BLUEMETAL
An Insight company

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

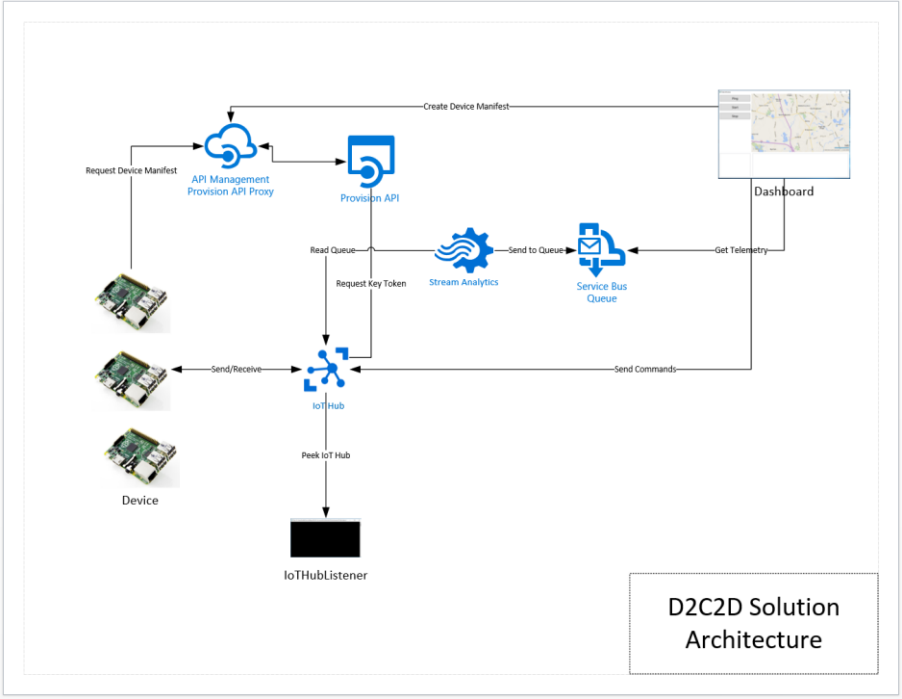
Workshop Overview

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

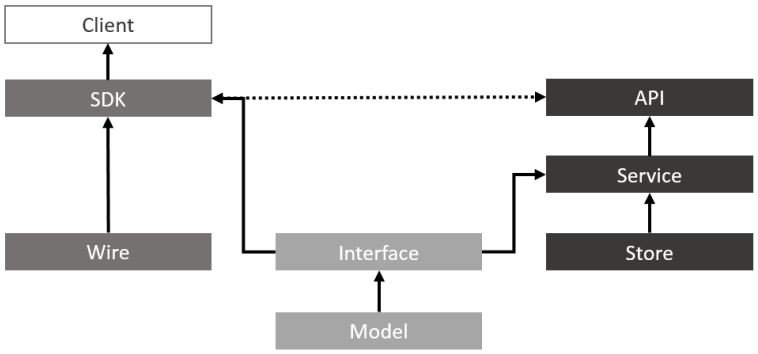
And the following Azure Services

- **API Management** – provides proxy, policy injection and developer registration services for ReST APIs
- **Service Bus Namespace** – 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
- **Stream Analytics Job –** two 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 Two Overview

In this lab your will update, build and deploy a microservice called Provision that will provide end points for registering a new device with IoT Hub registration and storing the device's manifest in DocumentDb. You will configure the Provision API within API Management. You will then create the Dashboard solution, configured with the appropriate connections strings and use the Dashboard to provision a new device.

