|  |  |
| --- | --- |
|  | |
|  |  |
|  | https://www.bluemetal.com/~/media/Images/CSS/bluemetal-logo |

d2c2d

Lab Workbook Two

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

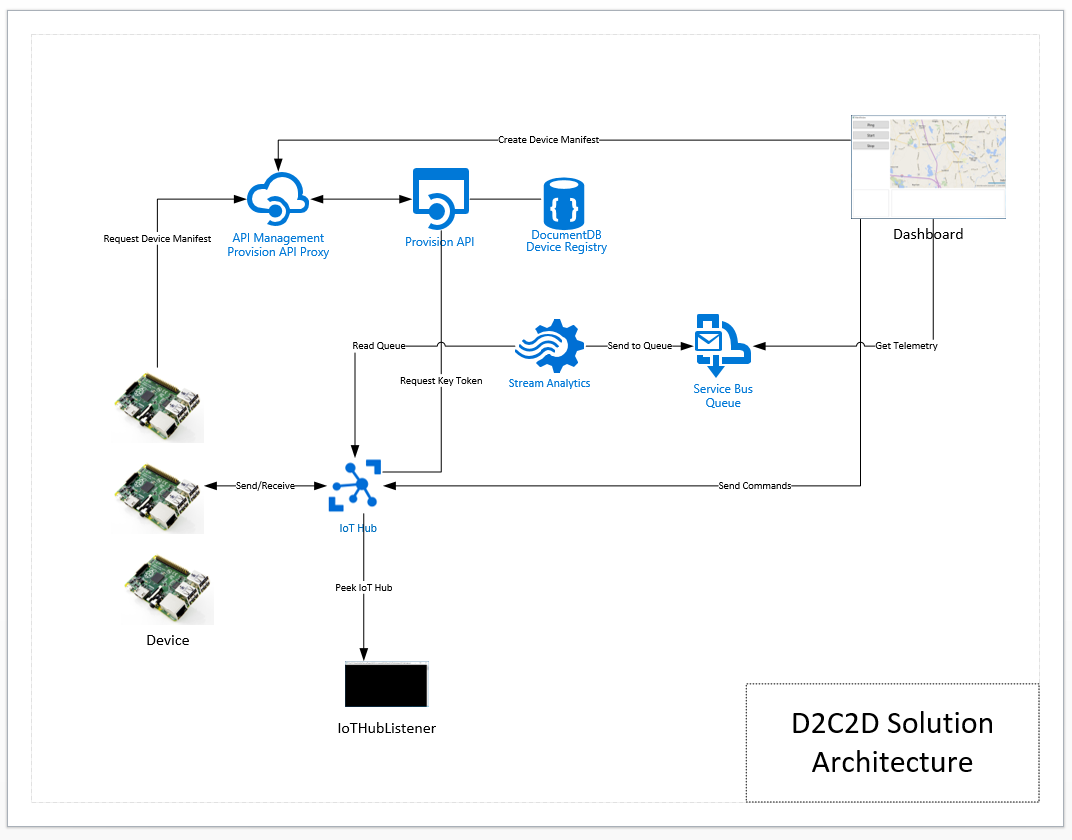
This training program provides foundational knowledge in how to architect and implement an IoT solution using Windows 10 Core IoT hardware devices and Azure IoT Hub and Stream Analytics. Both Device-to-Cloud and Cloud-to-Device communication patterns are discussed, designed, and implemented using best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

In this lab you will update, build, and deploy a microservice called Provision that provides endpoints for registering a new device with IoT Hub registration and storing the device’s manifest in DocumentDb. You will then create the Dashboard solution, configured with the appropriate connection strings, and use the Dashboard to provision a new device.

