BLUEMETAL
An Insight company

d2c2d

Lab Workbook One

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

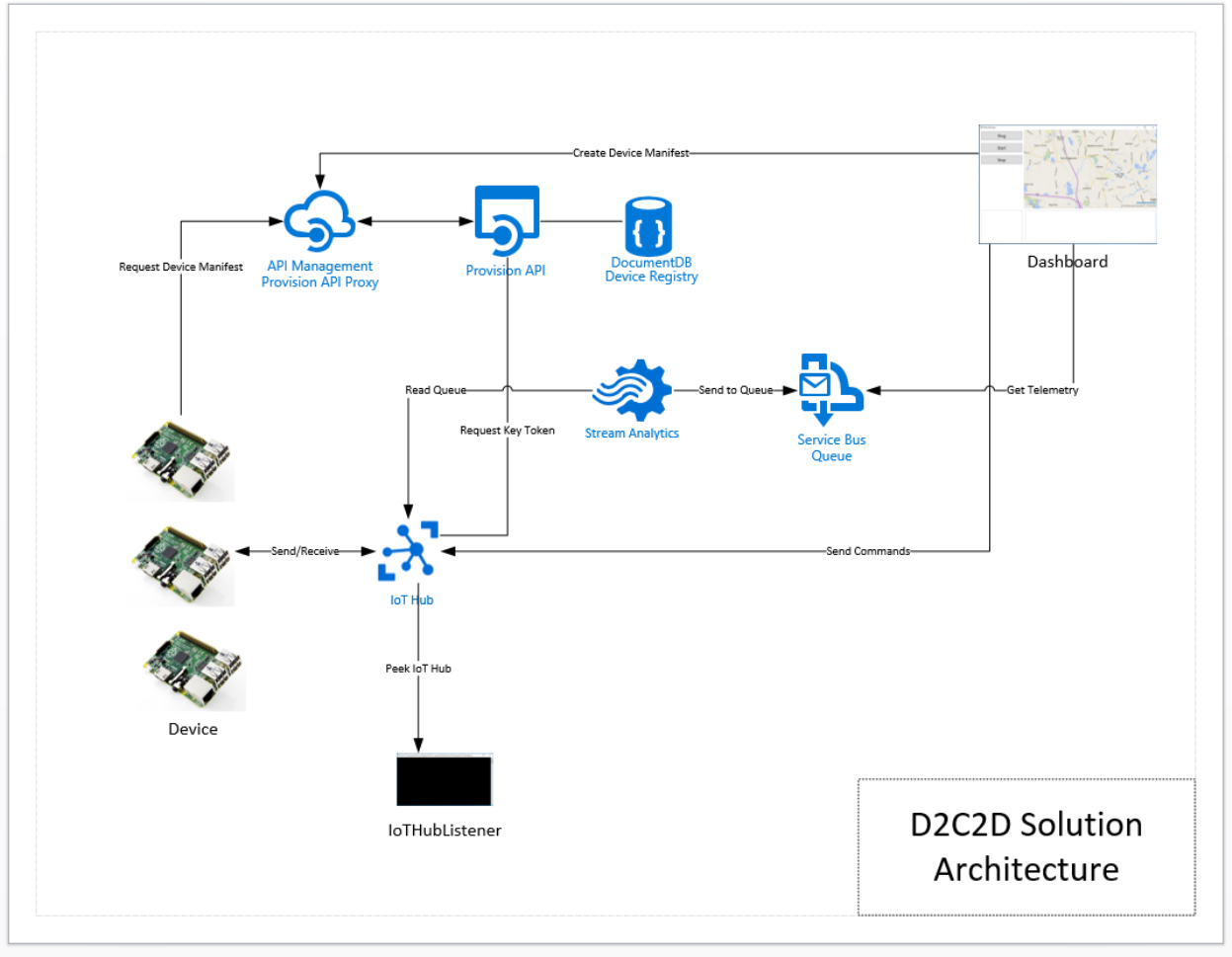
Workshop Overview

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 your local system or can be deployed to a Windows 10 Core IoT device
- **Dashboard** – a Windows 10 WPF application that displays registered devices, maps location using Bing Maps, displays 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

