# d2c2d

Lab Workbook One

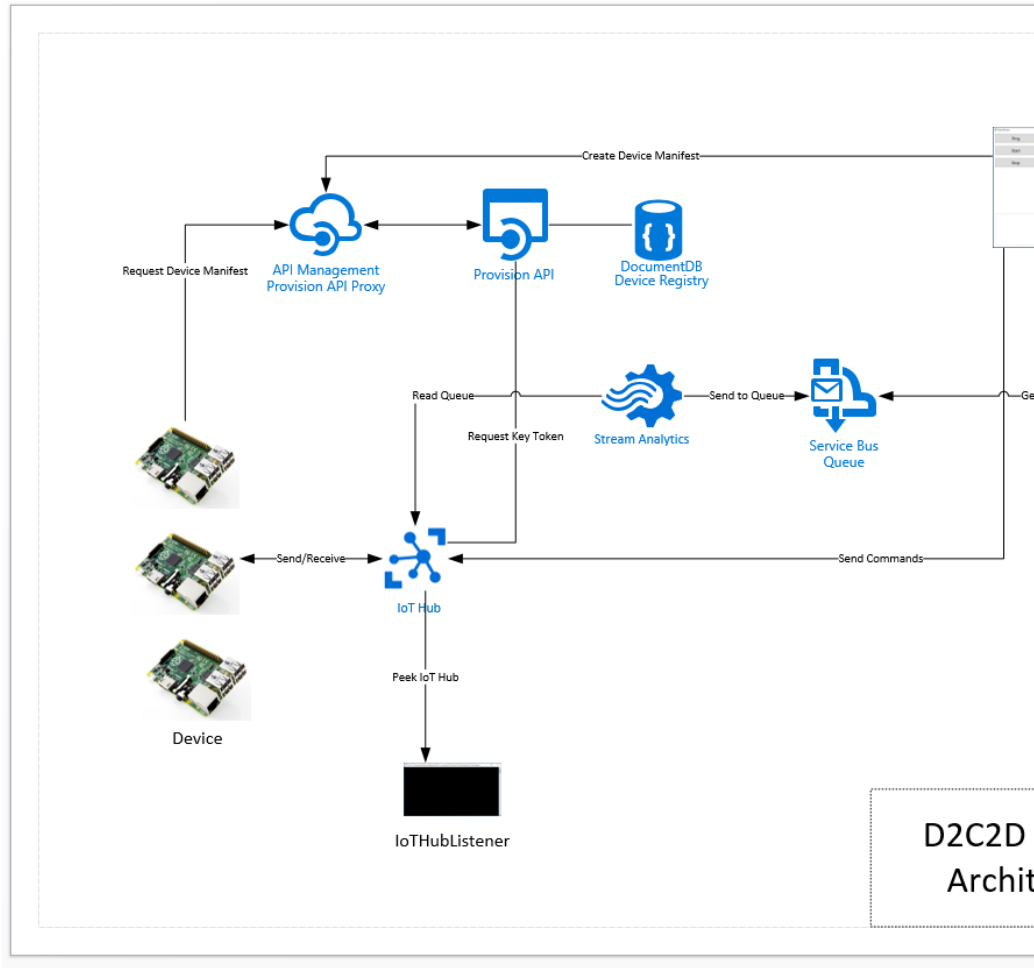
Device to Cloud to Device - a workshop for learning about Windows 10 Cor
development, Azure IoT Hub, Stream Analytics and automating Azure using

## Workshop Overview

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

At the conclusion of this workshop you will have provisioned, using PowerShell, an Azu
that contains IoT Hub, Stream Analytics Jobs that identify telemetry events and alarm s
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 t
receives incoming commands from the cloud as well as a real-time dashboard that can
directionally with the device (e.g., displaying telemetry readings and sending command
device).




