

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



IoT Kit Hands-on  
トレーニング

Step 4：表示編

改定: 2014年 12月 30日

初版: 2014年 12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](#_Toc407633832)

[0.1 はじめに 5](#_Toc407633833)

[0.2 学習内容 6](#_Toc407633834)

[1. Webアプリによる表示 8](#_Toc407633835)

[1.1 IoT Kitの送信コードを修正 9](#_Toc407633836)

[1.2 受信Webアプリのコード修正 11](#_Toc407633837)

[1.3 ストレージに蓄積されたデータを表示するWebアプリ作成 12](#_Toc407633838)

[2. Mobile Serviceによるモバイル端末連携 16](#_Toc407633839)

[2.1 Mobile Serviceの作成 17](#_Toc407633840)

[2.2 Windows ストアアプリによる表示アプリ開発 22](#_Toc407633841)

[2.3 WebアプリからMobile Serviceへのデータ転送 26](#_Toc407633842)

[2.4 表示用Webアプリの開発 27](#_Toc407633843)

[3. Notification Hubによるアプリ通知 31](#_Toc407633844)

[3.1 Mobile Service へのPush通知設定 32](#_Toc407633845)

[3.2 WindowsストアアプリへのPush通知ロジック追加 35](#_Toc407633846)

[3.3 Mobile ServiceへのPush通知送信ロジック追加 37](#_Toc407633847)

[4. SignalRによるリアルタイム通信 38](#_Toc407633848)

[4.1 SignalRハブの作成 39](#_Toc407633849)

[4.2 WebアプリへのSignalR送信ロジック追加 42](#_Toc407633850)

[4.3 表示WebアプリでのSignalR受信ロジック追加 43](#_Toc407633851)

[4.4 ストアアプリでのSignalR受信ロジック追加 45](#_Toc407633852)

[5. 試してみよう 50](#_Toc407633853)

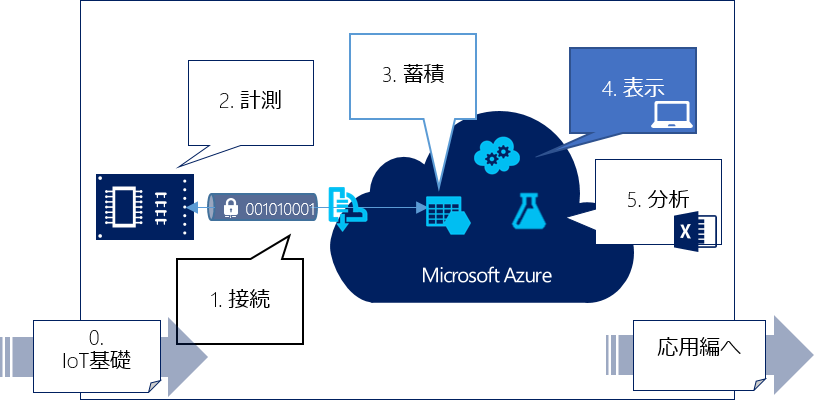
|  |
| --- |
| 1. はじめに |
|  |

## 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四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、組込み機器のセンサーで計測し、クラウドに蓄積したデータを基に、表示をはじめとするユーザーフロントエンドのパーツを作成する方法を学びます。

****

実習は以下の順番に従って行います。

1. Webアプリによる表示
   1. IoT Kit の送信コードを修正
   2. 受信Webアプリのコード修正
   3. ストレージに蓄積されたデータを表示するWebアプリ開発
2. Mobile Serviceによるモバイルアプリ連携
   1. Mobile Serviceの作成
   2. Windows ストアアプリによる表示アプリ開発
   3. WebアプリからMobile Serviceへのデータ転送
   4. 表示用Webアプリの開発
3. Notification Hubによるアプリへの通知
   1. Mobile ServiceへのPush通知設定
   2. WindowsストアアプリのPush通知設定
   3. Mobile ServiceへのPush通知送信ロジック追加
4. SignalRによるリアルタイム通信
   1. SignalRハブの作成
   2. Webアプリへのデータ受信ロジック追加
   3. Windows ストアアプリへのデータ受信ロジック追加と表示追加
5. 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。

