

 [](http://www.netmf.com/)



IoT Kit Hands-on  
トレーニング

Step 3：蓄積編

改定：2015年4月14日

初版: 2014年 11月 29日

日本マイクロソフト株式会社

デベロッパー エクスペリエンス ＆ エバンジェリズム 統括本部

太田　寛

Twitter：@embedded\_george

Blog : http://blogs.msdn.com/hirosho

この文章に含まれる情報は、公表の日付の時点でのMicrosoft Corporationの考え方を表しています。市場の変化に応える必要があるため、Microsoftは記載されている内容を約束しているわけではありません。この文書の内容は印刷後も正しいとは保障できません。この文章は情報の提供のみを目的としています。

Microsoft、SQL Server、Visual Studio、Windows、Windows XP、Windows Server、Microsoft Azureは Microsoft Corporation の米国およびその他の国における登録商標です。

その他、記載されている会社名および製品名は、各社の商標または登録商標です。

© Copyright 2014 Microsoft Corporation. All rights reserved.

目次

[0. はじめに 4](#_Toc416034286)

[0.1 はじめに 5](#_Toc416034287)

[0.2 学習内容 6](#_Toc416034288)

[1. Webアプリで受信したデータを蓄積 7](#_Toc416034289)

[1.1 IoT Kitのコード修正 8](#_Toc416034290)

[1.2 ストレージアカウントの作成 9](#_Toc416034291)

[1.3 受信データのテーブルへの蓄積 10](#_Toc416034292)

[2. Event Hubで受信したデータを蓄積 13](#_Toc416034293)

[2.1 IoT Kit 側の修正 14](#_Toc416034294)

[2.2 Event Hubで受信 15](#_Toc416034295)

[3. 試してみよう 20](#_Toc416034296)

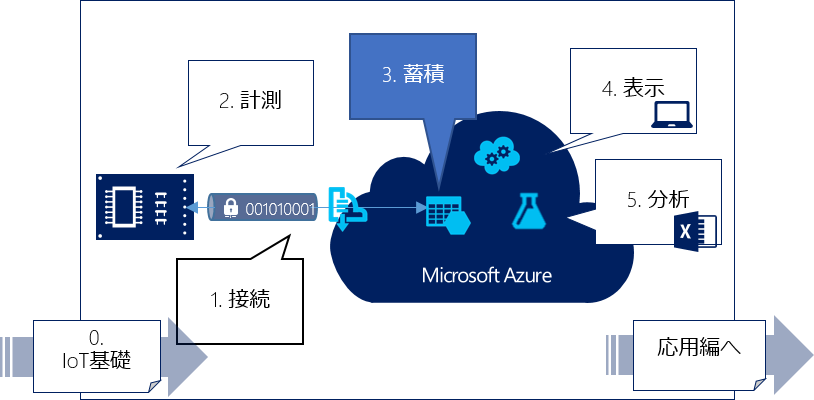
|  |
| --- |
| 1. はじめに |
|  |

## はじめに

本手順書は、一連のIoT Kit ハンズオントレーニングの三番目のステップで行う実習の具体的な手順を解説します。このステップの学習を開始する前に、前のステップの学習を行ってください。

## 学習内容

このステップでは、組込み機器でのセンサーによる計測と、クラウドへの計測データ送信方法を学びます。



実習は以下の順番に従って行います。

1. Webアプリで受信したデータを蓄積
   1. IoT Kitの送信コード修正
   2. ストレージアカウントの作成
   3. 受信データのテーブルへの蓄積
2. Event Hubで受信したデータを蓄積
   1. IoT Kitの送信コード修正
   2. 受信データのテーブルへの蓄積
3. 試してみよう

Visual Studioの使い方や、Gadgeteerのハードウェア設計等、前のステップまでで学習した項目は、本手順書では詳しい説明を省きます。やり方が分らない場合は、前のステップに戻って再学習してください。

実習には、.NET Micro Framework / Gadgeteer 対応の小型ハードウェア（IoT Kitと記載）を使用します。本手順書で使用しているハードウェア以外でも、実習は可能です。使用するハードウェアに合わせて、適宜読み替えて実習を行ってください。

