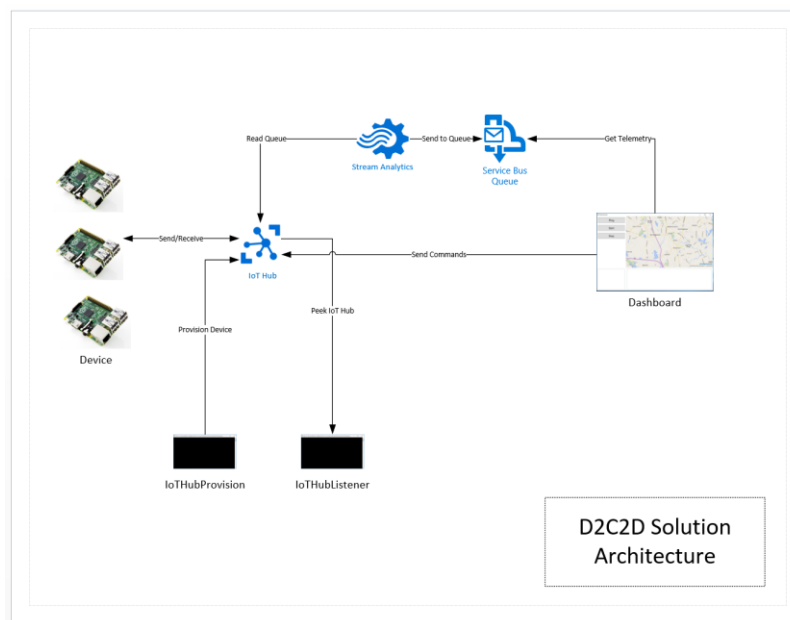


# D2C2D Lab Workbook

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

## Overview

This workshop provides foundational knowledge in how to architect and implement an IoT solution using Windows 10 Core IoT on the device and Azure IoT Hub and Stream Analytics in the cloud. Both Device to Cloud and Cloud to Device patterns are designed and implemented using best practices. At the conclusion of going through this lab you will have provisioned an Azure environment using PowerShell that contains IoT Hub, a Stream Analytics Job, a Service Bus Namespace and Queue and implemented the code for a Device and a Dashboard. In addition, there are a set of command line utilities that you will update with connection string information for provisioning and de-provisioning devices and peeking at the event stream in IoT Hub.



Solution Architecture

This lab will work whether you have a Windows 10 IoT device or not as the device code can run locally on your machine. The dashboard runs locally as well. Both applications use the Azure IoT Stack to communicate.

## Message Model

One of the most fundamental patterns in a complete end-to-end IoT solution is a common message model shared by both the device and the cloud services that process the message coming from the device. The D3C2D solution demonstrates this pattern providing a set of common data that is passed in

each message, defined as a base class, and then a set of child classes that define the telemetry and commands used for two-way communication.

## MessageBase

The Message Base class is a base class for all messages. Each message will carry its type, defined by an enum, its device id, GPS location and a timestamp.

```
public enum MessageTypeEnum
{
    NotSet = 0,
    Ping = 1,
    Climate = 2,
    Command = 3
}

public class MessageBase
{
    public MessageBase()
    {
        Id = Guid.NewGuid().ToString();
        DeviceId = string.Empty;
        MessageType = MessageTypeEnum.NotSet;
        Longitude = 0.0;
        Latitude = 0.0;
        Timestamp = DateTime.Now;
    }

    public string Id { get; set; }
    public string DeviceId { get; set; }
    public MessageTypeEnum MessageType { get; set; }
    public double Longitude { get; set; }
    public double Latitude { get; set; }
    public DateTime Timestamp { get; set; }
}
```

## Ping

The Ping class is used as a heartbeat message, informing the cloud environment that the device, regardless of whether it is currently sending telemetry, is active and able to communicate.

```
public class Ping : MessageBase
{
    public Ping()
    {
        MessageType = MessageTypeEnum.Ping;
        Ack = string.Empty;
    }

    public string Ack { get; set; }
}
```

## Command

The Command Class is used to send commands to the device. The class contains an enum that defines the type of command. The class also carries an optional JSON formatted string of command parameters. See ClimateSettings as an example of a class that represents parameters.

```
public enum CommandTypeEnum
{
    Ping = 0,
    Start = 1,
    Stop = 2,
    UpdateFirmware = 3
}

public class Command : MessageBase
```

```

{
    public Command()
    {
        CommandType = CommandTypeEnum.Ping;
        CommandParameters = string.Empty;
        MessageType = MessageTypeEnum.Command;
    }

    public CommandTypeEnum CommandType { get; set; }
    public string CommandParameters { get; set; }
}

