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

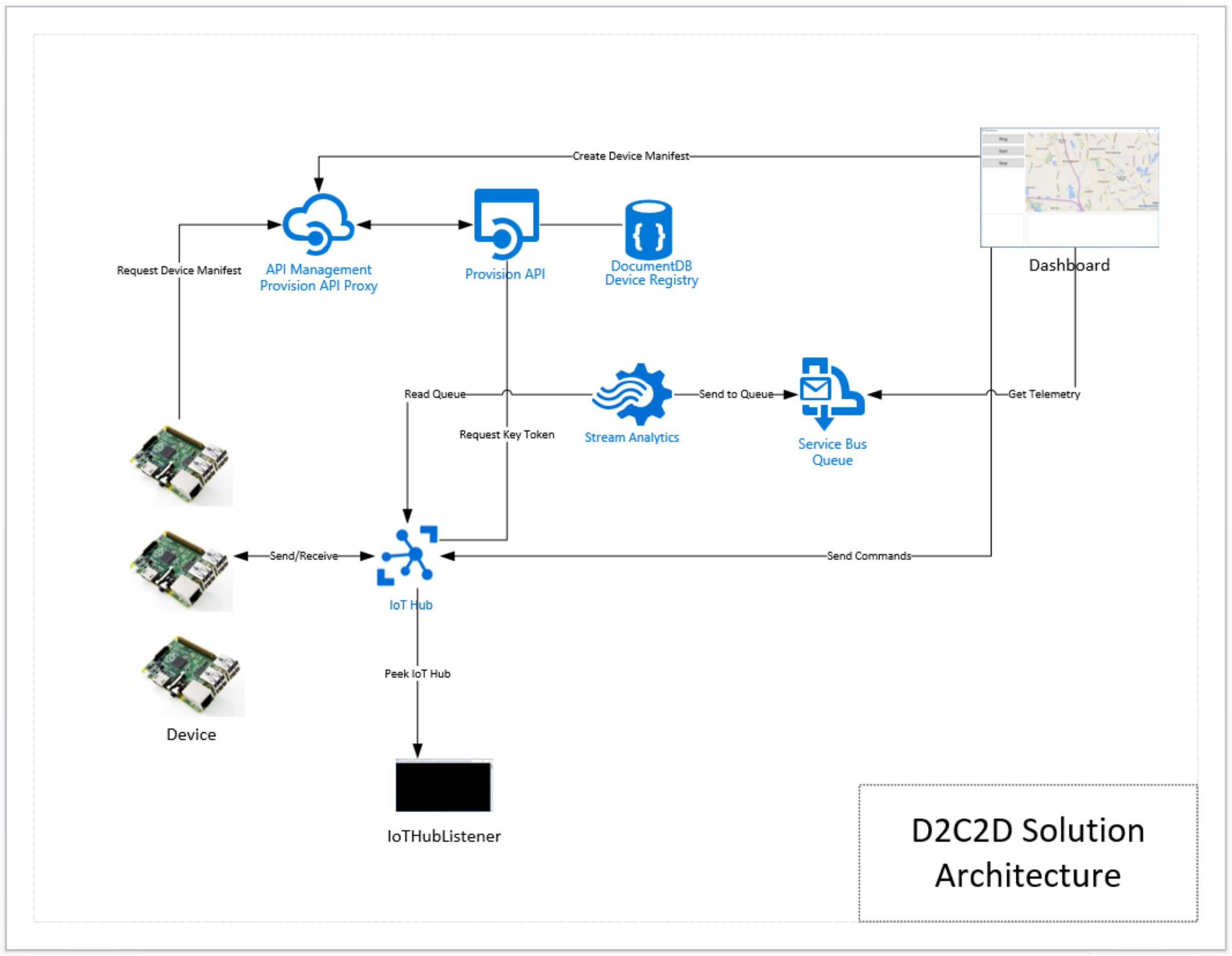
This training program provides foundational knowledge in how to architect and implement an IoT solution using Windows 10 Core IoT hardware devices and Azure IoT Hub and Stream Analytics. Both Device-to-Cloud and Cloud-to-Device communication patterns are discussed, designed, and implemented using best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

