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

d2c2d

Lab Workbook Four

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

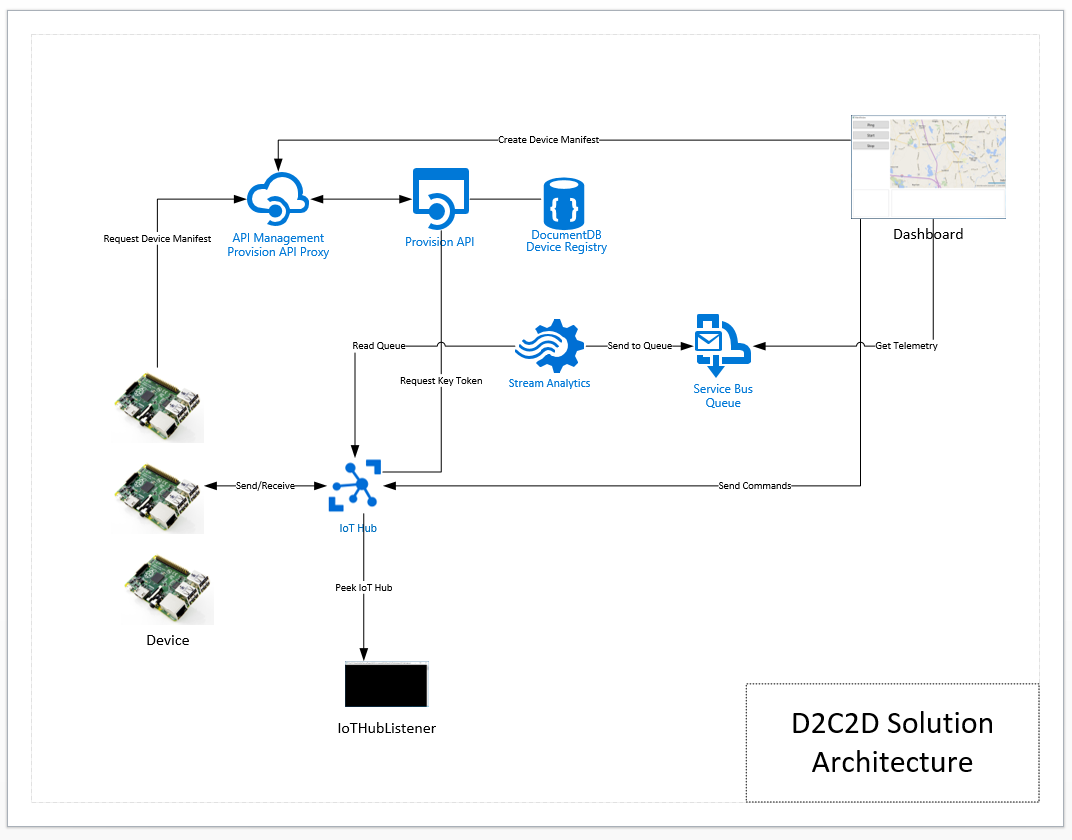
This training program provides foundational knowledge in how to architect and implement an IoT solution using Windows 10 Core IoT hardware devices and Azure IoT Hub and Stream Analytics. Both Device-to-Cloud and Cloud-to-Device communication patterns are discussed, designed, and implemented using best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

In Lab Three, you set up the solution to support sending messages from the device to the cloud. In this lab you will add command and control to the solution. Command and control is all about sending messages from the cloud to the device. In IoT solutions, the messages that are sent from the cloud can be commands such as upgrade firmware, start or stop taking measurements or even text to display on an LCD screen if the device has one.

In the generalized case there is a command and parameters of the command, such as where to download the firmware file or which telemetry to start or stop. How to package command parameters will be covered in this lab.