```

## Climate

Climate is an example of a telemetry message. This message provides temperature and humidity values.

```

public class Climate : MessageBase
{
    public Climate()
    {
        Humidity = 0;
        Temperature = 0;
        MessageType = MessageTypeEnum.Climate;
    }

    public double Humidity { get; set; }
    public double Temperature { get; set; }
}

```

## ClimateSettings

The ClimateSettings class is used to provide input parameters to the device for minimum and maximum temperature and humidity values. Providing input parameters allows end users the ability to calibrate devices remotely.

```

public class ClimateSettings : MessageBase
{
    public ClimateSettings()
    {
        MinHumidity = 0;
        MaxHumidity = 0;
        MinTemperature = 0;
        MaxTemperature = 0;
    }

    public double MinHumidity { get; set; }
    public double MaxHumidity { get; set; }
    public double MinTemperature { get; set; }
    public double MaxTemperature { get; set; }
}

```

## Location

The Location class is used by the device in order to capture the return message from the <http://ip-api.com/json> API.

```

public class Location
{
    public string @as { get; set; }
    public string city { get; set; }
    public string country { get; set; }
    public string countryCode { get; set; }
    public string isp { get; set; }
    public double lat { get; set; }
    public double lon { get; set; }
    public string org { get; set; }
    public string query { get; set; }
    public string region { get; set; }
    public string regionName { get; set; }
    public string status { get; set; }
    public string timezone { get; set; }
    public string zip { get; set; }
}

```

## Lab

### Step Details

#### 1 Requirements

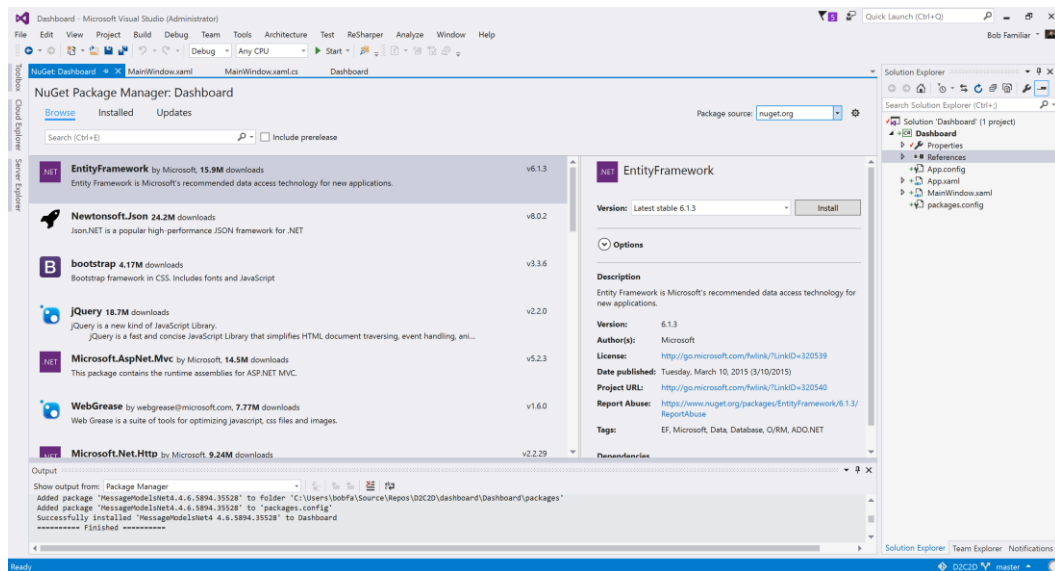
- Azure Account - <https://azure.microsoft.com/en-us/>
- Visual Studio 2015 - <https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx>
- Azure SDK 2.8 and Azure PowerShell 1.2.1 - <https://azure.microsoft.com/en-us/downloads/>
- Windows 10 Core IoT Templates - <https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ad-82a5-a88ac4b01dec>