Lab

Step	*Details*
1	**Complete the Provision Microservice Implementation** The Provision Microservice consists of several projects each of which is built as NuGet packages. The one that is not built as a NuGet package is the API solution. The API solution references the other packages, defines the ReST routes so that they can be deployed to Azure. The relationship between the projects is 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** – lasses that provide ReST invocation capabilities - **Store** – classes that provide an interface to DocumentDb - **Service** – provides the implementation of the service - **API** – defines the HTTP routes and is a deployable solution - **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 will use 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 end point for retrieving a device manifest by id

```
[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 the Common Frameworks and Provision API

- Open Azure PowerShell console as administrator
- Navigate to the microservices/common/automation
- Type the following command to build the common libraries

➢ .\Build-Common
Repo: [the path to the top level of your repo]
Configuration: [Debug | Release]

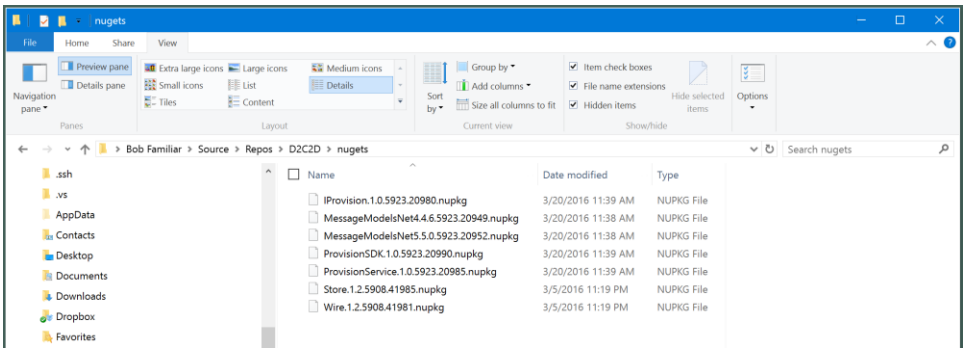
This script will build the NuGet packages and drop them into the NuGets folder at the top level of your repo,

3 Build the Provision Microservice

- From within the PowerShell console, navigate to microservices/provision/automation
- Type the following command to build all the solutions that comprise the microservice

➢ 02-Build-ProvisionM
Repo: [the path to the top level of your repo]
Configuration: [Debug | Release]

