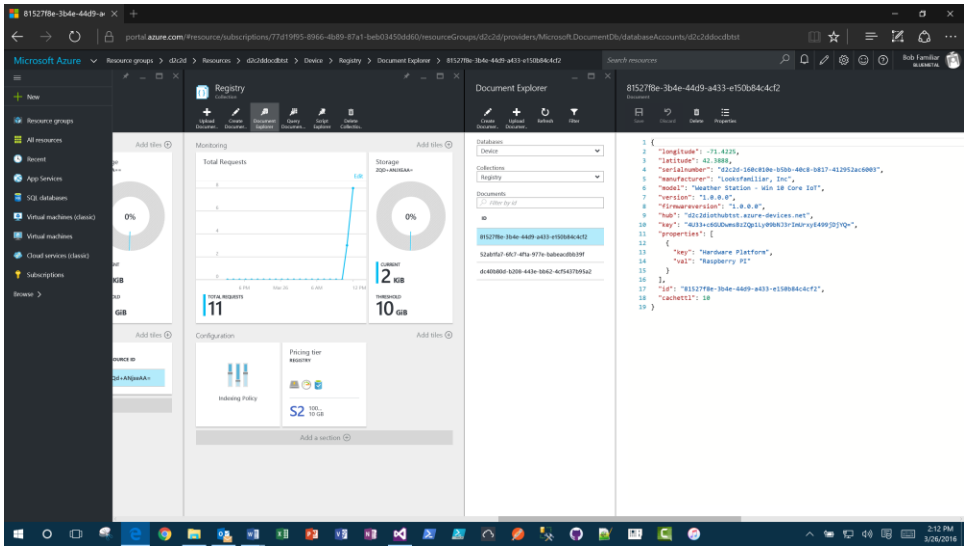
The device will need to know how to connect to the Provision API in order to retrieve its manifest.

- Add the following class member variables (within the MainPage class)

```
private const string DeviceSerialNumber = "[device-serial-number]";
private const string ProvisionApi = "[provision-api]" + DeviceSerialNumber;
private const string AckMessage = "Windows 10 Core IoT Device is Alive";

private static DeviceManifest _deviceManifest;
private static DeviceClient _deviceClient;

private static Task _pingTask;
private static Task _listenTask;
private static Task _telemetryTask;
private static bool _sendingTelemetry = false;
```

To retrieve the Device Serial Number for the lab, use the Azure Portal to navigate to DocumentDb and lookup the device you provisioned in Lab 2

- Click on DocumentDb in your Resource Group
- From the menu bar, click Document Explorer to open a new blade
- Select the *Device* Database and the *Registry* Collection from the drop downs
- Click on the document in the list to open the JSON document



- Copy the 'serialnumber' property and paste that into your source code as the `[device-serial-number]` initialization value; be sure to remove the surrounding brackets!
- Replace `[provision-api]` with the Provision microservice endpoint:

  `https://<prefix>provisiontapi<suffix>.azurewebsites.net/provision/devicemanifests/id/`

To invoke a ReST API from a Windows 10 Core IoT Device, you will use the HttpClient class.

- Add the method called GetDeviceManifest() to the MainPage class to invoke the Provision API and retrieve the device

```
private static async Task<DeviceManifest> GetDeviceManifest()
{
    var client = new HttpClient();
    var uriBuilder = new UriBuilder(ProvisionApi);
    var json = await client.GetStringAsync(uriBuilder.Uri);
    return JsonConvert.DeserializeObject<DeviceManifest>(json);
}
```

- Add a MainPage_OnLoaded() script that will call the GetDeviceManifest() routine and then use the DeviceClient to connect to IoT Hub.

```
private async void MainPage_OnLoaded(object sender, RoutedEventArgs e)
{
    Status.Text = "Main Page Loaded";

    try
    {
        _deviceManifest = await GetDeviceManifest();

        _deviceClient = DeviceClient.Create(
            _deviceManifest.hub,
            AuthenticationMethodFactory.CreateAuthenticationWithRegistrySymmetricKey(
                _deviceManifest.serialnumber, _deviceManifest.key),
                TransportType.Http1);

        Status.Text = "IoT Hub Connected";
    }
    catch (Exception connectionErr)
    {
```

```
            Status.Text = connectionErr.Message;
        }

        StartPingTask(Status);
    }
```

Each activity that we want our device to provide is implemented as a background task. This way the device can be doing multiple tasks at the same time such as issuing heartbeat messages, sending telemetry, and receiving commands.