#### 2 Download the lab materials from Git Hub

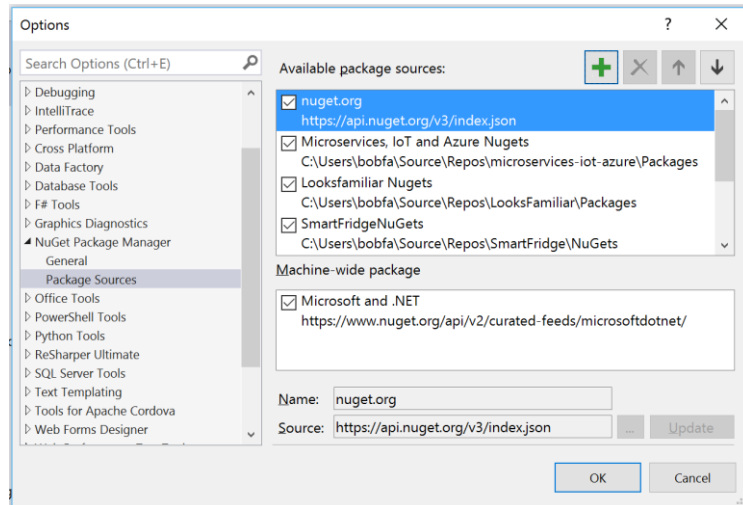
- <https://github.com/bobfamiliar/d2c2d>

Add the local *NuGets* folder to your Visual Studio environment

- Follow this menu path in Visual Studio: Tools Menu → NuGet Package Manager → Manage NuGet Packages for this solution. The NuGet Package Manager Dashboard will display.



- Click the Options button in the upper right corner of the manager window to bring up the options dialog



- To add the NuGet Packages location for this repo, click the + icon to add an additional package location
- Change the name to something meaningful (D2C2D Packages, for example).
- Use the ... button to navigate to the *NuGets* folder at the top level of the repo.
- Select the folder, click Update, and then OK. Now you can switch between the online NuGet catalogs and this local NuGet catalog when making NuGet package references. Referencing shared NuGet packages is now fully integrated into your development environment.

### 3 Provision Service bus

- Run PowerShell console as Administrator
- Navigate to the *Automation* folder of the D2C2D repo
- Login to Azure using 'Classic' mode using your Azure credentials

#### ➤ Add-AzureAccount

- Run the 01-Provision-ServiceBus.ps1 script. Enter the parameters as you are prompted:

#### ➤ .\01-Provision-ServiceBus

**Subscription:** [the name of your subscription]

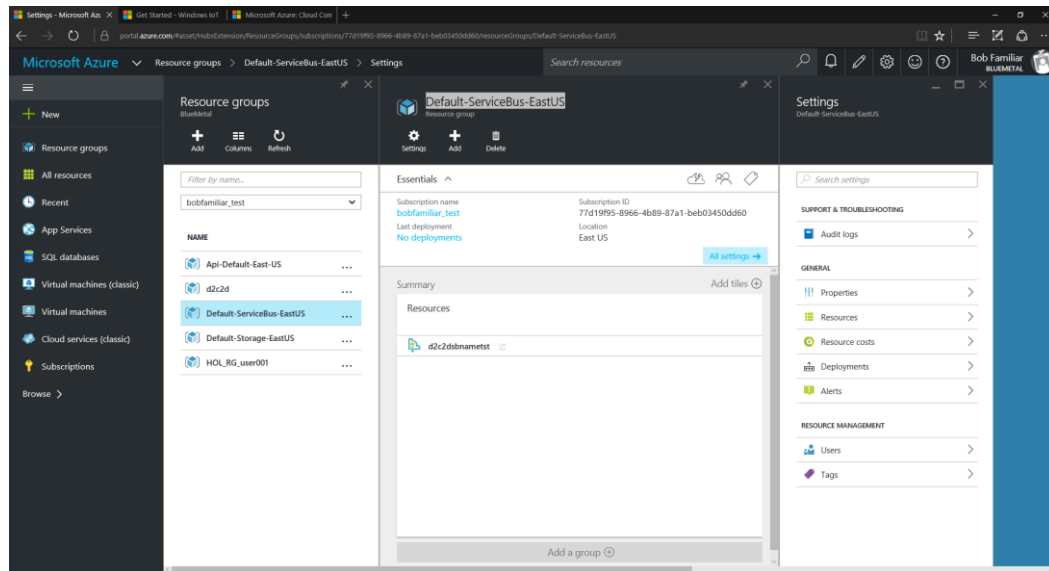
**Azure Location:** [East US for example]