Validate the build by using File Explorer to review the contents of the NuGets folder. You should see NuGet packages for the Models, the common libraries Wire and Store and the NuGets for the Provision Interface, Service and SDK.



4 Provision the Azure Environment for the Provision Microservices

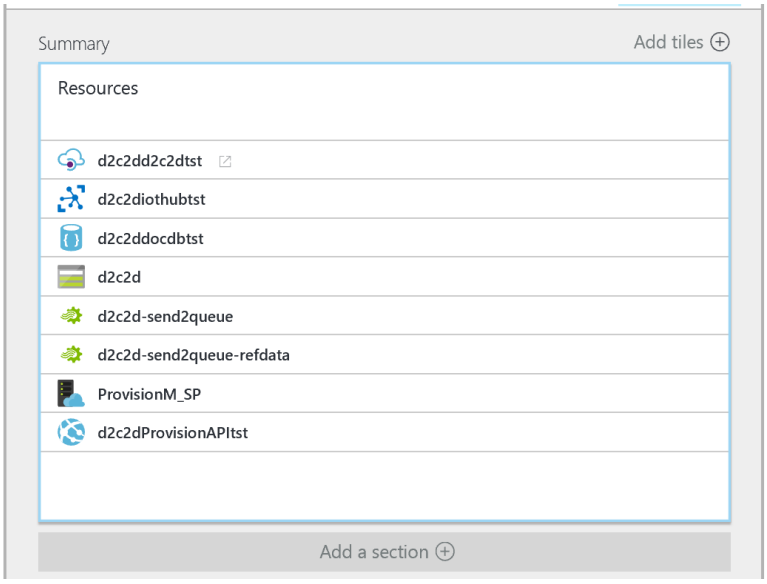
The Provision Microservice will be deployed as an Azure Web Site that is associated with an Azure Service Plan. In this step you will provision the Azure Web Site container and Service Plan.

- From within the PowerShell console, type the following command

➢ 01-Provision-ProvisionM
Repo: [the path to the top level of your repo]
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]

Note that the parameters are the same as those used in Lab 1 to provision the foundation services.

Validate that the provision process has completed by reviewing the contents of yoru Resource Group in the Azure Portal.



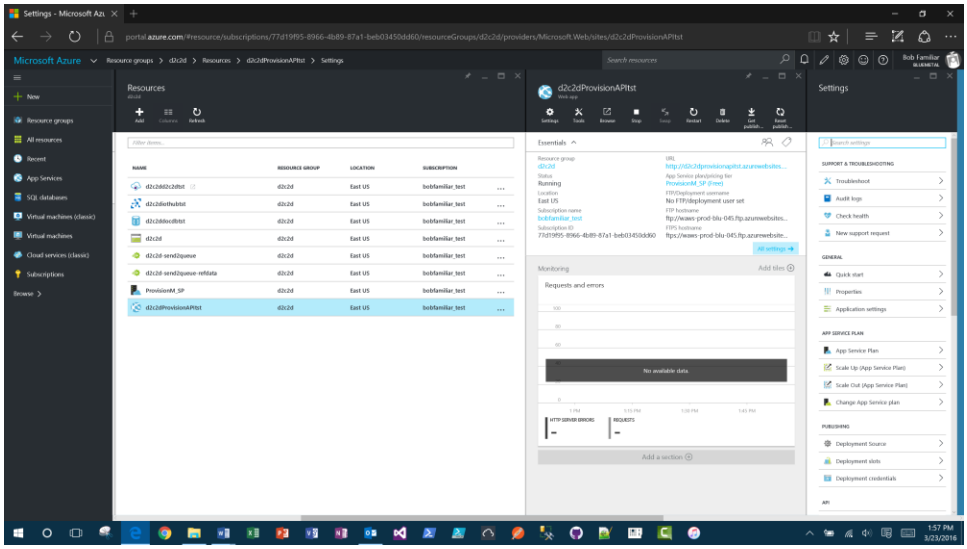
5 Deploy the Provision Microservice

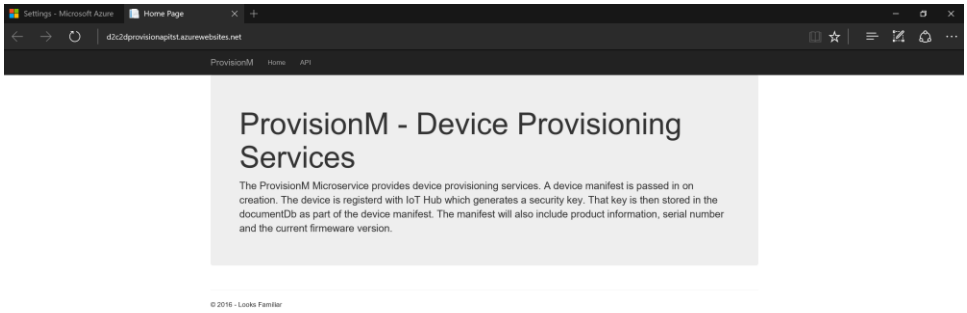
- From within the PowerShell console, type the following command

➢ 03-Deploy-ProvisionM
Repo: [the path to the top level of your repo]
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]

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.





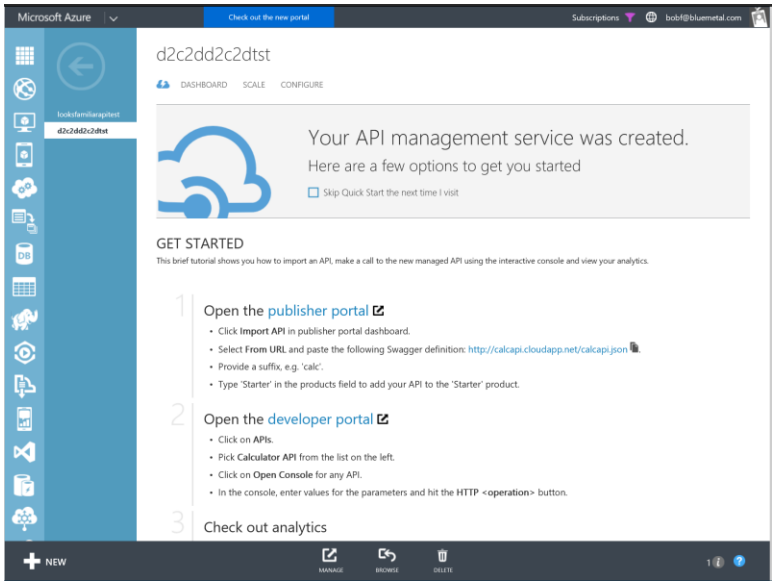
6 API Management

API Management is used to define a proxy end point for your ReST APIs. At the proxy level you can then gather analytics, engage a subscription model, inject policies such as throttling or custom headers and more.

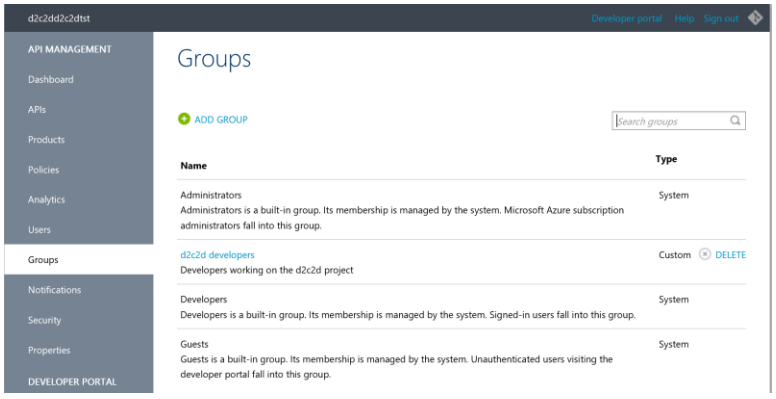
In this step you will configure a proxy for the Provision API.

