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

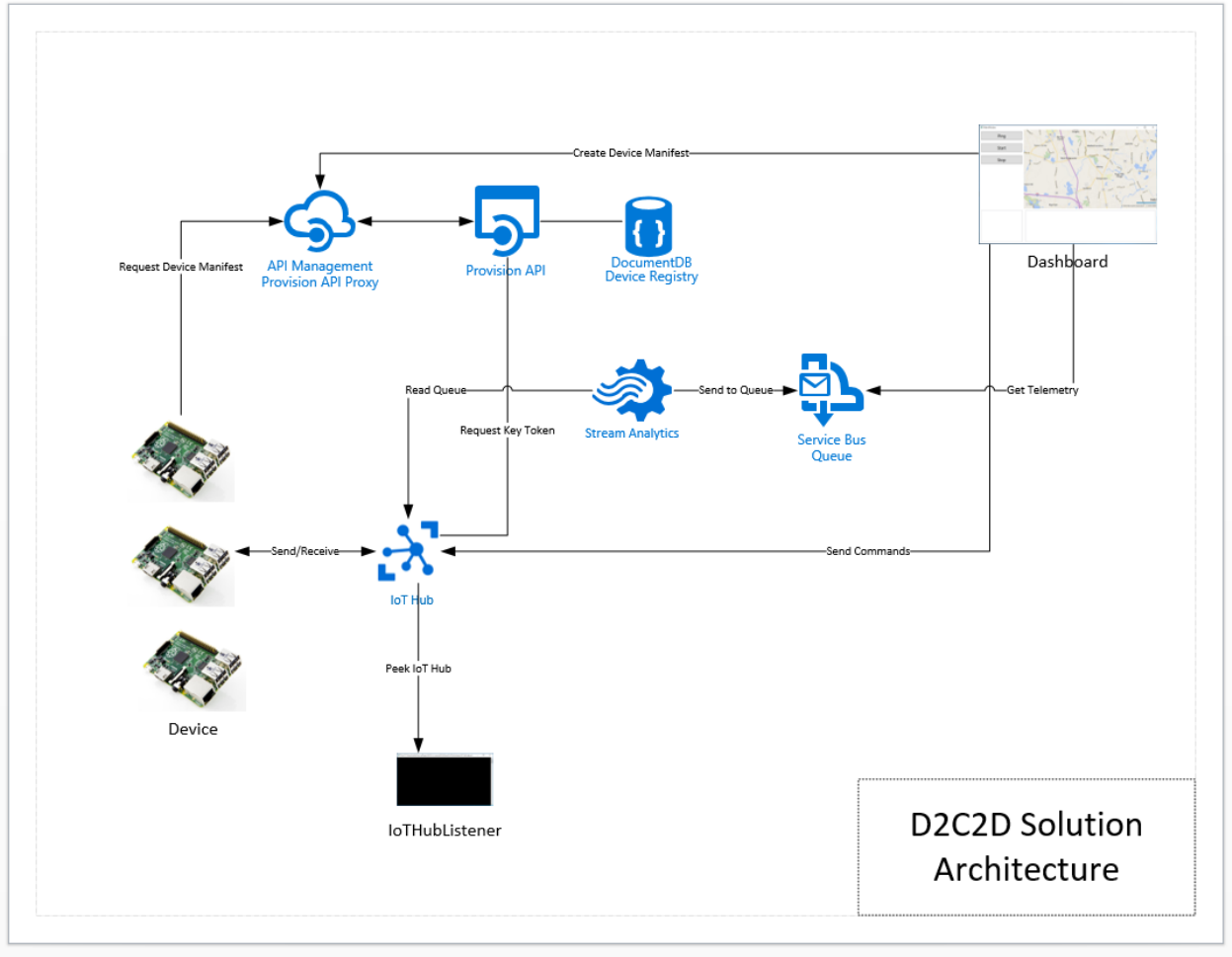
Workshop Overview

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 an Azure environment using PowerShell 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 develop a Windows 10 Core IoT application that sends telemetry and receives incoming commands as well as develop a real-time dashboard that displays incoming telemetry and has the ability to send commands to the remote device. Device Provisioning, IoT Hub monitoring and techniques for dynamic business rules will be covered.

.