**Prefix:** [a unique prefix to be used in the naming of service components]

**Suffix:** [dev | tst | stg | prd]

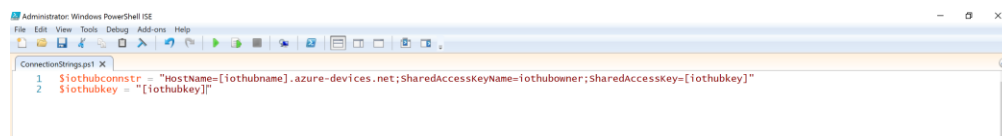
The script will create a Service Bus Namespace called [Prefix]sbname[Suffix]. This naming format is used to help achieve a unique name.

Validate that the service bus was created by viewing the Resource Groups in the Azure Portal. Look for a resource group called 'Default-ServiceBus-EastUS' or similar based on the region you selected. Click through to see that your service bus was created.



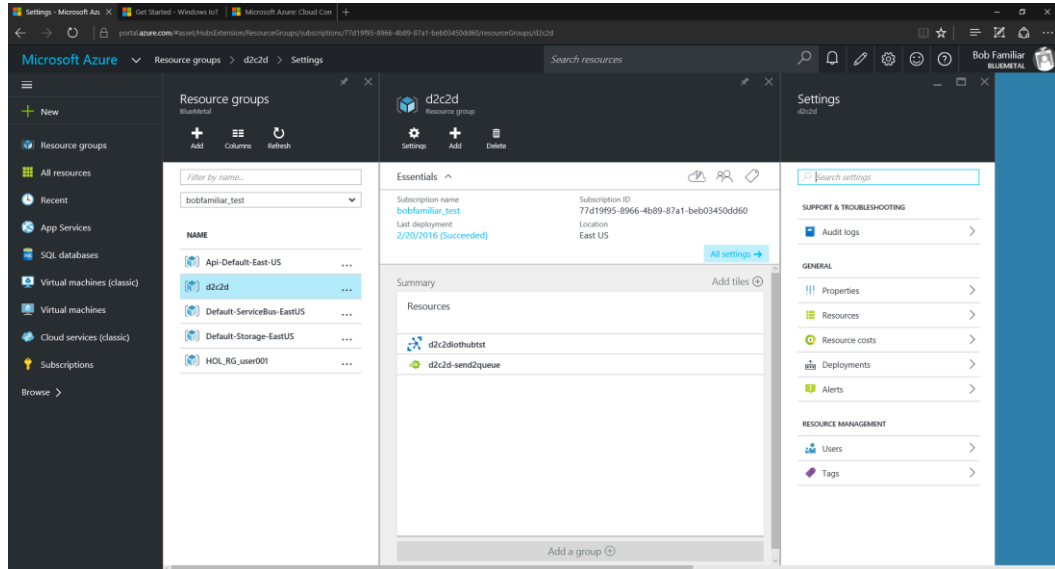
#### 4 Provision IoT Hub and Stream Analytics Job

- Login to Azure using the 'ARM' mode using your Azure credentials
  - Login-AzureRMAccount
- Run the 02-Provision-IoTHub.ps1 script. Enter the parameters as you are prompted
  - .\02-Provision-IoTHub.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]
- Note the name of the IoT Hub and the Primary Key that are output on the screen. Copy those two values and update the ConnectionStrings.ps1 file in the Automation folder.