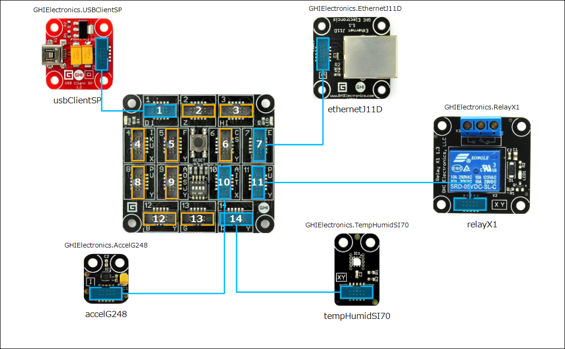
2014年12月末日現在、標準のIoT Kit ハードウェアは、GHI Electoronics 社製のFEZ Spider、及び、各種センサーを使ったFEZ版IoT Kitが販売されています。現行の手順書ではこのキットを使った手順を解説しています。GR PEACH版IoT Kitハードウェアの準備が整った段階で、改定の予定です。学習される皆様にはご迷惑をおかけしますが、ご了承ください。

|  |
| --- |
| 1. Webアプリによる表示 |
| 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** * **受信Webアプリのコード修正** * **ストレージに蓄積されたデータを表示するWebアプリ開発** |

## IoT Kitの送信コードを修正

Step 3蓄積編1.1.の学習で手を入れたプログラムを修正します。

FEZ IoT Kitの配線は、図のようにします。



リレーの接続を忘れずに。

修正は、Program.cs のmeasureTimer\_tick()メソッドと、uploadTimer\_tick()メソッドに行います。IoT Kitのデバイス状態を保持するdeviceStatusというメンバー変数も追加します。

**string deviceStatus = "";**

void measureTimer\_Tick(GT.Timer timer)

{

measureTimer.Stop();

var th = tempHumidSI70.TakeMeasurement();

var accel = accelG248.GetAcceleration();

lock (this)

{

lastTemperature = th.Temperature;

lastAccelX = accel.X;

lastAccelY = accel.Y;

lastAccelZ = accel.Z;

**if (lastTemperature > 30)**

**{**

**deviceStatus = "hot";**

**relayX1.TurnOn();**

**}**

**else**

**{**

**deviceStatus = "cool";**

**relayX1.TurnOff();**

**}**

}

Debug.Print("Temperature=" + lastTemperature);

measureTimer.Start();

}

30度を超えたらリレーをONにして状態を“hot”に、30度未満になったらリレーをOFFにして状態を”cool”にしています。

void uploadTimer\_Tick(GT.Timer timer)

{

uploadTimer.Stop();

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

**request.Headers.Add("device-status", deviceStatus);**

}

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();

}

ここでは、デバイスの状態を送信するコードを追加しています。

## 受信Webアプリのコード修正

Step 3 蓄積編の1.3で作成した“*IoTCloud*”プロジェクトのWebアプリを修正します。

先ずは、テーブルにデバイスの状態を蓄積するカラムを追加します。

ControllersフォルダーのSensorController.csファイルのSensorReadingクラスを以下のように変更します。

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 string DeviceStatus { get; set; }**

public DateTime UploadTime { get; set; }

}

デバイスの状態を蓄積するDeviceStatusというプロパティを追加しています。Azureのテーブルはフレキシブルでプロパティの増減は自由にできます。前のステップで作成したテーブルの再作成等の作業は必要ありません。

次に、REST APIの受け口であるGet()メソッドを修正します。

public async void Get()

{

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

var storageAccount = CloudStorageAccount.Parse(storeCS);

var tableClient = storageAccount.CreateCloudTableClient();

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

await sensorReadingTable.CreateIfNotExistsAsync();

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),

**DeviceStatus=header.GetValues("device-status").ElementAt(0)**

};