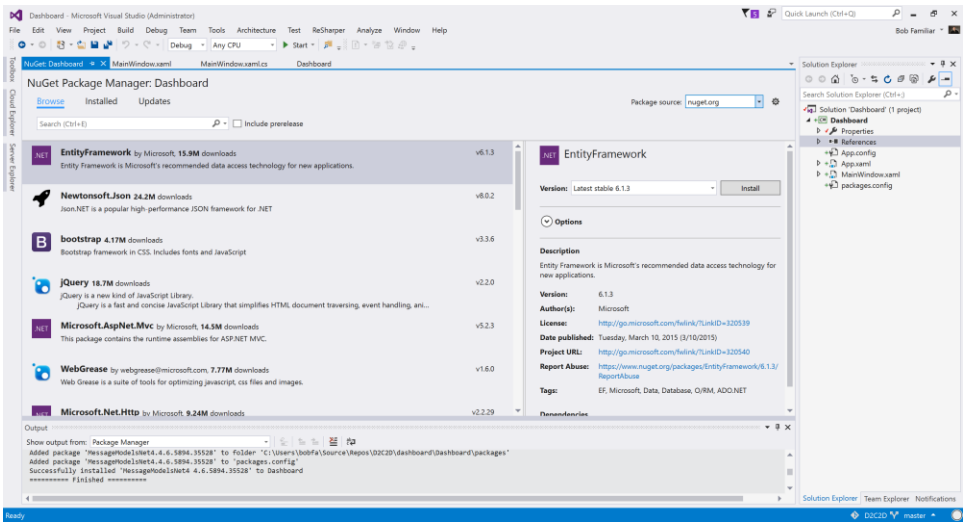
- **Service Bus** – two queues are defined, one that is a target for all incoming messages, the other will have receive messages that contain data that is an alarm state, an out of range value
- **IoT Hub –** IoT Hub provides device registration, incoming telemetry at scale and cloud to device message services
- **DocumentDb** – DocumentDb is a NoSQL database service that is used for manage Device Manifests, i.e. a Device Registry
- **Stream Analytics –** the 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 One Overview

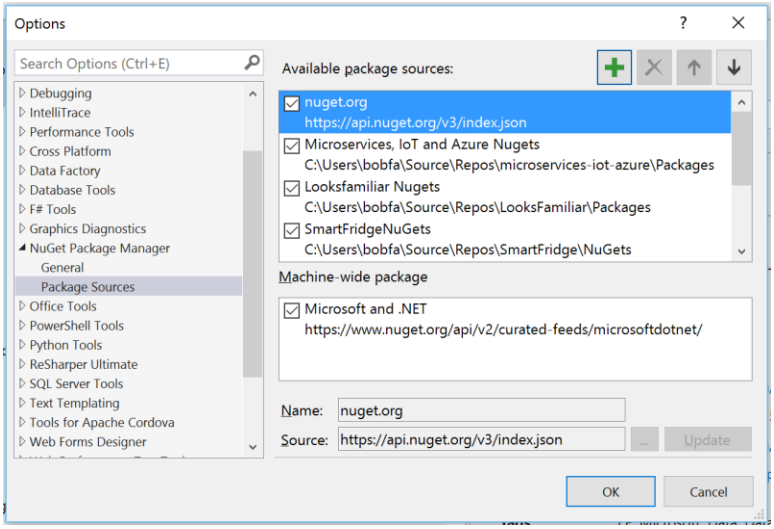
In this lab your will configure your development environment, provision Azure services, complete the development of and build the common Message Model library.

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 - Visual Studio 2015 Update 1 - https://www.visualstudio.com/en-us/news/vs2015-update1-vs.aspx - PowerShell 5 - https://www.microsoft.com/en-us/download/details.aspx?id=50395 - 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 - Go to the Bing Maps Portal, sign in and request a developer key - https://www.bingmapsportal.com - Install the Bing Maps WPF Control - https://www.microsoft.com/en-us/download/details.aspx?id=27165
2	**Download the lab materials from Git Hub** - https://github.com/bobfamiliar/d2c2d
3	**Add the repo's *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 ellipse '…' 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.

4 Configure Your Azure PowerShell environment

- Run PowerShell console as Administrator
- Execute the following commands. Note: You should only have to do this once

- Set-ExecutionPolicy Unrestricted
- Install-Module AzureRM
- Install-AzureRM
- Import-Module AzureRM
- Install-Module Azure
- Import-Module Azure