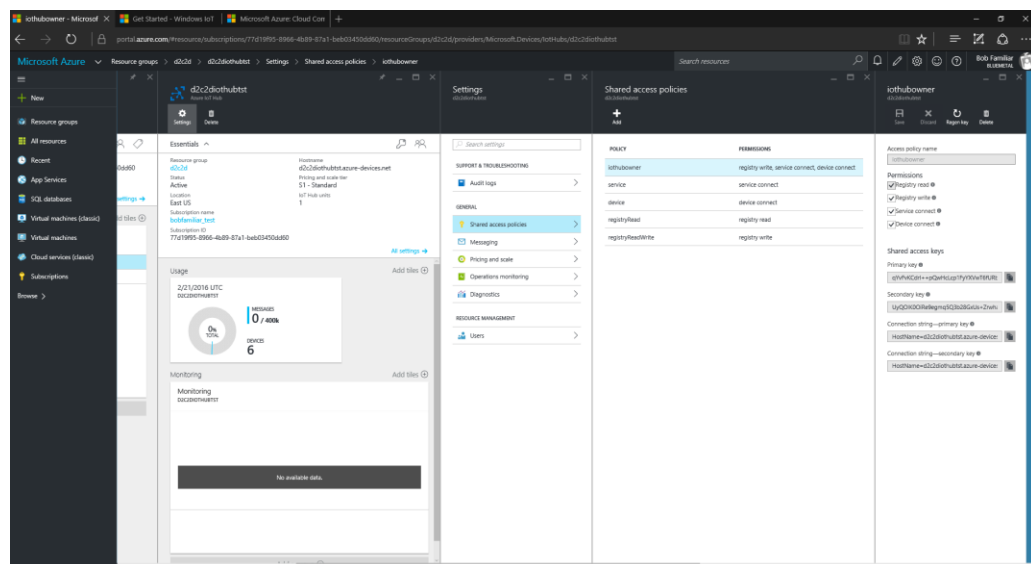
- Run the 03-Provision-SAJon.ps1 Script and provide the parameters as prompted:
  - .\03-Provision-IoTHub.ps1
    - Subscription:** [the name of your subscription]
    - ResourceGroup:** [the name of your resource group, d2c2d for example]
    - Azure Location:** [East US for example]
    - Prefix:** [a unique prefix to be used in the naming of service components]
    - Suffix:** [dev | tst | stg | prd]

- Validate that the script provision the IoT Hub and Stream Analytics job by navigating to the Azure Portal Resource Groups screen. Click on the Resource Group that you created:

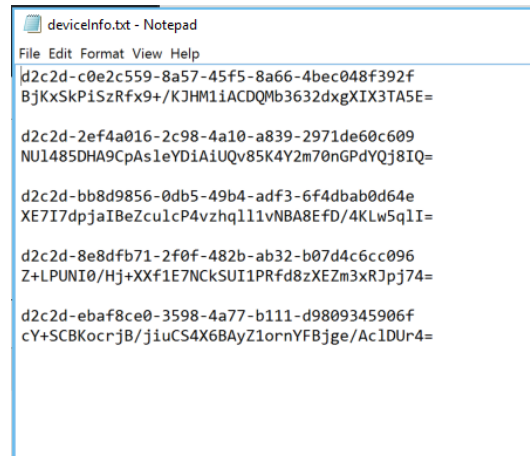


## 5 Provision a Device

- Each device that connects to IoT Hub requires a unique Id and IoT Hub needs to know that unique Id. This application will generate a unique id for your device and register it with IoT Hub.
- Start Visual Studio and open the solution tools\IoTHubProvision
- Update the App.Config setting for the setting called **IoTHubConnStr**. You can get the IoT Hub connection string from the Azure Portal. Use the connection associated with the *iothubowner* policy.



- Run the application. It will connect to your IoT Hub and provision a device with a unique id. If you run the app multiple times it will register multiple devices.
- Open the DeviceInfo.txt file in the tools\IoT Hub Provision\IoT Hub Provision\bin\Debug folder. It will contain the generated Id for your device along a unique device key. These values will be used by the Device Client to securely connect to IoT Hub.



```

deviceInfo.txt - Notepad
File Edit Format View Help
d2c2d-c0e2c559-8a57-45f5-8a66-4bec048f392f
BjKxSkPiSzRfx9+/KJHMIACDQMb3632dxgXIX3TA5E=

d2c2d-2ef4a016-2c98-4a10-a839-2971de60c609
NU1485DHA9CpAs1eYDiAiUQv85K4Y2m70nGPdYQj8IQ=

d2c2d-bb8d9856-0db5-49b4-adf3-6f4dbab0d64e
XE7I7dpjaIBeZcu1cP4vzhq111vNBA8EFd/4KLw5qlI=

d2c2d-8e8dfb71-2f0f-482b-ab32-b07d4c6cc096
Z+LPUNI0/Hj+XXf1E7NCKSUI1PRfd8zXEZm3xRj74=

d2c2d-ebaf8ce0-3598-4a77-b111-d9809345906f
cY+SCBKocrjB/jiuCS4X6BAyZ1ornYFBjge/Ac1DUr4=

```

## 6 Build the Message Model

- The message model is a common class library used by both the device and the applications to consume and send messages back and forth via IoT Hub.
- To support both .NET 4.6 (dashboard) and .NET 5 (device), two versions of the class library have been defined. The build script will build both versions of the library and drop NuGet packages into the top level *nugets* folder.
- Using the PowerShell console navigate to the models\automation folder and run the build script