2014年11月末日現在、標準のIoT Kit ハードウェアの準備が整っておりません。このバージョンでは、GHI Electoronics 社製のFEZ Spider、及び、各種センサーを使った場合の手順を解説しています。IoT Kitハードウェアの準備が整った段階で、改定の予定です。学習される皆様にはご迷惑をおかけしますが、ご了承ください。

|  |
| --- |
| 1. Webアプリで受信したデータを蓄積 |
| このステップでは、IoT Kitデバイスのセンサーで計測し、Web アプリに送信したデータをクラウドストレージに蓄積する方法を実習します。   * **IoT Kit の送信コードを修正** * **ストレージアカウントの作成** * **受信データのテーブルへの蓄積** |

## IoT Kitのコード修正

Step 2計測編2.1.の学習で作成したプログラムを修正します。

修正は、Program.cs のuploadTimer\_tick()メソッドに行います。

void uploadTimer\_Tick(GT.Timer timer)

{

uploadTimer.Stop();

var request = HttpWebRequest.Create("http://*[IoTCloud]*.azurewebsites.net/api/Sensor") as HttpWebRequest;

request.Headers.Add("device-id", deviceId.ToString());

request.Headers.Add("device-message", "hello from iot-kit");

lock (this)

{

**request.Headers.Add("sensor-type", "temperature");**

**request.Headers.Add("sensor-value", lastTemperature.ToString());**

}

using (var response = request.GetResponse() as HttpWebResponse)

{

if (response.StatusCode == HttpStatusCode.OK)

{

var reader = new StreamReader(response.GetResponseStream());

string message = reader.ReadToEnd();

Debug.Print(message);

}

}

uploadTimer.Start();

}

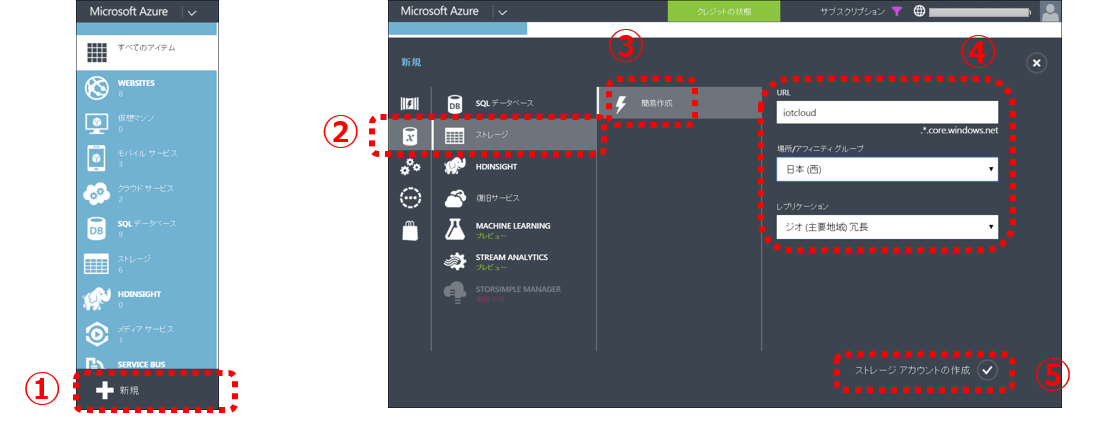
Step2では、温度と加速度の3軸の値をすべて送っていましたが、ここでは、クラウド側から見て様々なセンサーに対応可能なように、センサー種別とセンサー値を送る形式に変えています。修正は以上で終わりです。

## ストレージアカウントの作成

IoT Kitから送信されたセンサー計測データを、Azureのテーブルストレージに蓄積するために、まずは、ストレージアカウントを作成します。

<http://manage.windowsazure.com>

から管理ポータルを開き、



左下の“＋新規”をクリックします（①）。表示されたパネルで、“データサービス”(②)→“ストレージ”→“簡易作成”（③）を順にクリックしていきます。そして（④）URL、場所/アフィニティグループを入力します。URLは、各自適切な名前を付けてください。ここでは、”*[IoTCloud]*”としておきます。URLの項目に入力した値は、以降、“AccountName”という名前の項目で使います。入力したら（5）“ストレージアカウントの作成”をクリックします。Azure上でストレージアカウントの作成が開始されます。