- From the Azure Portal, click on your API Management instance from within your Resource Group. It will look something like this:

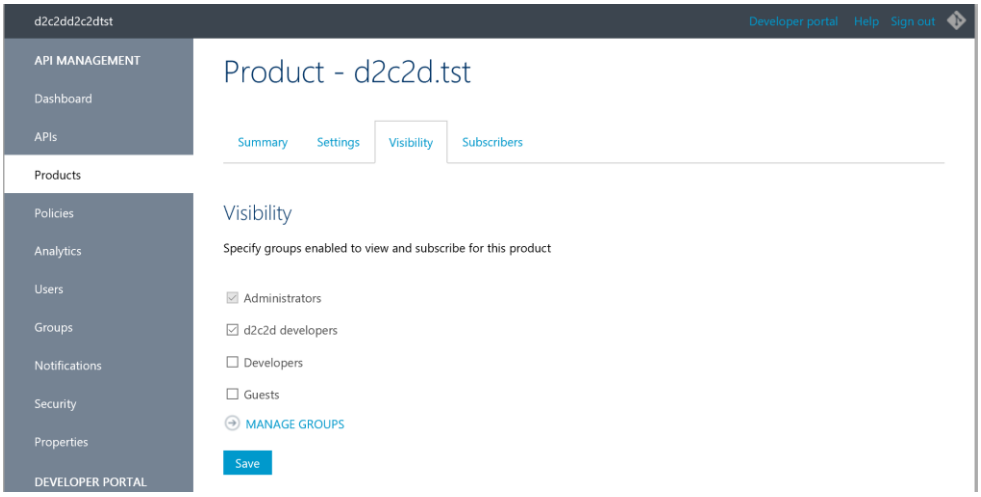
- This will bring you to the 'classic' portal. From there, click the Manage button on the bottom tool bar



- This will bring you to the API Management - Management Portal. Click Groups on the left hand menu and add a group called 'd2c2d developers'. Provide a simple description and click save.

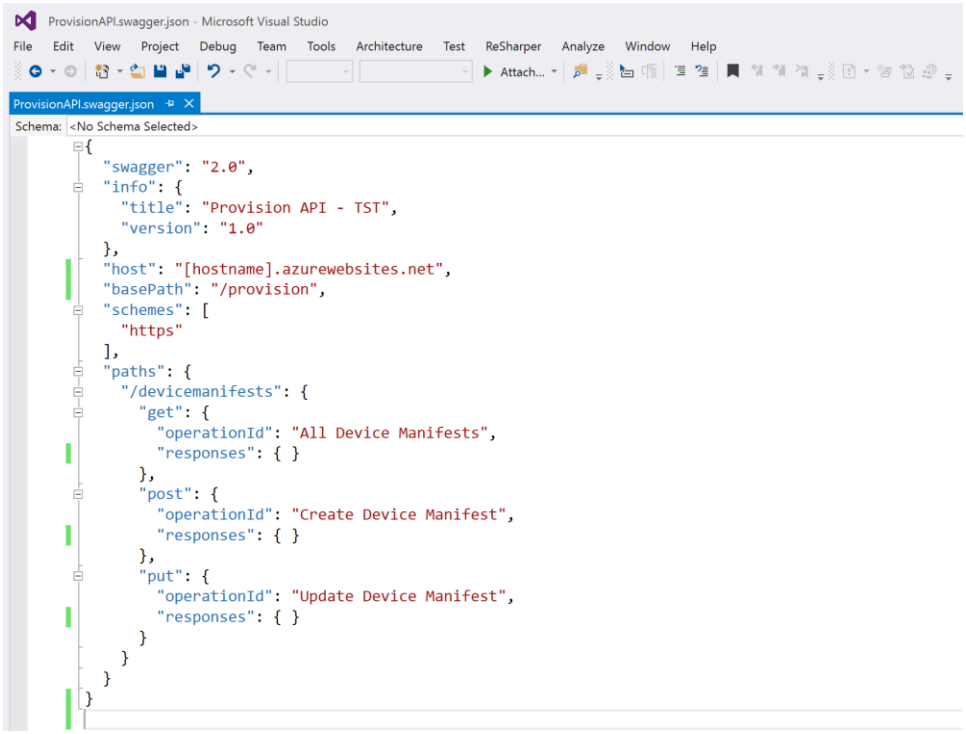


- Click Products in the left hand menu and create a product called ‘d2c2d.tst’. Once defined, select the Visibility Tab and make this product visible only to Administrators and the d2c2d developer group and click Save.

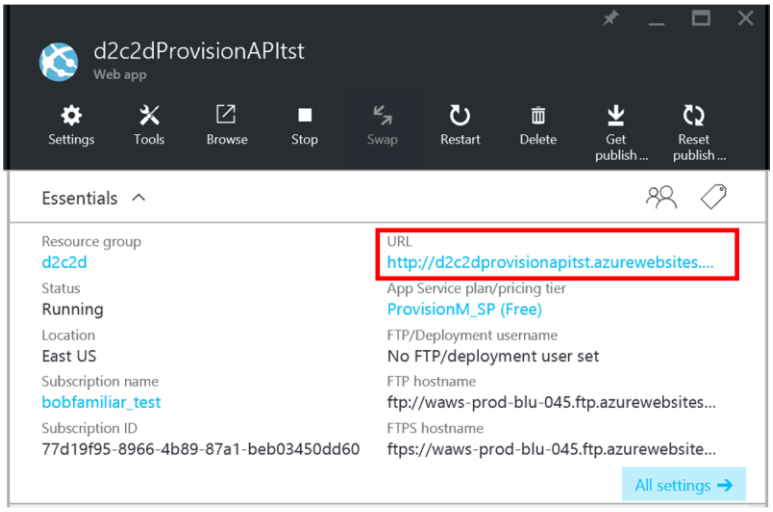


We will use the Import API feature to import a Swagger definition for the Provision API. Before we import it though, we will want to edit the Swagger definition to add in the location of your API in Azure and add an additional end point.