5 **Provision Service Bus**

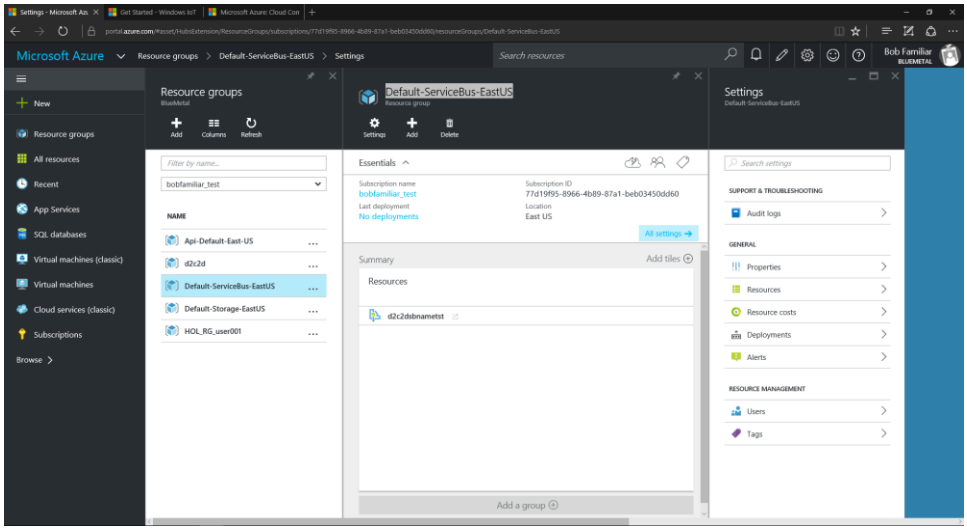
- Navigate to the *Automation* folder of the D2C2D repo
- Login to Azure using 'Classic' mode using your Azure credentials

- 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.

6 Provision IoT Hub

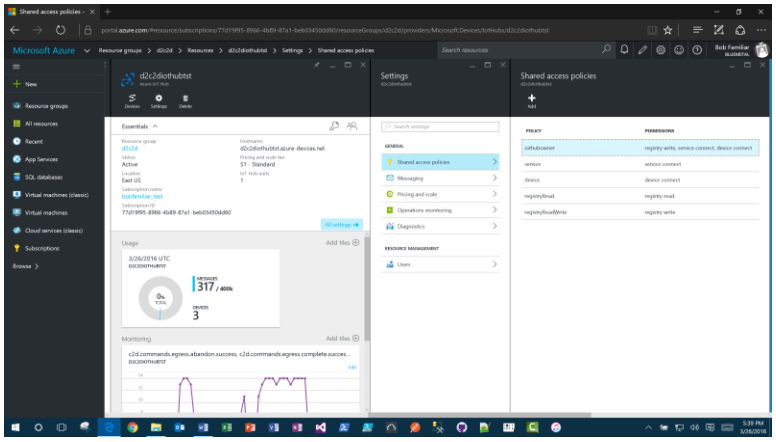
- Login to Azure using the 'ARM' mode using your Azure credentials

➢ 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]

- Validate that the IoT Hub service was provision. Navigate to the Azure Portal and select your resource group to examine the contents.

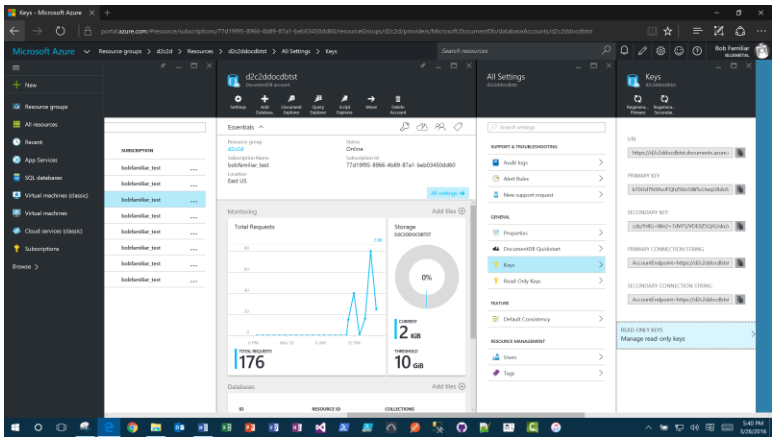


7 Provision DocumentDb

- Run the 04-Provision-DocumentDb.ps1 script and provide the parameters as prompted:

➢ .\04-Provision- DocumentDb.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 create an instance of DocumentDb. Validate that the script provision the DocumentDb has been created by navigating to the Azure Portal Resource Groups screen.