以上で、ストレージアカウントの作成作業は完了です。

Azure管理ポータル上で作成したストレージが“オンライン”になったら、作成したストレージアカウントを選択し、下の帯の“アクセスキーの管理”をクリックします。表示されたダイアログの”プライマリーアクセスキー“をコピーしておきます。AccountNameとこのアクセスキーは次の手順で使用します。

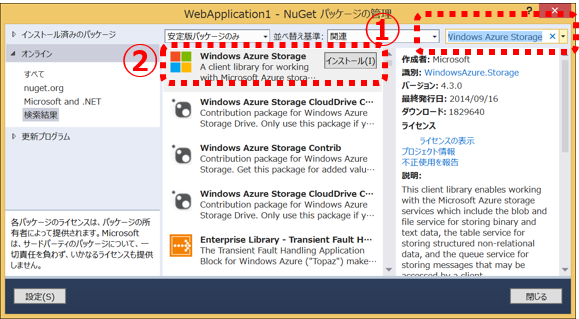
## 受信データのテーブルへの蓄積

前のステップで作成したクラウドストレージのテーブルにWebアプリで受信したセンサー計測データを蓄積します。Step 2計測の2.1で使ったWeb アプリを修正していきます。まずは、Visual Studioで、Webアプリプロジェクトを開きます。

Webアプリ内でのクラウドストレージへのアクセスは、Windows Azure Storage SDKを使用します。ソリューションエクスプローラで、“参照”を右クリックし、“NuGetパッケージの管理”を選択します。

ダイアログの右上の検索窓に“Windows Azure Storage”と入力（①）します。

※表示されなかったら“WindowsAzure Storage”で検索してみてください。



“Windows Azure Storage”が表示されたら“インストール”をクリックします。これで、プロジェクトにWindows Azure Storage SDKが組み込まれました。

※検索できなかったら、メニューの“ツール”→“NuGetパッケージマネージャー”→“パッケージマネージャーコンソール”を選択して、

Install-Package WindowsAzure.Storage -Pre

と入力してください。

次に、Step 2計測の2-1で作成したSensorController.csファイルを修正します。

まず、SensorController.csのSensorControllerクラスの次に、SensorReadingというクラスを追加します。

public class SensorReading : TableEntity

{

public SensorReading(string deviceId, string sensorType, DateTime time)

{

PartitionKey = deviceId;

RowKey = sensorType + time.Ticks.ToString();

DeviceId = deviceId;

SensorType = sensorType;

UploadTime = time;

}

public string DeviceId { get; set; }

public string SensorType { get; set; }

public string SensorValue { get; set; }

public DateTime UploadTime { get; set; }

}

このクラスは、Azureのクラウドストレージのテーブルのデータ構造を規定するものです。テーブルは、デフォルトで、PartitinKey、RowKeyというカラムを持っています。それに、DeviceId、SensorType、SensorValue、UploadTimeとうカラムを追加します。クラウドストレージのテーブル名は、“SensorReading”とします。

次に、SensorContollorクラスのGet（）メソッドを変更します。

public **string** Get()

{

**var storeCS = CloudConfigurationManager.GetSetting("StorageConnectionString");**

**var storageAccount = CloudStorageAccount.Parse(storeCS);**

**var tableClient = storageAccount.CreateCloudTableClient();**

**var sensorReadingTable = tableClient.GetTableReference("SensorReading");**

**await sensorReadingTable.CreateIfNotExistsAsync().Wait();**

var now = DateTime.Now;

var header = this.Request.Headers;

**var sensorReading = new SensorReading(header.GetValues("device-id").ElementAt(0), header.GetValues("sensor-type").ElementAt(0), now)**