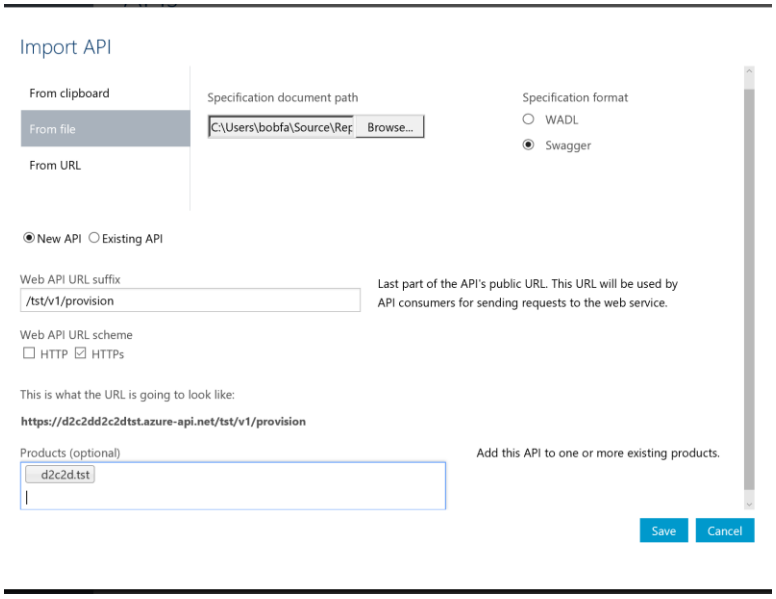
- Using Visual Studio, open the file called ProvisionAPI.swagger.json in the Lab 2 folder.



- Edit the host property adding your APIs host name and then save the file. You can find that value on the blade in the Azure Portal:

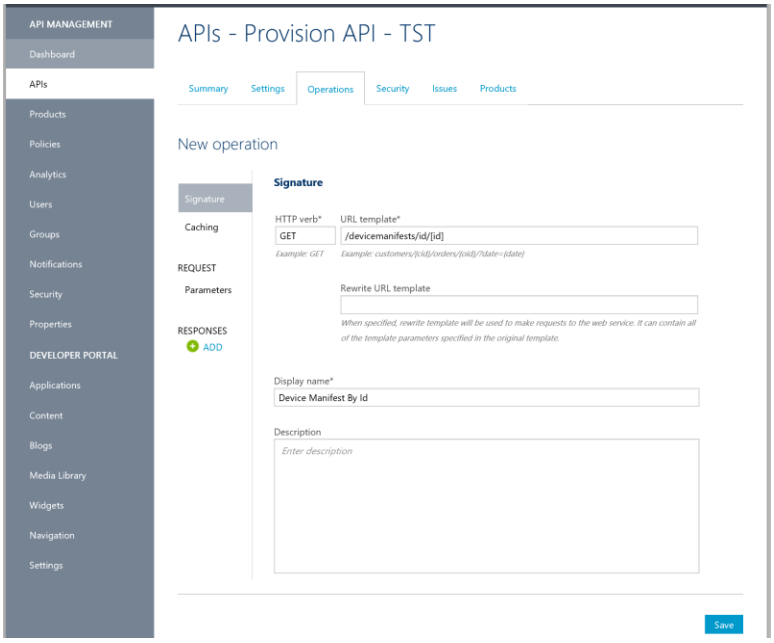


- From the API Management Portal, click APIs in the left hand menu and then Import API.
- Select ‘From file’, Swagger format and then browse to the Swagger definition file.
- Type in the Web API URL suffix to be ‘/tst/v1/provision’
- Add the API to the d2c2d.tst product. Click Save.



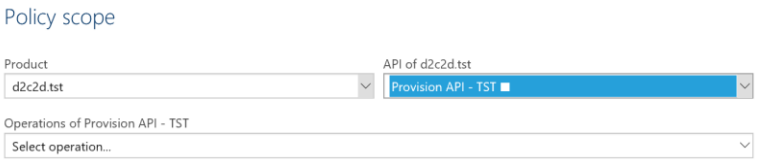
In this step, we will expose an additional end point that allows a developer to retrieve a device manifest by passing in its unique id.