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 h heartbeat and climate telemetry as well as responds to command from a dashb application can run on your local machine or be deployed to a Windows 10 Cor as a Raspberry Pi.
- **Dashboard** – a Windows 10 WPF application that lists registered devices, maps Bing Maps, and displays incoming device telemetry and alarms.
- **Provision API** – a ReST API that provides endpoints for device and device manif

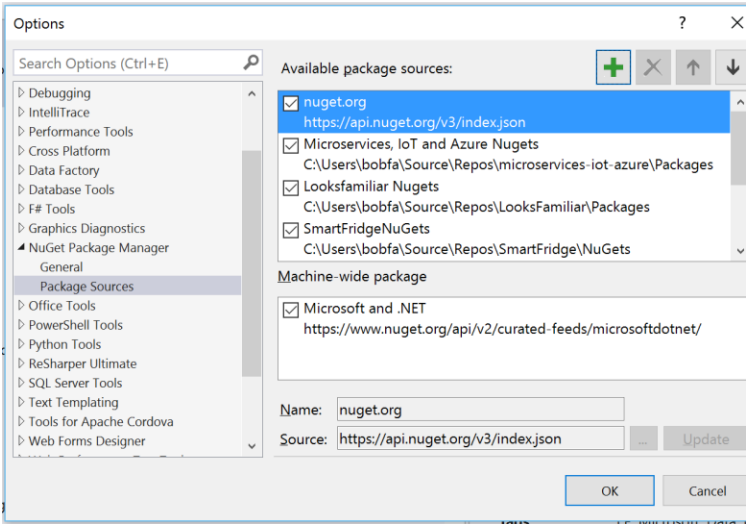
- **IoT Hub –** IoT Hub provides device registration, incoming telemetry at scale, an message services
- **DocumentDb** – DocumentDb is a NoSQL database service that is used for mana Manifests, i.e., a Device Registry
- **Stream Analytics –** the solution leverages two Stream Analytics jobs, one that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

## Lab One Overview

In this lab your will configure your development environment, provision Azure services, Message Model library.

## Lab

| *Step* | *Details* |
|---|---|
| *1* | **Requirements**<br><br>- Azure Account - https://azure.microsoft.com/en-us/<br>- Visual Studio 2015 - https://www.visualstudio.com/en-us/products/v editions.aspx<br>- Visual Studio 2015 Update 2 - https://www.visualstudio.com/news/vs vs<br>- PowerShell 5 - https://www.microsoft.com/en-us/download/details.a<br>- Azure SDK 2.9 (**for Visual Studio 2015**) - https://azure.microsoft.com/ us/blog/announcing-visual-studio-azure-tools-and-sdk-2-9/<br>- Azure PowerShell 1.3.0 – http://aka.ms/webpi-azps<br>- Windows 10 Core IoT Templates - https://visualstudiogallery.msdn.microsoft.com/55b357e1-a533-43ac a88ac4b01dec<br>- Go to the Bing Maps Portal, sign in and request a developer key - https://www.bingmapsportal.com<br>- 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**<br><br>- https://github.com/bobfamiliar/d2c2d<br>- Navigate to the root of the expanded repo, run the PowerShell consol |




- To add the NuGet Packages location for this repo, click the + icon to a package location
- Change the name to something meaningful (D2C2D Packages, for exa
- Use the ellipse '…' button to navigate to the *NuGets* folder at the top repo (that directory is currently empty except for a placeholder.txt fi
- Select the folder, click Update, and then OK. Now you can switch betw NuGet catalogs and this local NuGet catalog when making NuGet pack Referencing shared NuGet packages is now fully integrated into your environment.

## 4 Configure Your Azure PowerShell environment

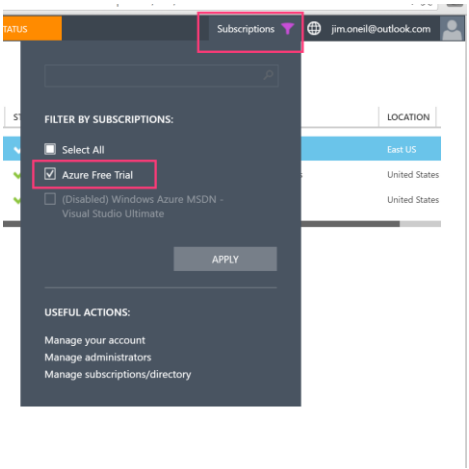
Run PowerShell console as Administrator and execute the following comman should only have to do this once.