- Add the following implementation for the StartPingTask() method. This routine kicks off a background thread which is a forever loop that sends Ping messages to IoT Hub every 10 seconds.

```
private static void StartPingTask(TextBox status)
{
    _pingTask = Task.Factory.StartNew(async () =>
    {
        while (true)
        {
            var ping = new Ping
            {
                Ack = AckMessage,
                Longitude = _deviceManifest.longitude,
                Latitude = _deviceManifest.latitude,
                DeviceId = _deviceManifest.serialnumber
            };

            var json = JsonConvert.SerializeObject(ping);

            var message = new Message(Encoding.ASCII.GetBytes(json));

            try
            {
                await _deviceClient.SendEventAsync(message);
            }
            catch (Exception err)
            {
                var errMessage = err.Message;
                status.Text = errMessage;
            }

            await Task.Delay(10000);
        }
    });
}
```

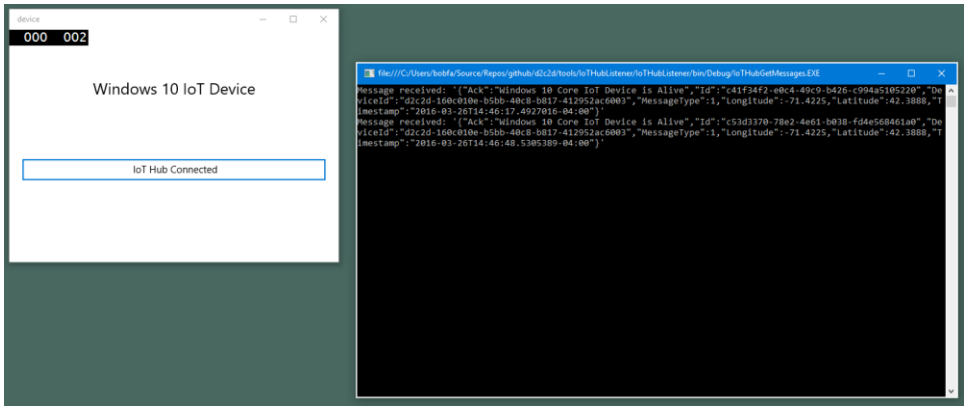
- Compile and run your code on the local machine; you should see a simple window open, with the heading “Windows 10 IoT Device.” When the device is connected to the IoT Hub in Azure, you should see a message populate the blue box.

- In a new instance of Visual Studio, navigate to the Tools/IoTHubListener/runtime folder
  - Open the IoTHubGetMessages.exe.config file and update the *iothubconnstr* app setting with connection string for IoT Hub. This is the same value you copied from the Azure Portal into the Dashboard application’s app.config file in the previous lab. It has a format like:

    HostName=<*prefix*>iothub<*suffix*>.azure-devices.net;SharedAccessKeyName=io…

- Run IoT Hub Listener application by double-clicking IoTHubGetMessages.exe. This utility will show you the messages arriving at IoT Hub from the device application you just built.

  Recall, the interval is set via the last line of the MainPage class to 10 seconds.



2 Provision a Stream Analytics Job

The messages coming from the device are first sent to IoT Hub. There they are held for a minimum of one day and a maximum of seven days. In order for the messages to get to our Dashboard, the messages need to be forwarded from IoT Hub to some type of additional processing and/or storage location.

Stream Analytics is the service in Azure that defines scalable jobs to read messages from IoT Hub, apply business rules and/or transformations, and route payloads to storage locations for application integration. The storage locations that Stream Analytics supports include DocumentDb, SQL Database, Service Bus Queues and Topics, Event Hubs, PowerBI, Blob and Table storage.

A Stream Analytics Job has one or more inputs and outputs and a query. Your first Stream Analytics Job will be used to route all incoming messages to a Service Bus Queue. In a later step, you'll add a more sophisticated job that catches alarm states.

The PowerShell script that your will use has all this information already defined:

- The Input for the Stream Analytics Job is IoT Hub
- The Output for the Stream Analytics Job is the *messagedrop* Service Bus Queue
- The query for the job will be:

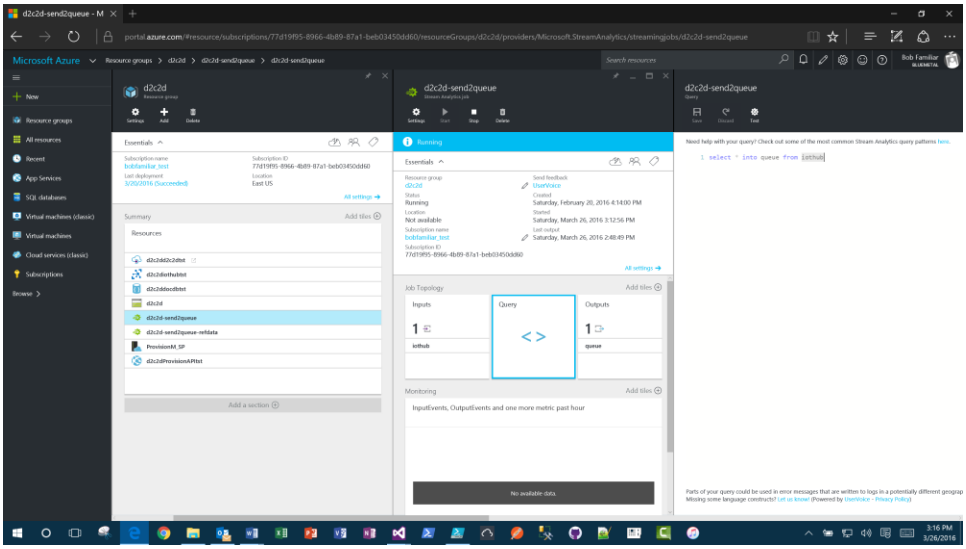
```
select * into queue from iothub
```

Run the 05-Provision-SAJob-1.ps1 script and provide the parameters as prompted:

- .\04-Provision- SAJob-1.ps1
  Subscription: [the name of your subscription]
  ResourceGroup: [the name of your resource group, d2c2d for example]

Azure Location: [East US for example]
Prefix: [a unique prefix to be used in the naming of service components]
Suffix: [dev | tst | stg | prd]

This script will deploy a Stream Analytics job called 'd2c2d-messages-queue'. Validate that the script provisions the Stream Analytics job by navigating to the Azure Portal Resource Groups screen and clicking on that resource.