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

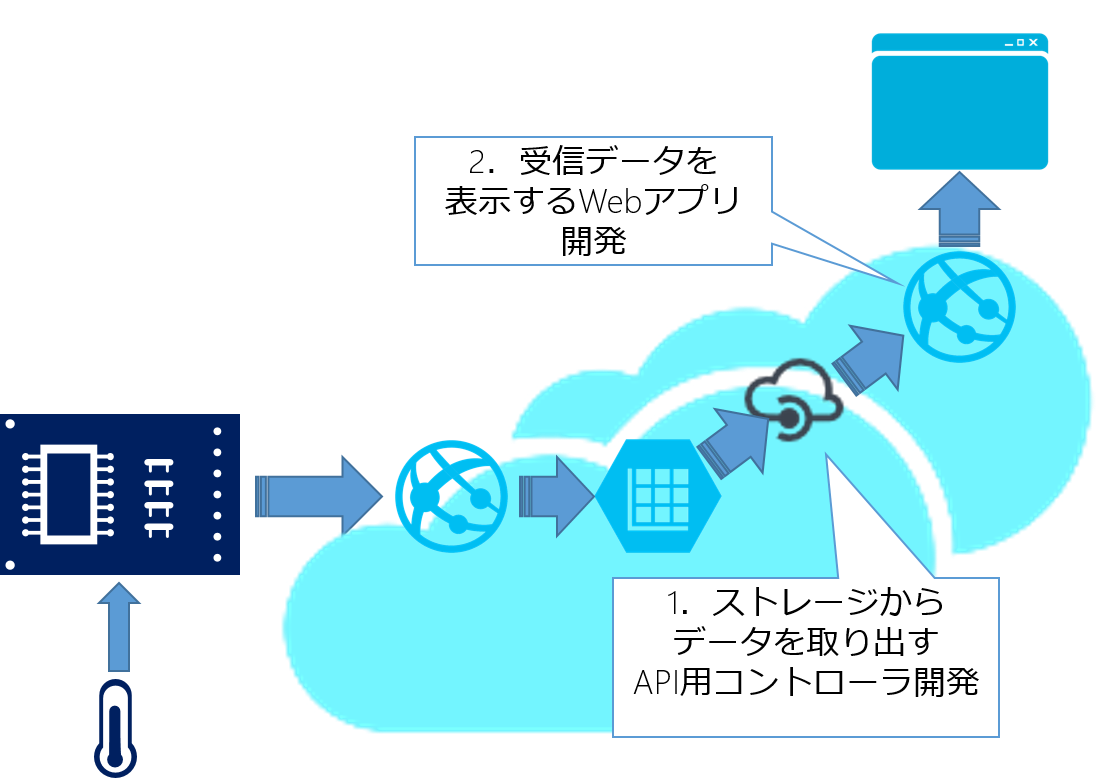
}

ヘッダーに格納されて送られてきたデバイス状態を取り出して、テーブルのプロパティに登録しています。

## ストレージに蓄積されたデータを表示するWebアプリ作成

Step 3 蓄積編の1.3で作成したWebアプリプロジェクトIoTCloudに表示用の機能を追加します。

ここでは、ASP.NETアプリプロジェクトに図のように二つの機能を加えます。



### ストレージからデータを取り出すAPI用コントローラー開発

Azureのテーブルストレージから、IoT Kitハードウェアから送信され蓄積されたデータを、HTTP REST形式で取り出すコントローラーを作成します。次のステップで作成する表示用Webアプリは、このコントローラーの機能を通じて蓄積データを表示します。

Visual Studioで、“*IoTCloud*”を開きます。

先ず、蓄積データを受け渡すためのデータクラスを“Models”フォルダーの下に作成します。

ソリューションエクスプローラーで“Models”フォルダーを右クリックし、コンテキストメニューで“追加”→“クラス”を選択します。“SensorReading”というクラス名で以下の様にコーディングします。

public class SensorReading:TableEntity

{

public string DeviceId { get; set; }

public string SensorType { get; set; }

public string SensorValue { get; set; }

public string DeviceStatus { get; set; }

public DateTime UploadTime { get; set; }

}

次に、“Controllers”フォルダーを右クリックし、コンテキストメニューで“追加”→“コントローラー”を選択します。名前は、“StorageAccessController”とします。

出来上がった“StorageAccessController”クラスに、以下のメソッドを追加します。

public IEnumerable<Models.SensorReading> Get()

{

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

var storageAccount = CloudStorageAccount.Parse(storeCS);

var tableClient = storageAccount.CreateCloudTableClient();

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

var query = new TableQuery<Models.SensorReading>().Where(

TableQuery.GenerateFilterConditionForDate("UploadTime",  
 QueryComparisons.GreaterThanOrEqual, DateTimeOffset.Now.AddMonths(-1))

);

var results = sensorReadingTable.ExecuteQuery(query).  
Select((ent => (Models.SensorReading)ent)).ToList();

return results;