**{**

**SensorType = header.GetValues("sensor-type").ElementAt(0),**

**SensorValue = header.GetValues("sensor-value").ElementAt(0)**

**};**

**await sensorReadingTable.ExecuteAsync(TableOperation.Insert(sensorReading)).Wait();**

**this.StatusCode(HttpStatusCode.OK);**

**return "stored";**

}

コードは、

1. CloudConfigurationManagerクラスのGetSetting（）メソッドで、クラウドストレージに接続するための接続文字列を取り出す
2. CloudStorageAccountクラスを使って、接続文字列からアカウントアクセスインスタンスを作成
3. CreateCloudTableClient（）メソッドをコールして、テーブルにアクセスするためのクライアントを作成
4. GetTableReference（）メソッドをコールして、“SensorReading”テーブルにアクセスするための参照を作成
5. CreateIfNotExistsAsync（）メソッドをコールして、テーブルが無い場合は作成する

という流れで、テーブルにアクセスするための準備を行います。後は、先ほど追加したSensorReadingクラスのインスタンスを作成し、最後のExecuteAsync（）メソッドをコールしてテーブルにストアする、これで、送られたデータのストレージへの蓄積に関するコーディングは終了です。

最後に、Web.configファイルにクラウドストレージへの接続文字列を追加します。Step 2で実習したEvent Hub向けのSDKとは異なり、Windows Azure Storage SDKのインストールでは、Web.configファイルへの、StorageConnectionStringの登録項目の自動追加は行われません。1.2のストレージアカウント作成時にURLに入力した文字列を*[AccountName]*に、コピーしておいたプライマリーキーを*[AccountKey]*に設定します。<appSettings>…</appSettings>の中に、以下の定義を追加します。

<add key="StorageConnectionString" value="DefaultEndpointsProtocol=https;AccountName=*[AccountName]*;AccountKey=*[AccountKey]*" />

全ての修正が終わったらAzureに発行し、リモートデバッグで接続し、IoT Kitも動かして、動作を確認してください。

受信データを保存するSensorReadingテーブルの中身を確認したい場合は、サーバーエクスプローラで、“Azure”→“ストレージ”→“AccountName”→… とずっと展開していくと、”テーブル“という項目の下にSensorReadingテーブルがあり、それをクリックすると、蓄積されたデータを確認できます。

|  |
| --- |
| 1. Event Hubで受信したデータを蓄積 |
| このSTEPでは、Event Hubで受信したデータを蓄積する方法を実習します。   * **IoT Kit側の変更** * **Event Hubで受信** |

## IoT Kit 側の修正

Step 2計測編2.2.の学習で作成したプログラムを修正します。修正は本Stepの1.1で行ったものと同等です。

void uploadTimer\_Tick(GT.Timer timer)

{

uploadTimer.Stop();

var message = new Message(System.Text.UTF8Encoding.UTF8.GetBytes("Sending Sensor"));

message.ApplicationProperties = new ApplicationProperties();

message.ApplicationProperties["device-id"] = deviceId.ToString();

lock (this)

{

message.ApplicationProperties["sensor-type"] = "Temperature";

message.ApplicationProperties["sensor-value"] = lastTemperature;

}

amqpSender.Send(message);

uploadTimer.Start();

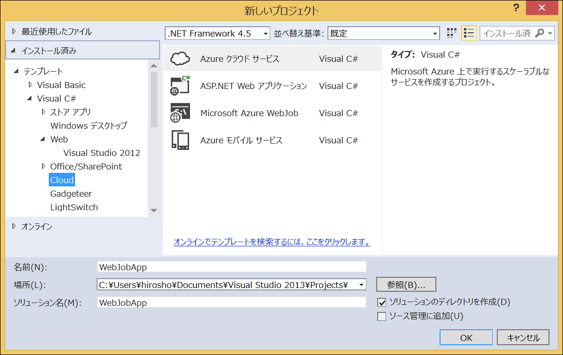
}

以上で終わりです。

## Event Hubで受信

Step 2計測編2.2.でEvent Hubに計測データを送信する方法を学習しましたが、ここでは、Event Hubに受信しているデータをWeb Jobという仕組みを使ってテーブルに蓄積する方法を学習します。

Visual Studioで新規プロジェクトを作成します。使用するプロジェクトテンプレートは、C#、Cloudの“Microsoft Azure WebJob”です。名前は“WebJobApp”としておきます。WebJobは、AzureのWeb Sitesの一機能で、バックエンドでの処理や定期的なバッチ処理などを行う用途で利用する仕組みであり、Event Hubに送られてくるたびにバックエンドで動くWebサービスを開発するのにうってつけです。