3 Update the Dashboard to receive Ping Messages

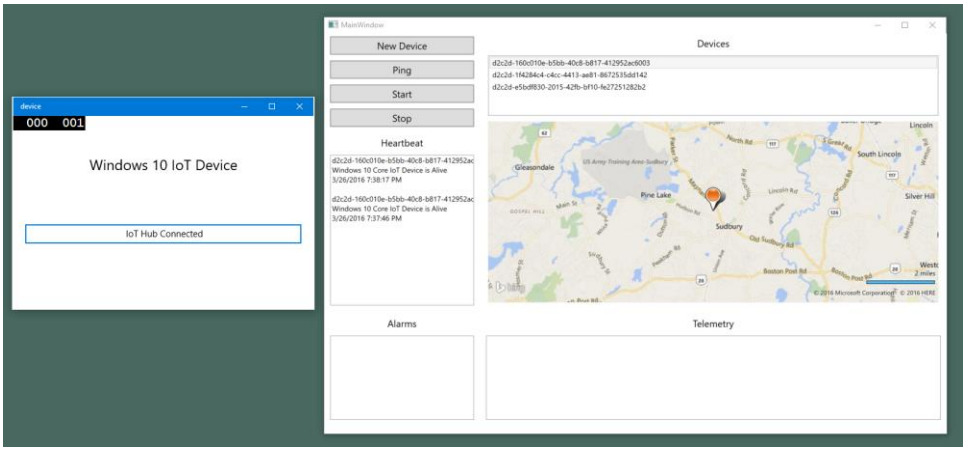
Now that the messages are being delivered to the Service Bus *messagedrop* queue, you can add code to the Dashboard to receive those messages and display them on the UI.

- Open the Dashboard Solution in Visual Studio
- Add the following code to the end of the MainWindow_OnLoaded() event to kick off a background thread that polls for Ping Messages arriving on the *messagedrop* queue:
-

```
var messageTask = Task.Factory.StartNew(() =>
{
    while (true)
    {
        var message = _messageClient.Receive();
        var messageBody = string.Empty;
        if (message == null) continue;

        try
        {
            messageBody = message.GetBody<string>();

            var obj = JsonConvert.DeserializeObject<MessageBase>(messageBody);

            switch (obj.MessageType)
            {
                case MessageTypeEnum.NotSet:

                    throw new Exception("Message Type Not Set");
```

```
                        break;

                    case MessageTypeEnum.Ping:

                        var ping = JsonConvert.DeserializeObject<Ping>(messageBody);

                        Application.Current.Dispatcher.Invoke(
                               DispatcherPriority.Background,
                            new ThreadStart(delegate
                            {
                                // update the map
                                var location = new Map.Location(
                                    ping.Latitude, ping.Longitude);
                                var pin = new Map.Pushpin {Location = location};

                                MyMap.Children.Add(pin);
                                MyMap.Center = location;
                                MyMap.ZoomLevel = 12;
                                MyMap.SetView(location, 12);
                                MyMap.Focusable = true;
                                MyMap.Focus();

                                // update the Ping message display
                                var currHeartbeat = PingFeed.Text;
                                PingFeed.Text = string.Empty;
                                PingFeed.Text += $"{ping.DeviceId}\r\n";
                                PingFeed.Text += $"{ping.Ack}\r\n";
                                PingFeed.Text += $"{ping.Timestamp}\r\n\r\n";
                                PingFeed.Text += $"{currHeartbeat}";
                            }));

                        break;
                }
                 message.Complete();
            }
            catch (Exception err)
            {
                Application.Current.Dispatcher.Invoke(
                   DispatcherPriority.Background, new ThreadStart(delegate
                {
                    var currTelemetry = TelemetryFeed.Text;
                    TelemetryFeed.Text = string.Empty;
                    TelemetryFeed.Text += $"{err.Message}\r\n";
                    TelemetryFeed.Text += $"{messageBody}\r\n\r\n";
                    TelemetryFeed.Text += $"{currTelemetry}\r\n\r\n";
                    message.Abandon();
                }));
            }
        }
    });
```

- Test the end-to-end message delivery
    - Start the Device Application
    - Start the Dashboard Application



4 Congratulations! You have competed Lab 3

Let's review:

- You created the Device Solution and added the code to initialize the device using the Provision API, connect to IoT Hub, and send Ping messages
- You provisioned a Stream Analytics Job that takes incoming messages and places them on a Service Bus Queue
- You updated the Dashboard to receive the Ping messages