➤ .\build-models

**Repo:** [path to the repo]

**Configuration:** [debug | release]

## 7 Create a Windows 10 Universal App for your Windows 10 Core IoT device

- Open Visual Studio and load the Device solution. This solution is based on a Windows 10 Core IoT project template. It consists of a single Xaml/CSharp page. The goal of this solution is to provide some minimal status on the UI while the app kicks off several background threads that both send and receive messages.
- Open the Main.Xaml.cs file and add the IoT Hub Name, Device Id and Device Key values to initialize the variables.
- Add a reference to the Microsoft.Azure.Devices.Client NuGet package using the NuGet Manager Dashboard
- Add a reference to the Message Model for .NET 5
- Add variables for location, device client and our background tasks

```

private static Location _location;
private static DeviceClient _deviceClient;

private static Task _pingTask;
private static Task _listenTask;

```



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

- Add a routine that dynamically looks up location.

```
private static async Task<Location> GetLocationAsync()
{
    var client = new HttpClient();
    var json = await client.GetStringAsync("http://ip-api.com/json");
    var location = JsonConvert.DeserializeObject<Location>(json);
    return location;
}
```

- Call this routine from MainPage\_OnLoaded() routine

```
Status.Text = "Main Page Loaded";

_location = await GetLocationAsync();

Status.Text = $"Location is {_location.city}, {_location.country}";
```

- Add the code that connects to IoT Hub

```
try
{
    _deviceClient = DeviceClient.Create(IotHubUri,
        AuthenticationMethodFactory.CreateAuthenticationWithRegistrySymmetricKey(
            DeviceId, DeviceKey),
        TransportType.Http1);

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

- Add the Ping Task. This task will send a Ping message once a minute. For testing, you can decrease the number of ticks to delay between messages to increase the send rate.

```
private static void StartPingTask(TextBox status)
{
    _pingTask = Task.Factory.StartNew(async () =>
    {
        while (true)
        {
            var ping = new Ping
            {
                Ack = "Windows 10 Core IoT Device is Alive",
                Longitude = _location.lon,
                Latitude = _location.lat,
                DeviceId = DeviceId
            };

            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(60000);
        }
    });
}
```

}

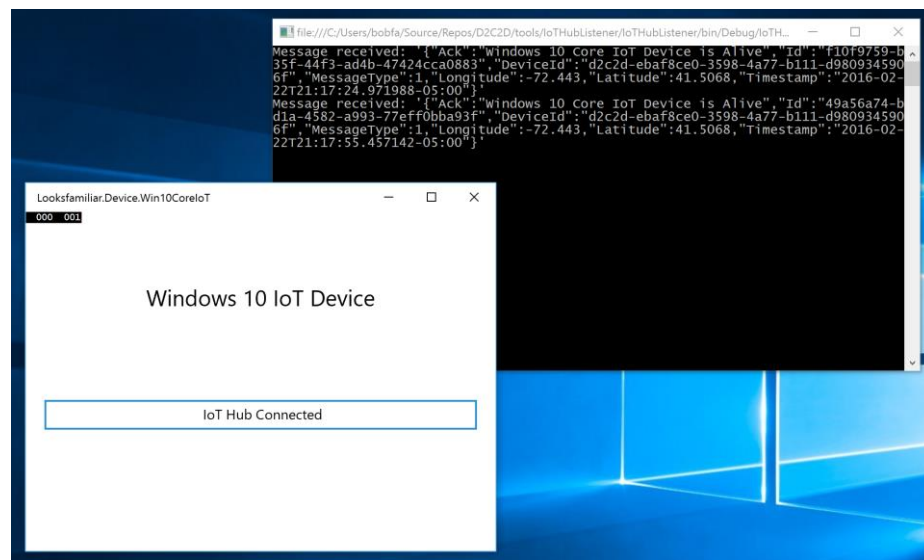
- Add the code that will start the Ping Task at the end of the MainPage\_OnLoaded()