Solution Architecture

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

- **Device** – a Windows 10 IoT Core IoT solution that dynamically connects to IoT hub providing heartbeat and climate telemetry and processes several incoming commands. The device application will run on yoru local system or can be deployed to a Windows 10 Core IoT device
- **Dashboard** – a Windows 10 WPF application that displays registered devices, map location using Bing Maps, incoming device telemetry and alarms
- **Provision API** – ReST API the provides end points for device registration with IoT Hub and DocumentDb and device manifest lookup via unique serial number. The Dashboard application registered devise 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

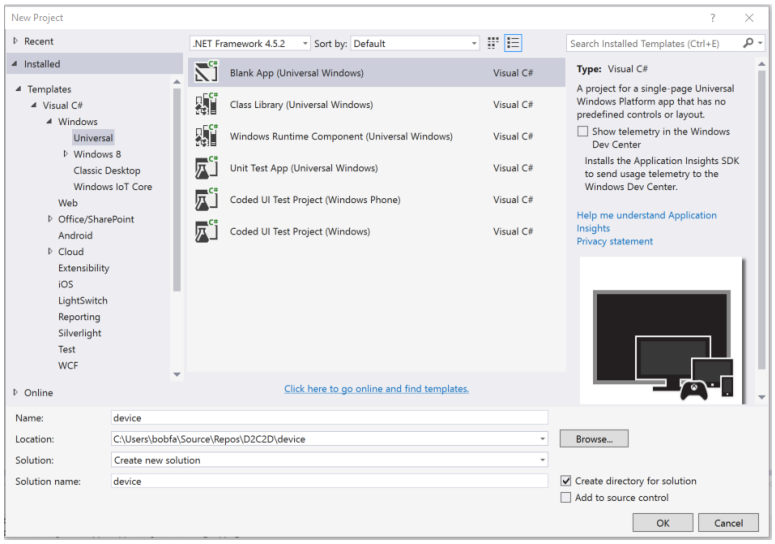
- **API Management** – provides proxy, policy injection and developer registration services for ReST APIs

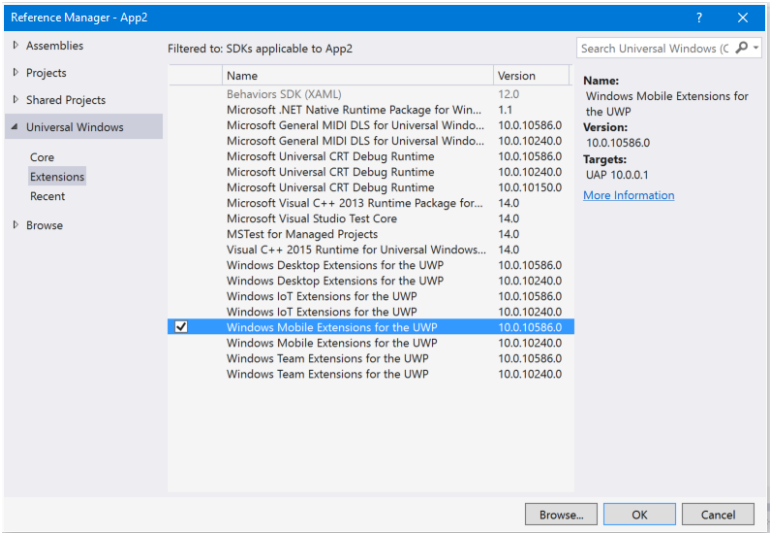
- **Service Bus Namespace** – two queues are defined, one that is a target for all incoming messages, the other will have receive messages that contain data that is an alarm state, an out of range value
- **IoT Hub –** IoT Hub provides device registration, incoming telemetry at scale and cloud to device message services
- **DocumentDb** – DocumentDb is a NoSQL database service that is used for manage Device Manifests, i.e. a Device Registry
- **Stream Analytics Job –** two solution uses two Stream Analytics jobs, one that handles all incoming messages routing them to one queue and the other identifies alarm states and routs those messages to another 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** 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  - Add a Reference to the Windows IoT Extensions for UWP 10.0.10586.0



- Add the following Grid and Control layout to the MainPage.xaml file by using the d2c2d-DeviceMainWindowXaml snippet to inject the Xaml below

```
<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>
```

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

- From within Visual Studio, open the Device solution.
- Open up the NuGet Manager Dashboard in Visual Studio
- With the source set to NuGet.Org, search for Microsoft Azure Devices Client. Add a reference to the Microsoft.Azure.Devices.Client NuGet 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 the d2c2d-DeviceUsings snippet

```
using Looksfamiliar.d2c2d.MessageModels;
using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
```