}

※using宣言等は適宜解決してください。

ここでは、Step3蓄積編で使用したAzure Storage SDKの機能を使って、“SensorReading”テーブルから蓄積データを取り出し、Models.SensorReadingのリストを返します。このGetメソッドは、

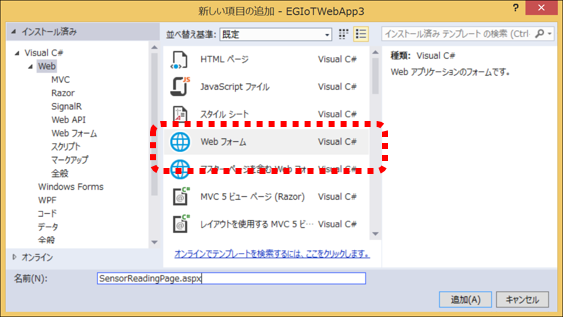
[*IoTCloud*のURL]/api/StorageAccess

というパスで、HTTPのGETバーブで利用可能です。同じWebサイトのアプリからは、“/api/StorageAccess” というURLでのアクセスが可能です。

コードの7行目からが、テーブルストレージから蓄積データを取り出す処理です。ここでは、処理を実行する日時からさかのぼって1か月間の蓄積データを取り出すように指定されています。実際のIoTシステムでは複数の組込み機器、設備機器が接続され、多くのセンサー計測値や状態が蓄積され膨大な量のデータが格納されています。そのすべてを取り出すことは実装上非常に困難です。全てのデータを一回のクエリで取り出す必要性があるシナリオは皆無なので、実際にシステムを構築する場合には適切なクエリ条件で取り出すよう考慮してください。

### 受信データを表示するWebアプリ開発

“*IoTCloud*“に表示用のページを追加します。ソリューションエクスプローラーでプロジェクトを右クリックし、コンテキストメニューで”追加“→”新しい項目”を選択します。



アイテムテンプレートの“Webフォーム”を選択し、“SensorReadingPage.aspx”という名前で作成します。

生成された“SensorReadingPage.aspx”の、<html>…</html>の部分を以下の様にコーディングします。

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

<title>IoT Kit Hands-on Step 4-1-3</title>

<script type="text/javascript" src="Scripts/jquery-1.10.2.min.js" ></script>

<script type="text/javascript">

$(function () {

var rows = 0;

var table = document.getElementById("table1");

function AddRow(pId,pStatus,pST,pSV) {

var row = table.insertRow(++rows);

var colName = row.insertCell(0);

colName.innerHTML = pId;

var colDesc = row.insertCell(1);

colDesc.innerHTML = pStatus;

var colDesc = row.insertCell(2);

colDesc.innerHTML = pST;

var colDesc = row.insertCell(3);

colDesc.innerHTML = pSV;

}

$.get("/api/StorageAccess", {},

function (result) {

// alert(result);

for (var i = 0; i < result.length; i++) {

var sr = result[i];

AddRow(sr.DeviceId, sr.DeviceStatus, sr.SensorType, sr.SensorValue);

}

});

});

</script>

</head>

<body>

<p>SensorReading</p>

<form id="form1" runat="server">

<div>

<table id="table1">

<tr><th>DeviceId</th><th>DeviceStatus</th><th>SensorType</th><th>SensorValue</th></tr>