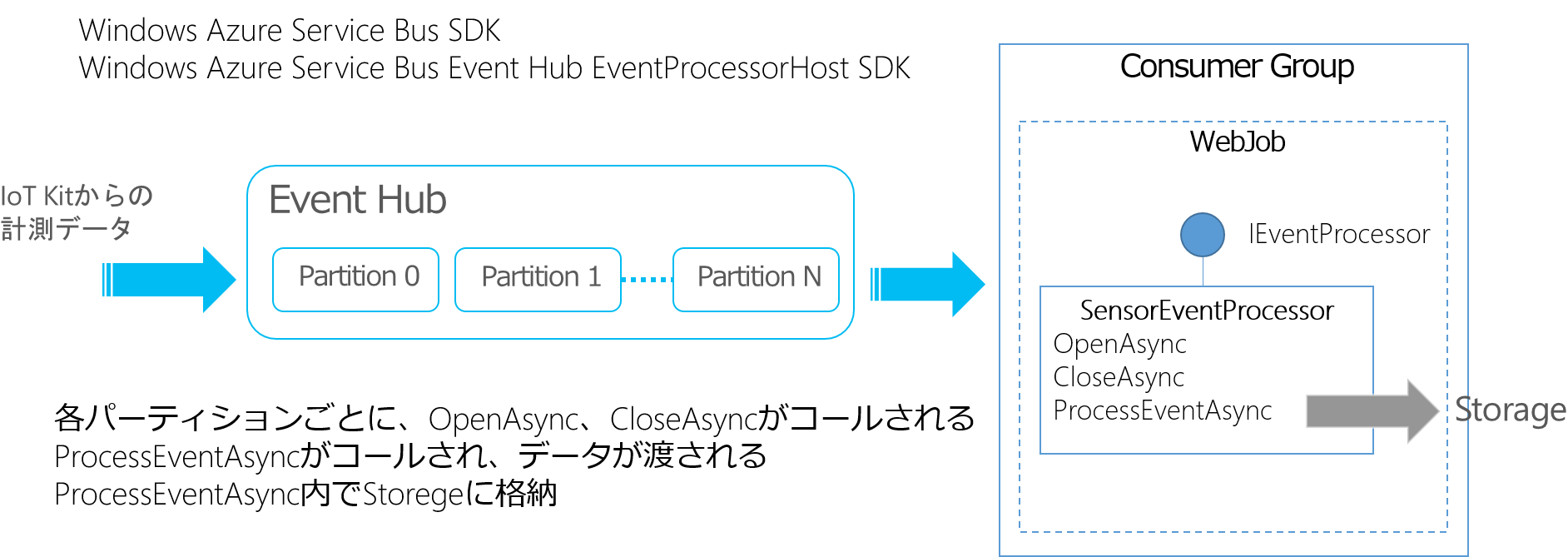
出来上がったプロジェクトに、NuGetで、

* “Windows Azure Service Bus SDK”
* “Windows Azure Service Bus Event Hub – Event Processor Host SDK”

の二つのSDKを追加します。2番目のSDKは、Event Hubにデータが受信されるたびにコールバックを受ける仕組みを作成するためのものです。

“Microsoft Azure WebJob”で作成されたプロジェクトは極めてそっけないものです。

Main（）メソッドがあるだけのProgramクラスが自動的に用意されます。これから手を加えていく手順はちょっと複雑なので、図を示しておきます。



MVAのオンラインコンテンツでの確認もしてください。

まずは、図に示しているIEventProcessorを実装するSensorEventProcessorを作ります。ソリューションエクスプローラでプロジェクトを右クリックし、“追加”→“クラス”を選択し、“SensorEventProcessor.cs”を追加します。追加したファイルの中身は以下のようなコーディングを行います。

public class StoreEventProcessor : IEventProcessor

{

Task IEventProcessor.CloseAsync(PartitionContext context, CloseReason reason)

{

return Task.FromResult<object>(null);