```
StartPingTask(Status);
```

- Build and run the Application



- Open another instance of Visual Studio and open the IoT Hub Listener project from the tools folder. The IoT Hub Listener allows you to see the messages as they arrive at IoT Hub.
- Open the App.Config file and update the IoT Hub Connection String
- Compile and run the application
- Are you seeing Ping messages? If the answer is yes, you may advance to the next step.



8 Create a Windows 10 Universal App that sends commands and receives telemetry from your Windows 10 IoT Device

- Go to the Bin Maps Portal, sign in and request a developer key - <https://www.bingmapsportal.com>
- Install the Bing Map WPF Control - <https://www.microsoft.com/en-us/download/details.aspx?id=27165>
- Open the Dashboard solution and then the App.Config file and update the settings for Bing Map Key
- Add a reference to the Bing Map WPF Control DLL - C:\Program Files (x86)\Bing Maps WPF Control\V1\Libraries
- Uncomment the reference to the WPF Control in the MainWindow.Xaml file
- Add references to the Microsoft.Azure.Devices and Microsoft.ServiceBus NuGet packages
- Add a reference to the Message Model for .NET 4.6
- Open the MainWindow.Xaml.cs file and add variables for Service Bus Queue Client and IoT Hub Service Client

```
private readonly QueueClient _messageClient;  
private ServiceClient _serviceClient;
```

- Add the code to initialize these variables in the MainWindows() constructor

```
_messageClient = QueueClient.CreateFromConnectionString(  
    ConfigurationManager.AppSettings["ServiceBusConnStr"], "messagingdrop");  
_serviceClient = ServiceClient.CreateFromConnectionString(  
    ConfigurationManager.AppSettings["IoTHubConnStr"], TransportType.Amqp);
```

- In the MainWindow\_OnLoaded() routine, add the code that kicks off a background thread that reads messages from the Service Bus Queue, updates the Bing Map and outputs the message to the screen.

```
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  
                        {  
                            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();  
                        }  
                    );  
                }  
            }  
        }  
    }  
});
```

```

        PingFeed.Text += $"{ping.DeviceId}\r\n";
        PingFeed.Text += $"{ping.Ack}\r\n";
        PingFeed.Text += $"{ping.Timestamp}\r\n\r\n";
    }));

    break;
case MessageTypeEnum.Climate:
    var climate = JsonConvert.DeserializeObject<Climate>(messageBody);
    Application.Current.Dispatcher.Invoke(
        DispatcherPriority.Background, new ThreadStart(delegate
        {
            TelemetryFeed.Text +=
                $"Timestamp {climate.Timestamp.ToLongDateString()}" +
                " {climate.Timestamp.ToLongTimeString()}\r\n";
            TelemetryFeed.Text += $"Temperature {climate.Temperature}\r\n";
            TelemetryFeed.Text += $"Humidity {climate.Humidity}\r\n\r\n";
        }));
    break;
case MessageTypeEnum.Command:
    // noop
    break;
default:
    throw new ArgumentOutOfRangeException();
}

message.Complete();
}
catch (Exception err)
{
    Application.Current.Dispatcher.Invoke(
        DispatcherPriority.Background, new ThreadStart(delegate
        {
            // Indicates a problem, unlock message in queue.
            TelemetryFeed.Text += err.Message;
            TelemetryFeed.Text += messageBody;
            message.Abandon();
        }));
}
}
});

```

- Add the click event handlers for the Start and Stop buttons

```

private void StartButton_Click(object sender, RoutedEventArgs e)
{
    var climateSettings = new ClimateSettings
    {
        MinHumidity = 0,
        MaxHumidity = 100,
        MinTemperature = 75,
        MaxTemperature = 110
    };

    var command = new Command
    {
        CommandType = CommandTypeEnum.Start,
        CommandParameters = JsonConvert.SerializeObject(climateSettings),
        DeviceId = ConfigurationManager.AppSettings["DeviceId"]
    };

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

    _serviceClient.SendAsync(ConfigurationManager.AppSettings["DeviceId"], message);
}