</table>

</div>

</form>

</body>

</html>

後半の<body>…</body>で囲まれた部分は表示の基となるテーブルを定義しています。

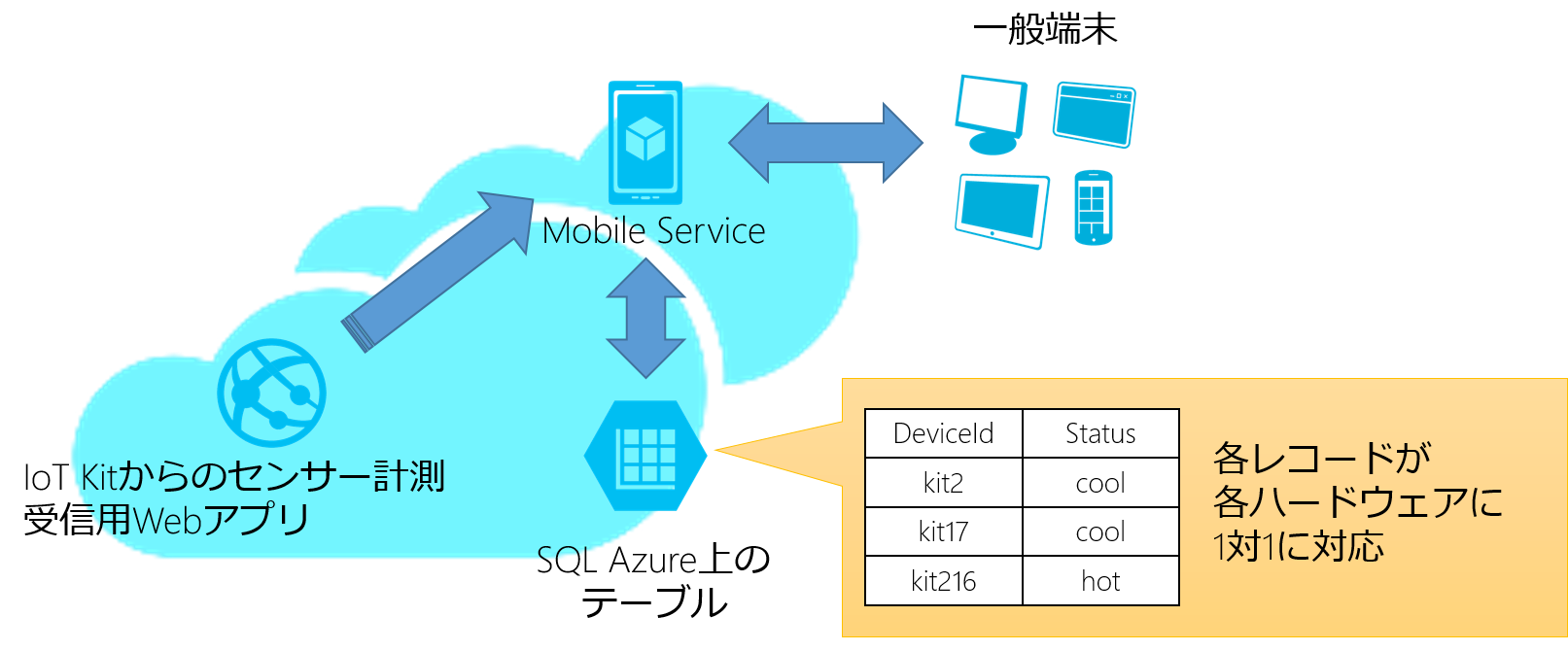
<head>…</head>の中の<script type=”text/javascript”>…</script>で囲まれた部分で、前のステップで作成したコントローラーにアクセスし一か月間蓄積されたSensorReadingのレコードを取り出して、AddRow()という<body>…</body>の<table…>…</table>に行を追加しています。HTTP REST APIへのアクセスは、$.get(…)を使っています。

以上で、作業は終わりです。“発行”でAzureのWebサイト上のWebアプリを更新し、IoT Kitハードウェアのアプリを動かしながら、“*IoTCloud*”のURLの後ろに“/SensorReadingPage.aspx”を加えて、Webブラウザで表示をしてみましょう。表示を更新するたびに、最新のセンサー計測情報が表示されます。

|  |
| --- |
| 1. Mobile Serviceによるモバイル端末連携 |
| このSTEPでは、Mobile Serviceを使ってIoT Kitとタブレットが連携するシステムの基礎を学びます。   * **Mobile Serviceの作成** * **Windows ストアアプリによる表示アプリ作成** * **Web アプリからMobile Serviceへのデータ送信** * **表示用Webアプリの開発** |