# Lab

|  |  |
| --- | --- |
| **Step** | **Details** |
| **1** | **Update the Device solution to support receiving commands**  In order for the device to receive incoming messages, it must set up a background thread that listens on the channel provided by IoT Hub. The DeviceClient provides a ReceiveAsync() method for this purpose.   * Add the StartListenTask() method to the MainPage class:   private static void StartListenTask(TextBox status)  {  \_listenTask = Task.Factory.StartNew(async () =>  {  while (true)  {  var message = await \_deviceClient.ReceiveAsync();  if (message == null)  continue;  var json = Encoding.ASCII.GetString(message.GetBytes());  var command = JsonConvert.DeserializeObject<Command>(json);  switch (command.CommandType)  {  case CommandTypeEnum.Ping:  var ping = new Ping  {  Ack = AckMessage,  Longitude = \_deviceManifest.longitude,  Latitude = \_deviceManifest.latitude,  DeviceId = \_deviceManifest.serialnumber  };  json = JsonConvert.SerializeObject(ping);  var pingMessage = new Message(Encoding.ASCII.GetBytes(json));  try  {  await \_deviceClient.SendEventAsync(pingMessage);  }  catch (Exception err)  {  var errMessage = err.Message;  status.Text = errMessage;  }  break;  case CommandTypeEnum.Start:  // the command is to start telemetry  // unpack the parameters that define the upper and lower bounds  var settings = JsonConvert.DeserializeObject<ClimateSettings>(  command.CommandParameters);  \_sendingTelemetry = true;  StartTelemetry(settings, status);  break;  case CommandTypeEnum.Stop:  \_sendingTelemetry = false;  break;  case CommandTypeEnum.UpdateFirmeware:  // imagine  break;  default:  throw new ArgumentOutOfRangeException();  }  await \_deviceClient.CompleteAsync(message);  }  });  }   * Add a call to this new method as the last line of the MainPage\_OnLoaded event (directly below the call to StartPingTask(Status);   StartListenTask(Status);  One of the commands that is sent to the device from the dashboard is to start sending telemetry. Sending telemetry, like sending Ping messages, is a background thread where readings are taken and then sent as a message to IoT Hub. The loop within the background thread is controlled by the Boolean \_sendingTelemetry. Stopping telemetry is accomplished by setting this value to false.  When sending telemetry, the Device creates a Climate message and fills out the Temperature and Humidity measurements using the ClimateSettings that were passed in from the Dashboard.   * Add the StartTelemetry() method to the MainPage class :   private static void StartTelemetry(ClimateSettings settings, TextBox status)  {  \_telemetryTask = Task.Factory.StartNew(async () =>  {  while (\_sendingTelemetry)  {  var random = new Random();  var climate = new Climate  {  Longitude = \_deviceManifest.longitude,  Latitude = \_deviceManifest.latitude,  DeviceId = \_deviceManifest.serialnumber,  Temperature = random.Next((int) settings.MinTemperature,  (int) settings.MaxTemperature),  Humidity = random.Next((int) settings.MinHumidity,  (int) settings.MaxHumidity)  };  var json = JsonConvert.SerializeObject(climate);  var message = new Message(Encoding.ASCII.GetBytes(json));  try  {  await \_deviceClient.SendEventAsync(message);  }  catch (Exception err)  {  var errMessage = err.Message;  status.Text = errMessage;  }  await Task.Delay(5000);  }  });  } |
| **2** | **Update the Dashboard to Send Commands to the Device**  The Device uses an object called DeviceClient to connect and communicate with IoT Hub. In order to perform service-side operations with IoT Hub, the IoT Hub SDK provides the ServiceClient.  The ServiceClient is used by the Dashboard to send messages to specific devices. When you send a message to a device using IoT Hub, it actually sits in a queue and is either retrieved by the Device if it is in listen mode, or it expires.  For purposes of the lab, the Dashboard will send both Start Telemetry and Stop Telemetry messages.   * Replace the empty StartButton\_Click() and StopButton\_Click() event handlers with the following code;   private void StartButton\_Click(object sender, RoutedEventArgs e)  {  var climateSettings = new ClimateSettings  {  MinHumidity = 0,  MaxHumidity = 100,  MinTemperature = 75,  MaxTemperature = 110  };  var command = new Command  {  CommandType = CommandTypeEnum.Start,  CommandParameters = JsonConvert.SerializeObject(climateSettings),  DeviceId = \_currDevice.serialnumber  };  var json = JsonConvert.SerializeObject(command);  var message = new Message(Encoding.ASCII.GetBytes(json));  \_serviceClient.SendAsync(\_currDevice.serialnumber, message);  }  private void StopButton\_Click(object sender, RoutedEventArgs e)  {  var command = new Command  {  CommandType = CommandTypeEnum.Stop,  DeviceId = \_currDevice.serialnumber  };  var json = JsonConvert.SerializeObject(command);  var message = new Message(Encoding.ASCII.GetBytes(json));  \_serviceClient.SendAsync(\_currDevice.serialnumber, message);  }   * Test your implementation   + Start the Device Solution   + Start the Dashboard   + Click the Start Button   + You should see messages arriving in the Telemetry output window |
| **3** | **Congratulations! You have competed Lab 4**  Let’s review:   * You added the background threads in the Device solution to listen for commands and to start and stop sending telemetry * You updated the Dashboard to send commands to the Device |
|  |  |