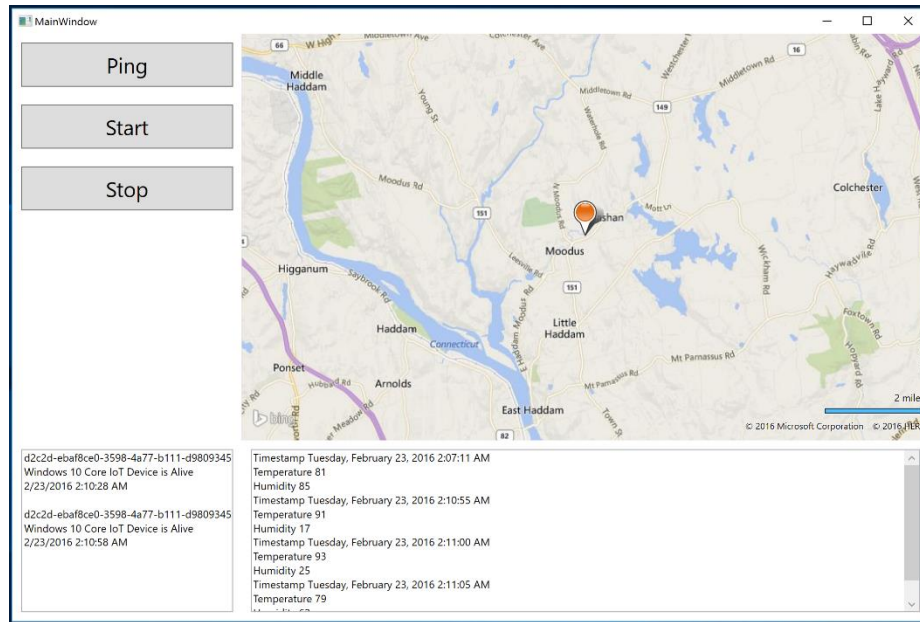
private void StopButton_Click(object sender, RoutedEventArgs e)
{
    var command = new Command()
    {
        CommandType = CommandTypeEnum.Stop,
        DeviceId = ConfigurationManager.AppSettings["DeviceId"]
    };

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

    _serviceClient.SendAsync(ConfigurationManager.AppSettings["DeviceId"], message);
}

```

- Build



## 9 Enhance the Device solution

- Add the Listener Task and the routine that kicks off the Telemetry task

```
private static void StartListenTask(TextBox status)
{
    _listenTask = Task.Factory.StartNew(async () =>
    {
        while (true)
        {
            var message = await _deviceClient.ReceiveAsync();

            if (message == null)
                continue;

            var json = Encoding.ASCII.GetString(message.GetBytes());
            var command = JsonConvert.DeserializeObject<Command>(json);

            switch (command.CommandType)
            {
                case CommandTypeEnum.Ping:
                    var ping = new Ping
                    {
                        Ack = "Windows 10 Core IoT Device is Alive",
                        Longitude = _location.lon,
                        Latitude = _location.lat,
                        DeviceId = DeviceId
                    };

                    json = JsonConvert.SerializeObject(ping);
                    var pingMessage = new Message(Encoding.ASCII.GetBytes(json));

                    try
                    {
                        await _deviceClient.SendEventAsync(pingMessage);
                    }
                    catch (Exception err)
                    {
                        var errorMessage = err.Message;
                        status.Text = errorMessage;
                    }
                }
            }
        }
    });
}
```

```

        break;
    case CommandTypeEnum.Start:
        var settings = JsonConvert.DeserializeObject<ClimateSettings>(
            command.CommandParameters);
        _sendingTelemetry = true;
        StartTelemetry(settings, status);
        break;
    case CommandTypeEnum.Stop:
        _sendingTelemetry = false;
        break;
    case CommandTypeEnum.UpdateFirmware:
        break;
    default:
        throw new ArgumentOutOfRangeException();
    }

    await _deviceClient.CompleteAsync(message);
});
}

private static void StartTelemetry(ClimateSettings settings, TextBox status)
{
    _telemetryTask = Task.Factory.StartNew(async () =>
    {
        while (_sendingTelemetry)
        {
            var random = new Random();

            var climate = new Climate
            {
                Temperature = random.Next((int) settings.MinTemperature, (int)
                    settings.MaxTemperature),
                Humidity = random.Next((int) settings.MinHumidity, (int) settings.MaxHumidity),
                DeviceId = DeviceId
            };

            var json = JsonConvert.SerializeObject(climate);

            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(5000);
        }
    });
}
}

```

- Add the line that calls the StartListenerTask() routine at the end of MainPage\_OnLoaded() routine

```
StartListenTask(Status);
```

- Build and run both the Device and the Dashboard.
- Test turning telemetry on and off

## 10 Next Steps

- If you don't have one yet, consider purchasing a Windows 10 IoT Device - <https://dev.windows.com/en-us/iot>  
<https://www.toradex.com/windows-iot-starter-kit>