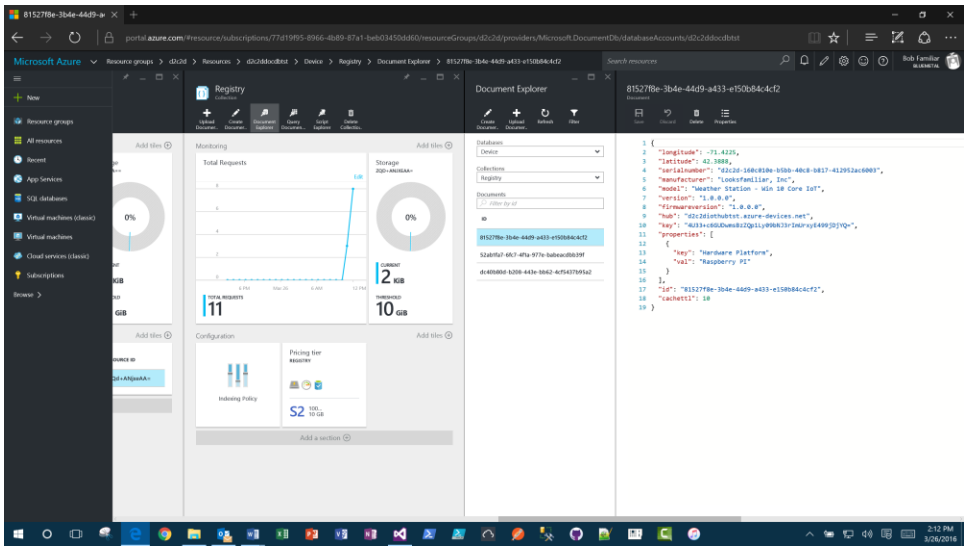
The device will need to know how to connect to the Provision API in order to retrieve its manifest.

- Add the following class member variables with the d2c2d-DeviceMembers snippet

```
private const string DeviceSerialNumber = "[device-serial-number]";
private const string ProvisionApi = "[provision-api]" + DeviceSerialNumber;
private const string SubscriptionKey = "subscription-key=[developer-key]";
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
- Click on the Device Database
- Click on the Registry Collection
- Click Document Explorer in the menu bar



- Copy the ‘serialnumber’ property and paste that into your source code as the DeviceSerialNumber initialization value.
- Add the managed Provision Microservice end point and developer key

In order 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 manifest with the d2c2d-DeviceGetDeviceManifest snippet

```
private static async Task<DeviceManifest> GetDeviceManifest()
```

```
{
    var client = new HttpClient();
    var uriBuilder = new UriBuilder(ProvisionApi) { Query = SubscriptionKey };
    var json = await client.GetStringAsync(uriBuilder.Uri);
    return JsonConvert.DeserializeObject<DeviceManifest>(json);
}
```

- Replace the MainPage_OnLoaded() routine using the d2c2d-DeviceMainPageOnLoaded snippet. This code 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";

    _deviceManifest = await GetDeviceManifest();

    try
    {
        _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);
    StartListenTask(Status);
}
```

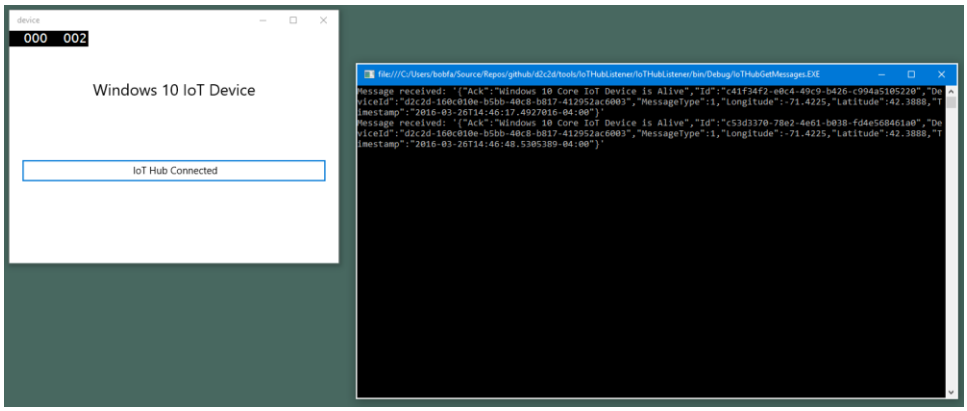
Each activity that we want our device to provide is implemented as a background task. This way the device can be doing multiple tasks at the same time such as sending heartbeat messages, sending telemetry and receiving commands.