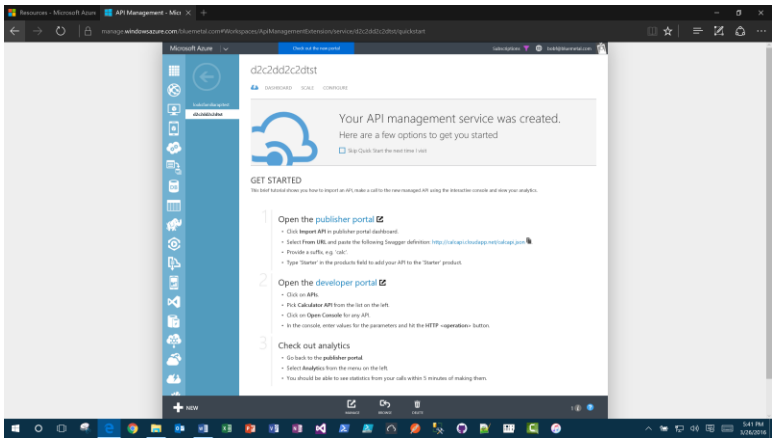


8 **Provision API Management**

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

➢ .\05-Provision- APIManagement.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]
APIAdminEmail: [admin email]

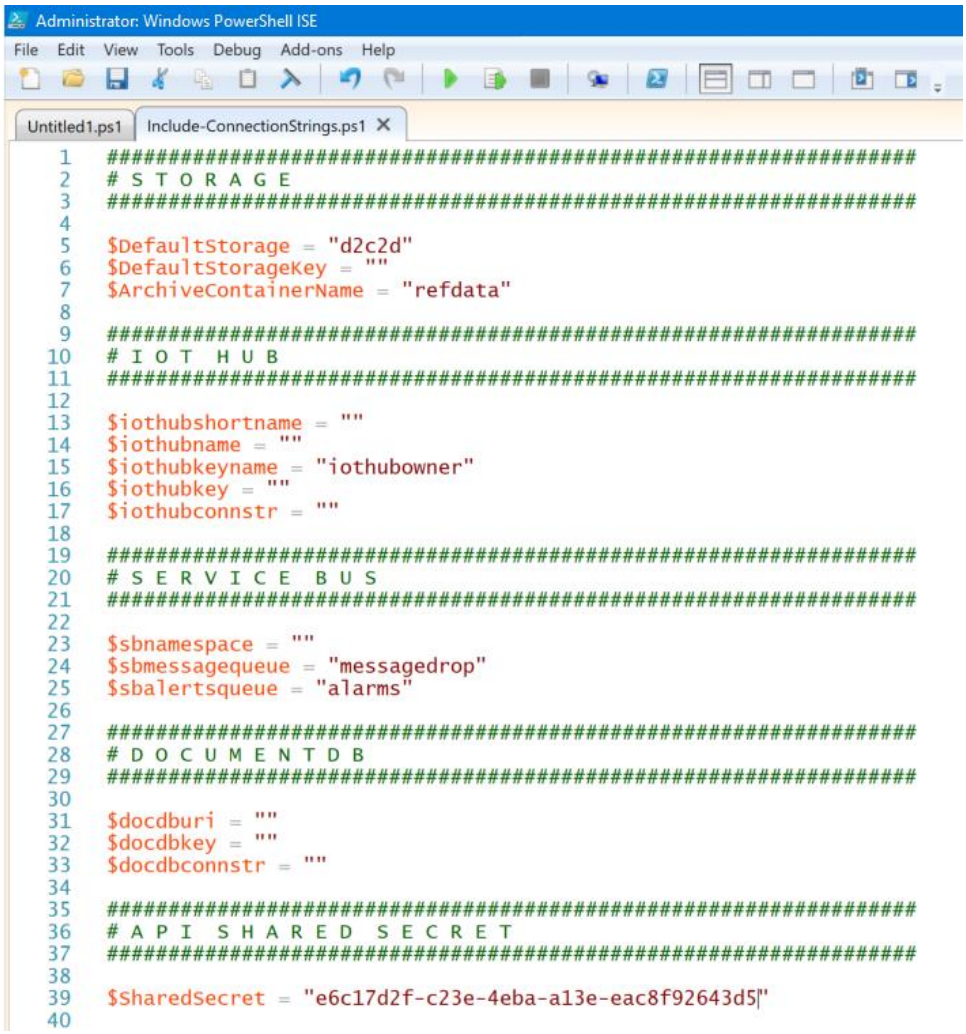
This script will create an instance of APIManagement. Validate that the script provisions APIManagement has been created by navigating to the Azure Portal Resource Groups screen.



9 **Collect Connection Strings**

Several of the services that we have provisioned have connection strings and keys that are needed by our custom code. So that subsequent provisioning, build and deployment scripts can pick up these connection strings, we will collect them into a common file that will be referenced by these other scripts.