- Click on the Operations Tab and then New Operation
- Select the GET verb
- Set the URL template to be ‘/devicemanifests/id/{id}’
- Change the name to be ‘Device Manifests by Id’
- Click Save

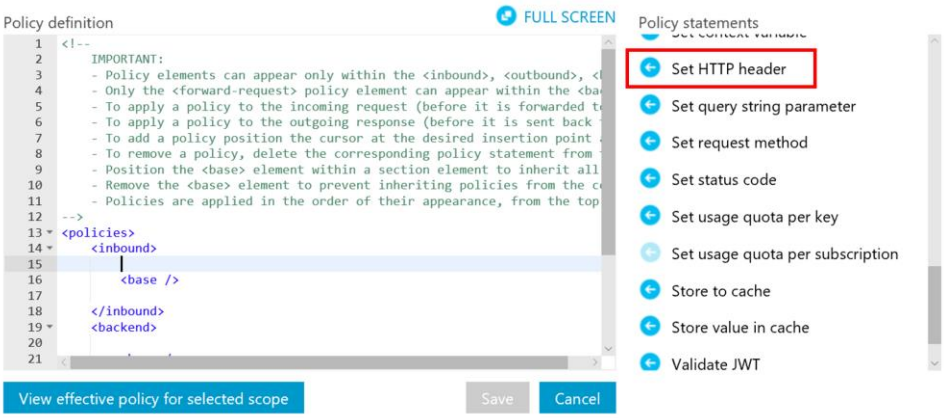


Now that the API proxy has been defined, we can specify a Policy that will create a custom HTTP header that contains a shared secret. The shared secret is a Guid that only the API Proxy and the ReST API know. That value is passed into the API which then compares it with its shared secret and if they match the call is allowed to proceed. This adds an additional level of security between the proxy and the API in addition SSL.

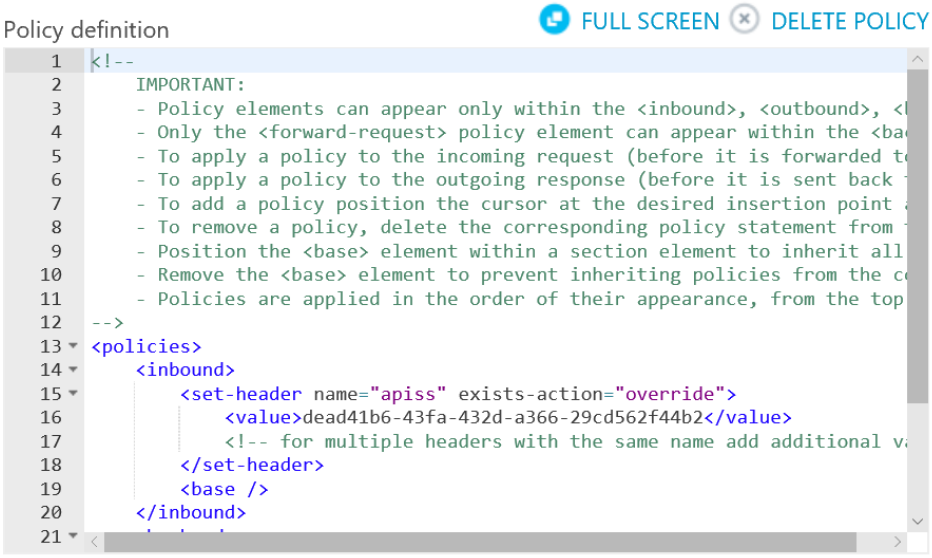
- Click Policies in the left hand menu
- Set the Policy Scope to the Product and API and just defined



- Click Add Policy to activate the policy editor
- Place the cursor in the XML file just below the <inbound> tag and before the <base> tag.
- Scroll down to the Set HTTP Header option and click to inject that policy into the configuration file



- Change the header name to ‘apiss’ and the exists-action setting to ‘override’.
- Add the shared secret value. You can find this value in the Include-ConnectionStrings.ps1 file located in the Automation folder.