At the conclusion of this workshop you will have provisioned, 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

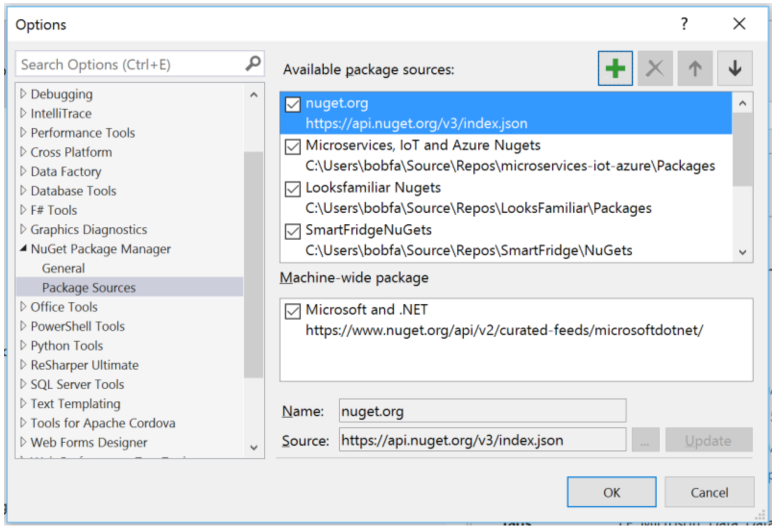
- **IoT Hub –** IoT Hub provides device registration, incoming telemetry at scale, and cloud-to-device message services
- **DocumentDb** – DocumentDb is a NoSQL database service that is used for managing Device Manifests, i.e., a Device Registry
- **Stream Analytics –** the solution leverages two Stream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

Lab One Overview

In this lab your will configure your development environment, provision Azure services, and review the 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 2 - https://www.visualstudio.com/news/vs2015-update2-vs - Azure SDK 2.9 (**for Visual Studio 2015**) - https://azure.microsoft.com/en-us/blog/announcing-visual-studio-azure-tools-and-sdk-2-9/ - Azure PowerShell 1.3.0 – http://aka.ms/webpi-azps - 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 - Navigate to the root of the expanded repo, run the PowerShell console and enter this command to unblock the downloaded files: ➢ PS C:\Users\mshir\Source\d2c2d> gci .\ -recurse | unblock-file
3	**Add the repo's *NuGets* folder to your Visual Studio Environment** Follow this menu path in Visual Studio: Tools Menu → NuGet Package Manager → Package Manager Settings. The NuGet Package Manager Settings dialog will appear. Select Package Sources.


Options

Search Options (Ctrl+E)

Available package sources:

- ☑ nuget.org
  https://api.nuget.org/v3/index.json
- ☑ Microservices, IoT and Azure Nugets
  C:\Users\bobfa\Source\Repos\microservices-iot-azure\Packages
- ☑ Looksfamiliar Nugets
  C:\Users\bobfa\Source\Repos\LooksFamiliar\Packages
- ☑ SmartFridgeNuGets
  C:\Users\bobfa\Source\Repos\SmartFridge\NuGets

Machine-wide package

- ☑ Microsoft and .NET
  https://www.nuget.org/api/v2/curated-feeds/microsoftdotnet/

Name: nuget.org

Source: https://api.nuget.org/v3/index.json

- 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 your repo (that directory is currently empty except for a placeholder.txt file).
- 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 and execute the following commands. Note: You 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