- Add an implementation for the StartPingTask() method with the d2c2d-DeviceStartPingTask snippet. This routine kicks off a background thread which is a forever loop that setups Ping Messages and sends them to IoT Hub every 30 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(30000);
            }
        });
    }
```

- Compile and test that your code runs on your local machine
- Navigate to the Tools folder and open up the IoT Hub Listener solution
- Open the App.Config file and update the *iothubconnstr* app setting with connection string for IoT Hub.
- Start the application - This utility will show you the incoming messages arriving in IoT Hub



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 1-day and a maximum of 7 days. In order for the messages to get to our Dashboard, we need to grab the messages out of IoT Hub and send them to some kind of storage location.

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

A Stream Analytics Job is defined as having an input, an output and a query. Our first Stream Analytics Job will be used to catch all incoming messages and route to a Service Bus Queue. In a later step we will add a more sophisticated job that will catch alarm states.

The script that we will use has all this information re-defined so let's review:

- The Input for the Stream Analytics Job will be IoT Hub

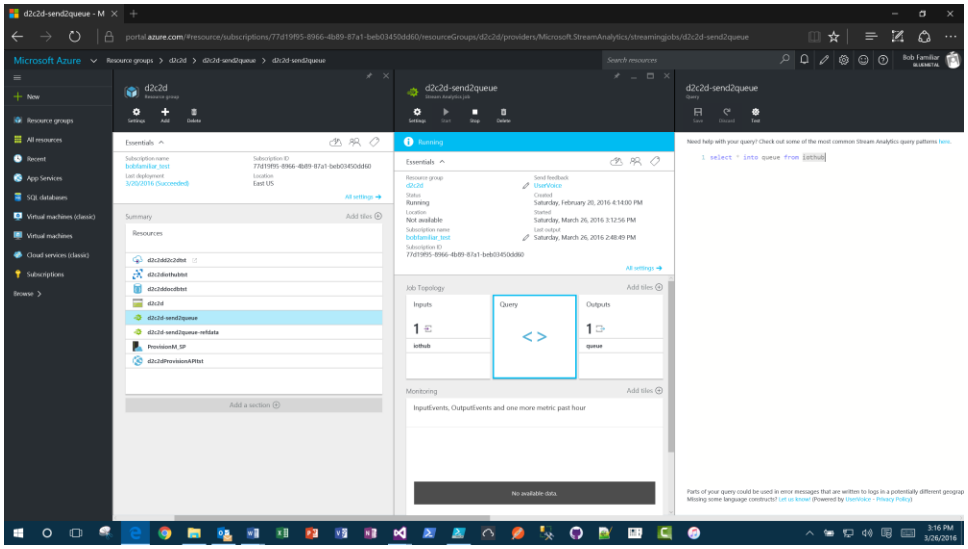
- The Output for the Stream Analytics Job will be the *messagedrop* Service Bus Queue
- The query for the job will be:

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

- Run PowerShell Console 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

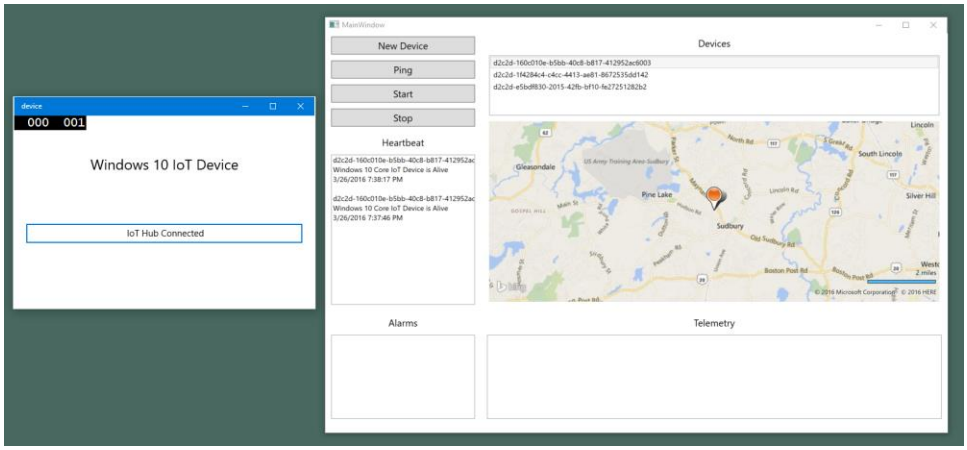
Now that the messages are being delivered to the Service Bus messagedrop queue, we can add code to the Dashboard to receive those messages and display them on the UI.

- Open the Dashboard Solution in Visual Studio
- Update the MainWIndows_OnLoaded() event to kick off a background thread that will wait for Ping Messages to arrive on the messagedrop queue using the d2c2d-DashboardMessageTask snippet.

```
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 Location(
                                        ping.Latitude, ping.Longitude);
                                    var pin = new 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