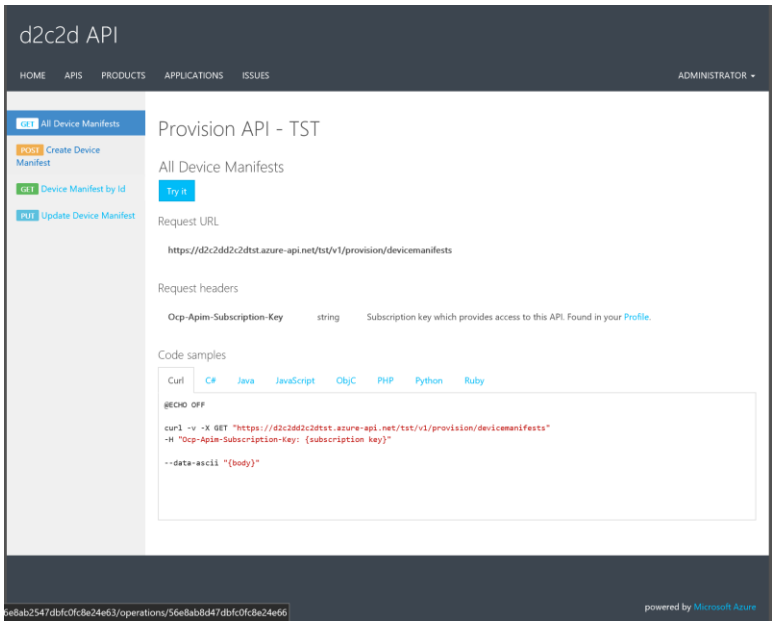


NOTE: If you want to generate a new shared secret, there is a utility in the Tools folder that will generate new Guids. Make sure to update both the Include-ConnectionStrings.ps1 file, this policy definition and also redeploy the API to complete the configuration update.

- Click Products in the left hand menu and then click on your product to navigate to its Summary page.
- Click the Publish button to publish product to the Developer Portal making it available for developers who are in the appropriate security group
-

The final step is to test the API to make sure all the proxy and policy settings are working. We will navigate to the Developer Portal and test the GET operations of the API. The Developer Portal is generated when you provision the API Management Service. This is the portal that developers will use to request access to APIs and learn how to use them.

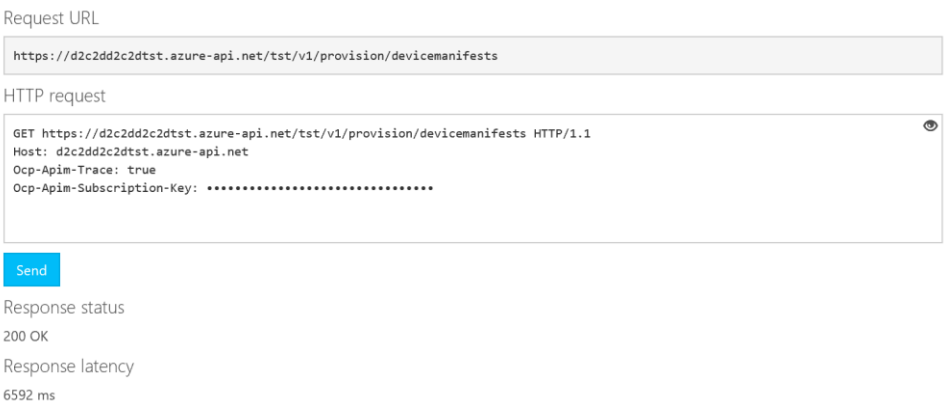
- In the upper right hand corner of the API Management Portal, click the Developer Portal link.
- Click on APIs in the menu bar and then click on the name of your API. You will see the list of operations on the left hand side and documentation on the right hand side of the page.



- Click the Try It button for the Get All Manifests operation
- Click the Send Button

Note: there isn't any data in the data base yet for devices so what we expect to get back is an empty list. That is OK. What we are looking for here is that the call succeeds in getting from the proxy all the way back to the database and returns without error.

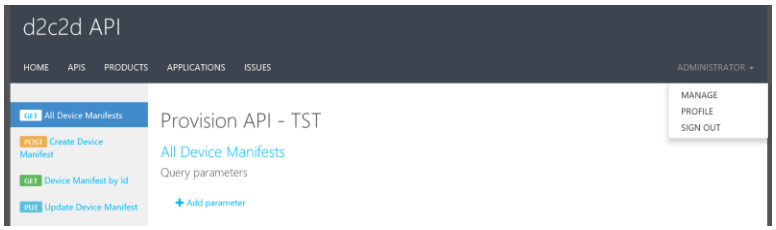
- Look for a Response Status = 200 OK



In order for a developer to call a managed API, they would need to register on the portal and then be assigned by the administrator to be a member of one or more developer groups and

approved to be able to access API Products that those groups are allowed to call. For purposes of the lab, we will use Administrator level access to the APIs.