# Lab

|  |  |
| --- | --- |
| **Step** | **Details** |
| **1** | **Complete the Provision Microservice Implementation**  The Provision Microservice consists of an ASP.NET WebAPI solution and several projects that are built as NuGet packages. The WebAPI solution references the other packages and defines the ReST routes deployed to Azure.  The relationships among the projects are depicted here:    Microservice Project Construct   * **Model** – defines the messages that are passed into and out of the microservice * **Interface** – defines the interface that is implemented by both the Service and the SDK * **Wire** – protocol-level classes that provide ReST invocation capabilities * **Store** – classes that provide an interface to DocumentDb * **Service** – the implementation of the service * **API** – defines the HTTP routes and is deployed as an AppService to Azure * **SDK** – class library that provides a high level access to the deployed microservice * **Client** – any application that uses the SDK   What you will note is that the Model and the Interface are reused by both the Service and the SDK. The API solution defines the HTTP ReST routes, and the Service provides the implementation. The SDK exposes the same interface as the Service and uses the Wire library to invoke the ReST API.  In this step, you will complete the implementation of the API.   * From within Visual Studio, navigate to the microservices\provision\API folder and open the ProvisionAPI solution file. * Open the Controllers\ProvisionMControllers.cs file and add an additional endpoint for retrieving a device manifest by id * Cut-and-paste the following code within the ProvisionMController class, below the constructor method:   [Route("provision/devicemanifests/id/{id}")]  [RequireHttps]  [HttpGet]  public DeviceManifest GetById(string id)  {  return \_provisionM.GetById(id);  }  Save the solution and close Visual Studio. |
| **2** | **Build and Deploy the Provision Microservice**  From within the PowerShell console, navigate to the automation folder at the top level of your source tree and type the following command:   * 03-Provision-Microservices   **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]  **Configuration**: [debug | release]  **Note that the parameters are the same as those used in Lab 1 to deploy the foundation services.** Validate that the deployment process has completed by navigating to the blade for your Azure Web site, and click on the link to the service home page. |
| **3** | **Dashboard**  The Dashboard application will provide a command and control platform for the lab environment. From the dashboard you will be able to provision devices, start and stop telemetry, see the location of the devices, and view the incoming telemetry and alarm states. In this step of the lab, you will augment the implementation of the Dashboard.     * From within Visual Studio, open the Dashboard solution. * Open the app.config file and update the values for each of the following app settings   + Service Bus Connection String – retrieve from the Azure *Classic* Portal by selecting the Service Bus and clicking the “Connection Information” (Key) icon at the footer of the portal.   + IoT Hub Connection String – retrieved from the Azure Portal as follows:  1. Select the IoT hub from your list of resources 2. Select the Settings Icon to open the Settings blade 3. Select the Shared access policies item in the GENERAL group 4. Select iothubowner in the Shared access policies blade 5. Copy the contents of the “Connection string – primary key” field   + IoT Hub Name – full IoT Hub Hostname (e.g., myhub.azure-devices.net) from the Azure Portal   + Provision API – the URI for the Provision API; this is the link to the Azure App Service you created in the last lab, with the resource name *provision* appended:  https://*<prefix>*provisiontapi*<suffix>*.azurewebsites.net/provision/   <appSettings>  <add key="ServiceBusConnStr" value="" />  <add key="IoTHubConnStr" value="" />  <add key="IoTHubName" value="" />  <add key="ProvisionAPI" value="" />  </appSettings>  The Dashboard requires several NuGet package references, some that are specific to Azure and a couple that you have built locally, namely the Message Model and the Provision SDK.   * Open up the NuGet Manager Dashboard in Visual Studio * With the source set to NuGet.Org, search for Microsoft Azure Devices. Add a reference to the Microsoft.Azure.Devices NuGet package * With the source set to NuGet.Org, search for Service Bus. Add a reference to the WindowsAzure.ServiceBus NuGet package * With the source set to the d2c2d NuGets folder location, add a reference to the MessageModelsNet4 NuGet package * With the source set to the d2c2d NuGets folder location, add a reference to the ProvisionSDK NuGet package   Your device will provide its longitude and latitude on each incoming message. The Dashboard uses Bing Maps to display the location of our devices.   * Using the Add Reference dialog, click on Browse and navigate to C :\Program Files (x86)\Bing Maps WPF Control\V1\Libraries to add a local DLL reference to the Bing Maps WPF Control DLL * Open the Main.Windows.Xaml file and uncomment the reference to the WPF Control isn the MainWindow.Xaml file (Line 34, beginning <!--<m:Map x:Name="MyMap… ) * Add your Bing Maps Key as the Credential Provider property   Update the code behind for the Main Window:   * Open the MainWindow.xaml.cs file * Add the following using statements:   using Looksfamiliar.d2c2d.MessageModels;  using LooksFamiliar.Microservices.Provision.SDK;  using Microsoft.Azure.Devices;  using Microsoft.ServiceBus.Messaging;  using Newtonsoft.Json;  using Map = Microsoft.Maps.MapControl.WPF; using System.Windows.Threading;  You will need a set of members defined at the class level so that all the methods in the Main class can access them.   * Define the following class members for working with Azure Service Bus and the Provision SDK (directly above the line public MainWindow())   private readonly QueueClient \_messageClient;  private readonly QueueClient \_alarmClient;  private ServiceClient \_serviceClient;  private ProvisionM \_provisionM;  private DeviceManifest \_currDevice;   * Replace the MainWindow constructor to initialize the member variables with the following:   public MainWindow()  {  InitializeComponent();  \_messageClient = QueueClient.CreateFromConnectionString(  ConfigurationManager.AppSettings["ServiceBusConnStr"], "messagedrop");  \_alarmClient = QueueClient.CreateFromConnectionString(  ConfigurationManager.AppSettings["ServiceBusConnStr"], "alarms");  \_serviceClient = ServiceClient.CreateFromConnectionString(  ConfigurationManager.AppSettings["IoTHubConnStr"], TransportType.Amqp);  \_provisionM = new ProvisionM  {  ApiUrl = ConfigurationManager.AppSettings["ProvisionAPI"],  DevKey = ConfigurationManager.AppSettings["DeveloperKey"]  };  }   * In the MainWindows.xaml.cs file, uncomment the following routines:   + MainWindow\_OnLoaded()   + GetDeviceList()   + GetLocationAsync()   + DeviceList\_OnSelectionChange() routines   **Tip:** To uncomment a block of lines in Visual Studio, select the lines and press and hold the Ctrl key, then press K followed by U. |
| **4** | **Provision a Device**  Before a device can connect to IoT Hub and send telemetry, it must be registered. Additionally, you’ll want to be able to easily build a list of registered devices and retrieve the metadata, such as model number, firmware revision, longitude, latitude, etc. In this step you will add the code that provisions a new device with IoT Hub and stores the manifest in DocumentDb. The ProvisionAPI that you built and deployed in the previous step provides this capability.   * In the MainWindow.xaml.cs file, locate the ProvisionButton\_Click event handler; you’ll be adding code to this implementation in several phases below   private async void ProvisionButton\_Click(object sender, RoutedEventArgs e)   * Add a call to obtain you location based on your IP address. The implementation of GetLocationAsync is also included in the MainWindow.xaml.cs; it leverages a public API ip-api.com to identify your location.   var location = await GetLocationAsync();   * Create an instance of a DeviceManifest and initialize its properties:   // initialize a device manifest  var manifest = new DeviceManifest  {  latitude = location.lat,  longitude = location.lon,  manufacturer = "Looksfamiliar, Inc",  model = "Weather Station - Win 10 Core IoT",  firmwareversion = "1.0.0.0",  version = "1.0.0.0",  hub = ConfigurationManager.AppSettings["IoTHubName"],  serialnumber = "d2c2d-" + Guid.NewGuid()  };  manifest.properties.Add(new DeviceProperty("Hardware Platform", "Raspberry PI"));   * Call the Create endpoint of the ProvisionM API (a message box will display if successful)   // provision the device in IoT Hub and store the manifest in DocumentDb  manifest = \_provisionM.Create(manifest);  MessageBox.Show($"New Device Provisioned: {manifest.serialnumber}",  "Confirmation",  MessageBoxButton.OK);   * Update the DeviceList list box on the Dashboard UI   DeviceList.Items.Clear();  var devices = \_provisionM.GetAll();  foreach (var device in devices.list)  {  DeviceList.Items.Add(device.serialnumber);  }  if (DeviceList.Items.Count <= 0) return;  StartButton.IsEnabled = true;  StopButton.IsEnabled = true;  PingButton.IsEnabled = true;  }   * Compile and run the Dashboard. Click the New Device button. You can provision multiple devices. |
| **5** | **Congratulations! You have completed Lab 2**  Let’s review:   * You completed the Provision Microservice implementation * You built the Common Framework libraries * You built and deployed the Provision Microservice * You updated the Dashboard * You provisioned a Device |