Other useful resources

- Using Azure PowerShell with Azure Resource Manager

5 Provision Foundational Services

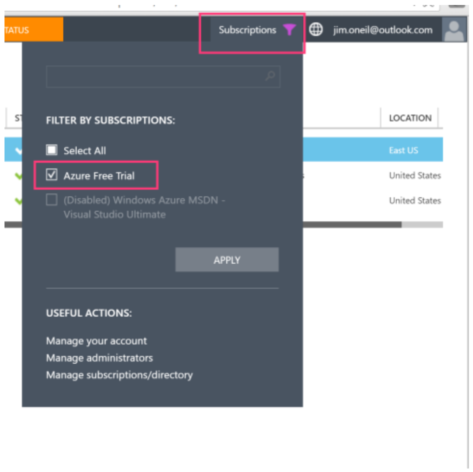
Navigate to the *Automation* folder of the D2C2D repo and run the 01-Provision-IO.ps1 script. Enter the parameters as you are prompted:

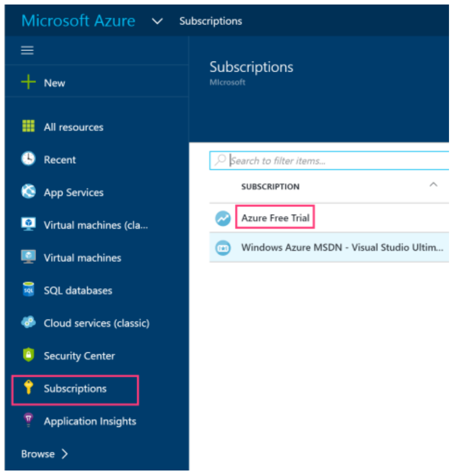
- .\01-Provision-IO.ps1
  Subscription: [the name of your subscription – ***see images below***]

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; ***component names need to be universally unique, so pick something non-trivial***]
Suffix: [dev | tst | stg | prd] – used to differentiate resources used for the different development phases

Tip: Make a note of the values provided above; you'll be supplying them to additional PowerShell scripts in subsequent labs.

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

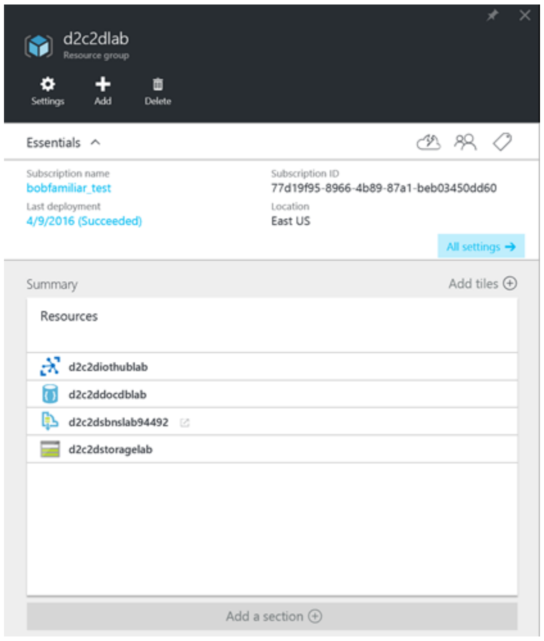




This script creates:

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

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



By the way, the connection strings for these services are written to a JSON file called provision-[ResourceGroupName]-output.json, which is then parsed by the EnvironmentVariables.ps1 to load these connections strings for use by the other scripts. This 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 is a common message model shared by the device and the cloud services that process the messages 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 comprises the following classes:

- Climate – climate telemetry class
- ClimateSettings – climate telemetry value bounds (*included in Climate.cs file*)
- Command – command class
- DeviceManifest – device details
- Location – location class
- MessageBase – base class for all messages
- Ping – simple heartbeat message implementation

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 the 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));
        }
    }
```

Ping

Prior to implementing device specific messages, it's a good idea to verify end-to-end connectivity with a simple heartbeat type message. The Ping class does just that, sending a 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 be used in subsequent labs.