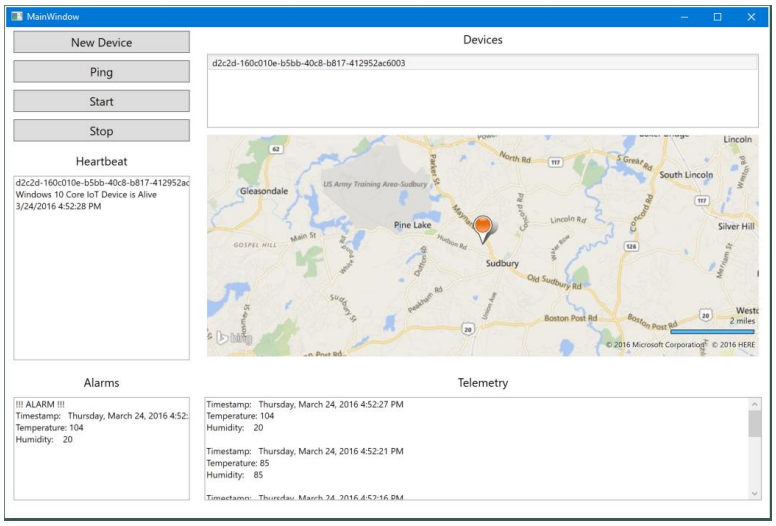
- Click on Administrator and then Profile in the upper right hand corner of the Developer portal



- Click the 'Show' link for your product and copy the key value. Put this aside in a notepad file. We will use it in the next step of the lab.

7 Dashboard

The Dashboard application will provide us a command and control platform for our lab environment. From the dashboard we will be able to provision devices, start and stop telemetry, see the location of our devices and see the incoming telemetry and alarm states. In this step of the lab, we you will begin 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
    - IoT Hub Connection String – retrieve from the Azure Portal
    - IoT Hub Name – retrieve from the Azure Portal

    - Provision API – the managed URI for the Provision API Proxy
    - Developer Key – the API Management Developer Key

```
<appSettings>
    <add key="ServiceBusConnStr" value="" />
    <add key="IoTHubConnStr" value="" />
    <add key="IoTHubName" value="" />
    <add key="ProvisionAPI" value="" />
    <add key="DeveloperKey" value="" />
</appSettings>
```

The Dashboard will make use of several NuGet packages, some that are specific to Azure and a couple that we have built in our environment, 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 Microsoft.ServiceBus.Messaging 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

Our 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 is the MainWindow.Xaml file
- 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 Microsoft.Maps.MapControl.WPF;
```

We 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

```
private readonly QueueClient _messageClient;
private readonly QueueClient _alarmClient;
private ServiceClient _serviceClient;
private ProvisionM _provisionM;
private DeviceManifest _currDevice;
```

- Update the MainWindow routine to initialize the member variables

```
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

8 Provision a Device

Before a device can connect to IoT Hub and send telemetry, it must be registered. In addition, we want to be able to easily build a list of registered devices and retrieve the meta-data about that device such as its model number, firmware revision, longitude, latitude, etc. In this step we will add the code that provisions a new device with IoT Hub and stores the manifest in DocumentDb. The ProvisionAPI that we built and deployed in the previous step provides this capability.

- In the MainWindow.xaml.cs file, locate the ProvisionButton_Click event handler

```
private async void ProvisionButton_Click(object sender, RoutedEventArgs e)
```

- Add a call to the GetLocationAzync() routine. This routine calls a public API to get yoru current longitude/latitude based on your IP address. We will use this information to initialize the Device Manifest.

```
    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"));
```

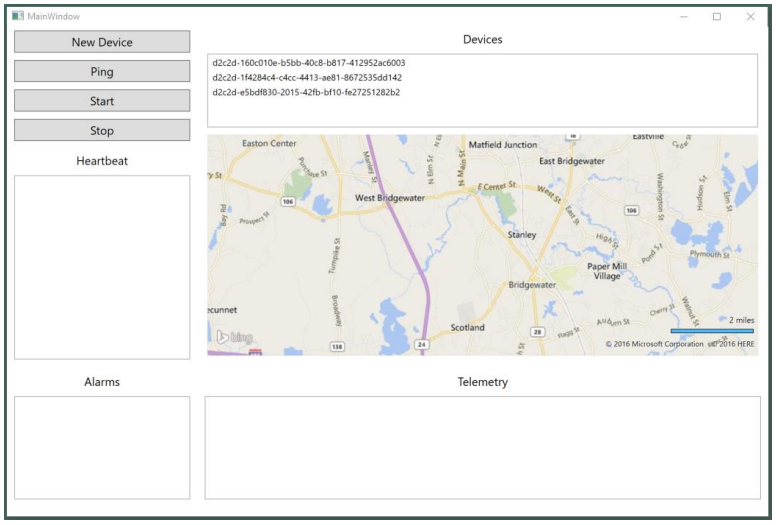
- Call the Create end point on the ProvisionM API and display a message box if successful

```
        // 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);
```

- Add code to 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. We will use one of these provisioned devices in the



9 **Congratulations! You have competed Lab 2**

Let's review:

- You completed the Provision Microservice implementation
- You built the Common Framework libraries
- You built the Provision Microservice
- You provisioned the Azure environment for the microservice
- You deployed the Provision Microservice
- You configured API Management
- You updated the Dashboard
- You provisioned a Device