}

CloudTable sensorReadingTable;

async Task IEventProcessor.OpenAsync(PartitionContext context)

{

var storageAccount = CloudStorageAccount.Parse(CloudConfigurationManager.GetSetting("StorageConnectionString"));

var tableClient = storageAccount.CreateCloudTableClient();

sensorReadingTable = tableClient.GetTableReference("SensorReading" + context.Lease.PartitionId);

await sensorReadingTable.CreateIfNotExistsAsync();

}

async Task IEventProcessor.ProcessEventsAsync(PartitionContext context, IEnumerable<EventData> messages)

{

foreach (var message in messages)

{

var sensorReading = new SensorReading(context.EventHubPath, message.Offset);

sensorReading.EventHubPartitionKey = message.PartitionKey;

sensorReading.EnqueudTime = message.EnqueuedTimeUtc;

foreach (var propKey in message.Properties.Keys)

{

switch (propKey)

{

case "device-id":

sensorReading.DeviceId = (string)message.Properties[propKey];

break;

case "sensor-type":

sensorReading.SensorType = (string)message.Properties[propKey];

break;

case "sensor-value":

sensorReading.SensorValue = (string)message.Properties[propKey];

break;

}

}

await sensorReadingTable.ExecuteAsync(TableOperation.Insert(sensorReading));

}

}

}

public class SensorReading : TableEntity

{

public SensorReading(string eventHubName, string offset)

{

this.PartitionKey = eventHubName;

this.RowKey = eventHubName + "\_" + offset;

}

public string EventHubPartitionKey { get; set; }

public string DeviceId { get; set; }

public string SensorType { get; set; }

public string SensorValue { get; set; }

public DateTime EnqueudTime { get; set; }

}

後半のSensorReadingクラスは、Azureストレージのテーブルの定義です。Event Hubの特性を鑑み、1.3から若干プロパティ（テーブルのカラム）を追加しています。SensorEventProcessorクラスは、IEventProcessorで定義されたOpenAsync（）、CloseAsync()、ProcessEventAsync()の三つのメソッドを実装しています。SensorEventProcessorクラスをひな形に、インスタンスが各パーティションごとに作成され、それぞれのパーティションごとに、処理の開始時にOpenAsync()メソッドがコールされ、メッセージをEvent Hubが受信するたびにProcessEventAsync()メソッドがコールされ、更に、処理の終了時にCloseAsync()メソッドがコールされます。

OpenAsync()メソッドでは、引数で受け取ったパーティション情報を基に、そのパーティションで受信されるセンサー計測データを格納するテーブルを作成しています。作成したテーブルは、CloudTableクラスのsensorReadingTableに参照を保持しておきます。

CloseAsync()メソッドでは、このトレーニングでは特に何もすることはないので特に何も実装はしていません。実際のシステムでは、関連するサービスへの蓄積処理の停止通知などを実装することになります。

EventProcessorAsync（）メソッドでは、受信したメッセージからセンサー計測データに関する情報を取り出し、OpenAsync()メソッドで作成したsensorReadingTableにデータを格納しています。IEventProcessorのEventProcessorAsync()メソッドの定義上、複数のメッセージを一括して受け取る仕様になっているので、foreach(var message in message)…で受信した全てのデータをsensorReadingTableに格納します。

ここまで来ると、例えば、受信したセンサー計測値のトレンド分析等したい場合、このメソッド内で処理したい誘惑にかられますが、ここで作成しているSensorEventProcessorクラスは、受信データの蓄積を目的としたクラスなので、保守性や拡張性の観点から、その役割に専念することとし、トレンド分析等は、別途IEventProcessorインターフェイスを実装するクラスを作成し、そのEventProcessorAsync()で処理を行うよう設計するのが、ベターです。

次に、このSensorEventProcessorクラスをWeb Jobの処理に組込みます。

Program.csファイルを開き、以下の修正を加えます。

class Program

{

static void Main()