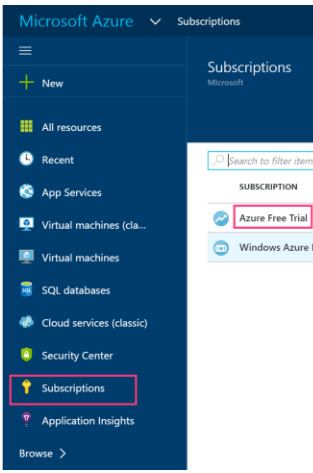
- Set-ExecutionPolicy Unrestricted
- Install-Module AzureRM
- Install-AzureRM
- Install-Module AzureRM -RequiredVersion 1.2.2
- Install-Module Azure
- Import-Module Azure

**Subscription**: [the name of your subscription – ***see images below***]
**ResourceGroup**: [a name for your resource group, d2c2d for example
**Azure Location**: [East US, for example]
**Prefix**: [a prefix to be used in the naming of service components; ***com***
***need to be universally unique, so pick something non-trivial***]
**Suffix**: [dev | tst | stg | prd] – used to differentiate resources used for
development phases

> **Tip:** Make a note of the values provided above; you'll be supplying t
> to additional PowerShell scripts in subsequent labs.

The name of your subscription can be found in the Azure portal ('classic' and

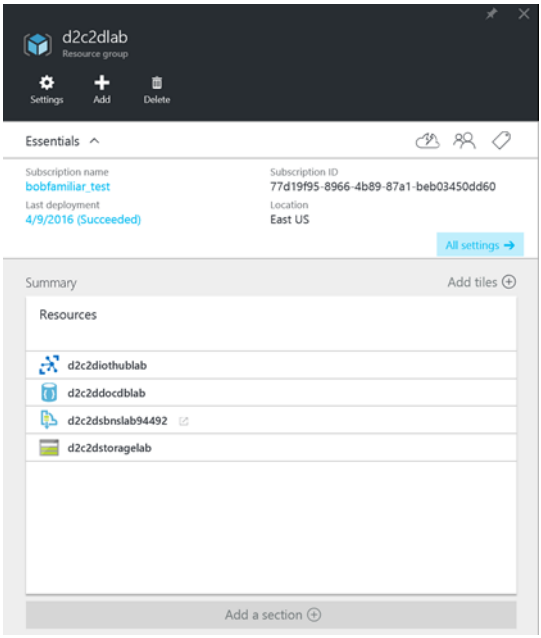



This script creates:

- A Resource Group
- A Service Bus Namespace containing two Queues called *messagedrop*
- An instance of DocumentDb
- An instance of IoT Hub

After running this script, you should see four services within the resource grou
when running the PowerShell script. Note, you'll need to use the new Azure p
of these resources; the classic portal is not equipped to ARM (Azure Resource
managed services.




By the way, the connection strings for these services are written to a JSON file provision-[ResourceGroupName]-output.json, which is then parsed by the EnvironmentVariables.ps1 to load these connections strings for use by the oth will come into play in later labs.

6 **Review the Message Model**

One of the most fundamental patterns in a complete end-to-end IoT solution message model shared by the device and the cloud services that process the r coming from the device.

The D2C2D solution demonstrates this pattern providing a set of common dat in each message, defined as a base class, and then a set of child classes that d telemetry and commands used for two-way communication.

The Message Model library comprises the following classes:

- Climate – climate telemetry class
- ClimateSettings – climate telemetry value bounds (*included in Climate*
- Command – command class
- DeviceManifest – device details

```
        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 contai
defines the type of command. The class also carries an optional JSON formatt
command parameters. See ClimateSettings as an example of a class that repre
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 temper
humidity values.

**ClimateSettings**

The ClimateSettings class is used to provide input parameters to the device fo
maximum temperature and humidity values. Providing input parameters allov
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 messag
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 the devices including geo
unique serial number, manufacturer, model, version, firmware and a list of cu
that allow for extensibility of the manifest. In addition, the manifest contains
IoT Hub and the security key for the device to allow for secure communicatio
device and IoT Hub.

```
public class DeviceManifest : ModelBase
{
    public DeviceManifest()
```

```
        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));
        }
    }
```

**Ping**

Prior to implementing device specific messages, it's a good idea to verify end-
connectivity with a simple heartbeat type message. The Ping class does just th
simple message and timestamp from the device to the cloud.

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

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

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

Let's review:

- You configured your development environment
- You provisioned a set of Azure Services
- You familiarized yourself with the Message Model library which will b
  subsequent labs.