- Open the Include-ConnectionStrings.ps1 file using Notepad or PowerShell ISE and collect each of the following connections stings and keys. Each of these values can be retrieved from the Azure Portal
    - Storage
        - Default Storage Key – default storage is called ‘d2c2d’
    - IoT Hub
        - IoT Hub Short Name – the IoT Hub host name
        - IoT Hub Name – full URI for IoT Hub
        - IoT Hub IoT Hub Owner Key – key associated with the IoT Hub Owner profile
        - IoT Hub Connection String – Connection string associated with the IoT Hub owner profile
    - Service Bus
        - Service Bus Namespace – the namespace generated by the script
    - DocumentDb
        - DocumentDb URI
        - DocumentDb Key
        - DocumentDb Connection String



10 **Review and Update 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 D2C2D 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.

The Message Model library defines the following classes:

- MessageBase – base class for all messages
- Command – command class
- Climate – climet telemetry class
- ClimateSettings – command parameters
- Location – location class
- DeviceManifest – device details

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; }
}
```

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,
    UpdateFirmeware = 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;
            MaxHumiditiy = 0;
            MinTemperature = 0;
            MaxTemperature = 0;
        }

        public double MinHumidity { get; set; }
        public double MaxHumiditiy { 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; }
    }
```

DeviceManifest

The DeviceManifest class provides the details about our devices including geo-location, unique serial number, manufacturer, model, version, firmware and a list of custom properties that allow for extensibility of the manifest. In addition, the manifest contains the name of the IoT Hub and the security key for the device to allow for secure communication between the device and IoT Hub.

```
    public class DeviceManifest : ModelBase
    {
        public DeviceManifest()
        {
            longitude = 0.0;
            latitude = 0.0;
            serialnumber = string.Empty;
            manufacturer = string.Empty;
            model = string.Empty;
            version = string.Empty;
```

```
            firmwareversion = string.Empty;
            hub = string.Empty;
            key = string.Empty;
            properties = new DeviceProperties();
        }

        public double longitude { get; set; }
        public double latitude { get; set; }
        public string serialnumber { get; set; }
        public string manufacturer { get; set; }
        public string model { get; set; }
        public string version { get; set; }
        public string firmwareversion { get; set; }
        public string hub { get; set; }
        public string key { get; set; }
        public DeviceProperties properties { get; set; }

        public bool isValid()
        {
            return ((serialnumber != string.Empty) &&
                    (manufacturer != string.Empty) &&
                    (model != string.Empty) &&
                    (version != string.Empty) &&
                    (firmwareversion != string.Empty));
        }
    }
```

In this step of the lab, you will add a class that will be used to provide a heartbeat between the device and cloud. We will call this class *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. It contains a timestamp and a message.

- From within Visual Studio, navigate to the models\Net4 folder and open the MessageModels solution.
- Add a class to the solution and call it Ping
- Modify the contents of the class file with the code below

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

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

The Message Model library provides a common set of classes that represent the messages that are sent and received by both the Device and the Dashboard. The Device solution is built to the .NET 5 Core IoT libraries and the Dashboard is built to .NET 4.6. There are therefore two separate solutions that each build as a distinct NuGet package.

- From within Visual Studio, navigate to the models\Net5 folder and open the MessageModels solution
- Add the existing Ping class from the Net4 version of the library

11 Build the Message Model Libraries

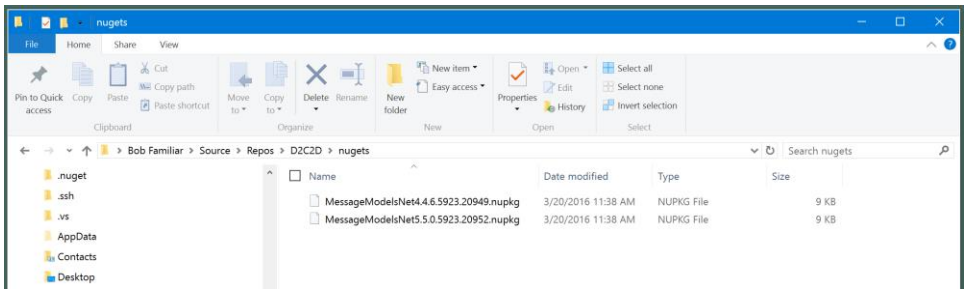
The MessageModel libraries are built as NuGet packages. There is a PowerShell script that

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 PowerShell 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.ps1
Repo: [path to the repo]
Configuration: [debug | release]

Validate that the NuGet packages have been built and dropped into the repo NuGets folder:



12 **Congratulations! You have completed Lab 1.**

Let's review:

- You configured your development environment
- You provisioned a set of Azure Services
- You deployed a Stream Analytics Job
- You updated the MessageModel library to include a Ping message
- You built the MessageModel NuGet libraries