{

**var program = new Program();**

**program.SetupStoring().Wait();**

**Thread.Sleep(Timeout.Infinite);**

}

**const string eventHubName = "*[EventHubName]*";**

**const string eventHubNamespace = "*[EventHubNamespace]*";**

**const string storeConsumerGroupName = "storing";**

**async Task SetupStoring()**

**{**

**var storeCS = CloudConfigurationManager.GetSetting("StorageConnectionString");**

**var storageAccount = CloudStorageAccount.Parse(storeCS);**

**var eventHubCS = CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");**

**var ehClient = EventHubClient.CreateFromConnectionString(eventHubCS, eventHubName);**

**var nsManager = NamespaceManager.CreateFromConnectionString(eventHubCS);**

**var ehDescrip = await nsManager.GetEventHubAsync(eventHubName);**

**var storeCGDescrip = await nsManager.CreateConsumerGroupIfNotExistsAsync(eventHubName, storeConsumerGroupName);**

**var storeCG = ehClient.GetConsumerGroup(storeConsumerGroupName);**

**var epHost = new EventProcessorHost(eventHubNamespace, eventHubName, storeConsumerGroupName, eventHubCS, storeCS);**

**await epHost.RegisterEventProcessorAsync<SensorEventProcessor>();**

**}**

}

Main()メソッドでは、SetupStoring()をコールし、Asyncメソッドであるこのメソッドの処理完了を待ちます。SetupStoring()メソッドのコードは、蓄積用のConsumerGroupの作成やIEventProcessorを実装したクラスの登録等を行った後、処理が戻ってきて、そのままでは、大元の実行プロセスが終わってしまうので、最後のThread.Sleep(…)メソッドでプロセスが終わらないようにしています。

SetupStoring()メソッドでは、Step2計測の3.2で学習したConsumerGroupの作成を行っています。ここでは、センサー計測データ蓄積用のConsumerGroupとして“storing”という名前で作成しています。そして最後の二行で、SenosorEventProcessorのクラス処理を登録しています。この登録処理により、SensorEventProcessorクラスに実装されたOpenAsync()メソッドやEventProcessorAsync()メソッドがコールされるようになります。

コードの中の、*[EventHubName]*や*[EventHubNamespace]*は、Step1接続の学習で作成したEvent Hubに合わせて変えてください。

この節で紹介してきたコードは、StorageConnectionStringやMicrosoft.ServiceBus.ConnectionStringを参照しています。これらの定義は、SDKをインストールしたときに、app.configファイルに追加されているので、Step1接続やこのステップの1.3で説明したのと同じ方法で、各自のサービス定義に合わせて修正してください。

以上でコーディングは完了です。作業が終わったら、Azure上にWebJobプロジェクトをパブリッシュします。ソリューションエクスプローラで、プロジェクトアイコンを右クリックし、“Azure WebJobとして発行する”を選択して、発行処理を行います。

WebJobプロジェクトも、リモート接続してデバッグできるので、リモート接続デバックを開始し、一つ前の節で改変したIoT Kitのアプリケーションを実行して、処理の実際の実行状況を確認してください。

Event Hubのパーティションの数だけ、OpenAsyn()メソッドがコールされ面食らうかもしれませんが、マルチスレッドプログラミングや、きちんとモデル化されたオブジェクト指向で設計されたプログラムのデバッグのよいトレーニングになるので、頑張ってみてください。はじめのうち、OpenAsync()、CloseAsync()、EventProcessorAsync()でブレークポイントを設定してステップ実行し、各行の実行状況を確認したら、徐々にブレークポイントを外していくのがよいでしょう。

しばらくIoT Kitからセンサー計測データを送信し続けた後、サービスエクスプローラで、Azureストレージの状態を確認してください。パーティションの数だけテーブルが作成され、センサー計測データが送られたパーティションに対応するテーブルにのみ、計測データが格納されていることが確認できれば、このステップは完了です。

|  |
| --- |
| 1. 試してみよう |
| このステップの学習を完了すると、組込み機器とクラウドの接続、センサー計測、計測データのクラウドへの送信、クラウドからIoT Kitへのコマンド受信、クラウドに送信された計測データの永続ストレージへの蓄積ができるようになりました。  計測の時間間隔や、アップロードの頻度、クラウドに送信するデータ構成など、色々と考えて、拡張してみてください。  次のステップでは、送信された、あるいは、蓄積されたセンサー計測データの活用方法を学びます。 |