## Mobile Serviceの作成

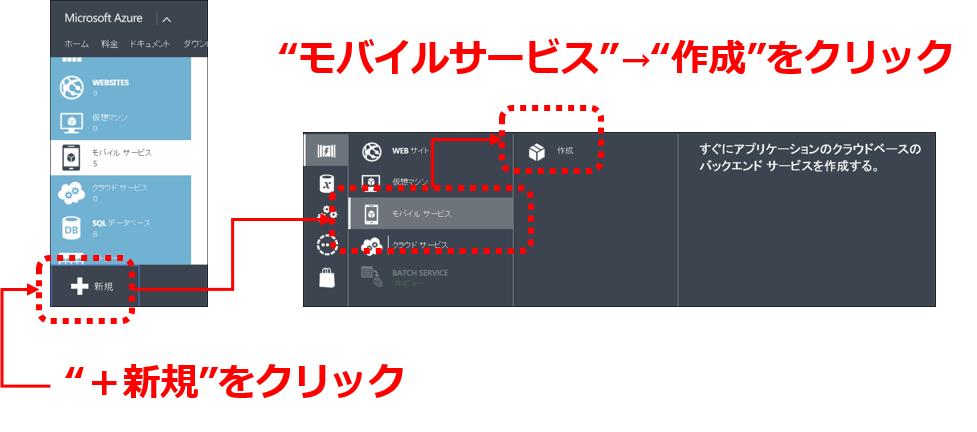
PCやタブレット、スマートフォンとクラウドを連携させるために、Microsoft Azureが提供するMobile ServiceというPaaSを使用します。各IoT Kitハードウェアの状態をMobile Serviceが管理するSQL Azure上のテーブルに格納し、モバイルアプリと連携させます。



### Mobile Serviceを作成する

Microsoft AzureのMobile Serviceを作成します。

<http://manage.windowsazure.com> を開きます。左側のタブの“モバイルサービス”を選択し、左下の“＋新規”をクリックします。“モバイルサービス”を選択し、“作成”をクリックします。



表示されたダイアログに対して、順に項目を入力、または、選択していきます。



最初の“モバイルサービスの作成”では、

* URL

インターネット上で一意の名前になるように入力。この手順書では、”*iotkitms*”としておきます。

* データベース

“新しいSQLデータベース”を選択します。

* 地域

好きなところを選んでください。

* バックエンド

“.NET”を選択してください。

次のページ、“データベースの設定の指定”では、

* サーバー

“新しいSQLデータベース”を選択してください。

* サーバーログイン名

“iotkit”としておきます。

* パスワード

任意で設定してください

* データベースの詳細設定を構成します

チェックを入れてください

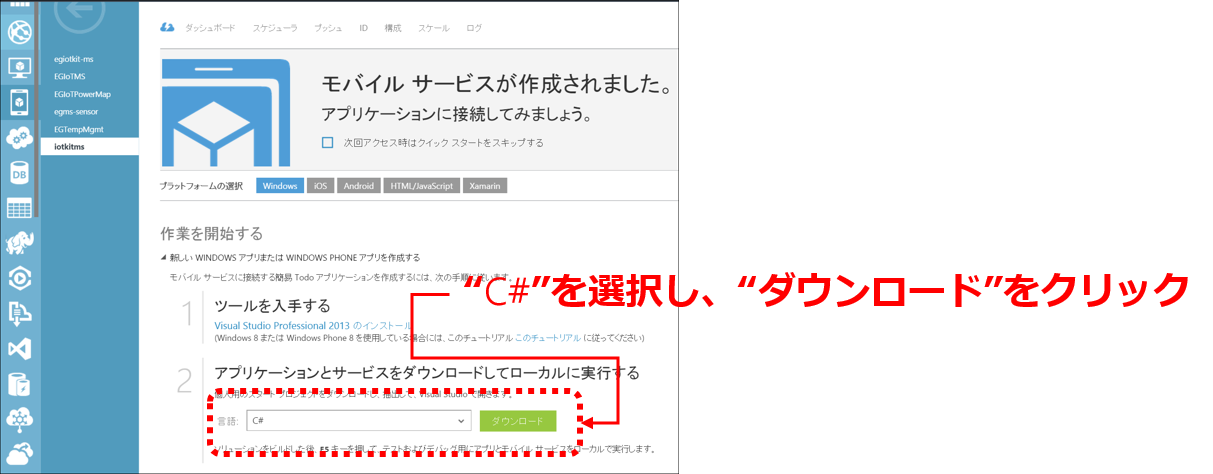
3番目の“データベースの詳細設定”は表示されたままで、右下の✓マークをクリックしてください。

これで、モバイルサービスが作成されます。作成されたら、左の“モバイルサービス”タブを選択し、表示されたリストの中から、作成された“*iotkitms*”の名前の部分をダブルクリックしてください。

表示されたページで、

“2 アプリケーションとサービスをロードしてローカルに実行する”で、言語を“C#”に選択し、“ダウンロード”ボタンをクリックします。

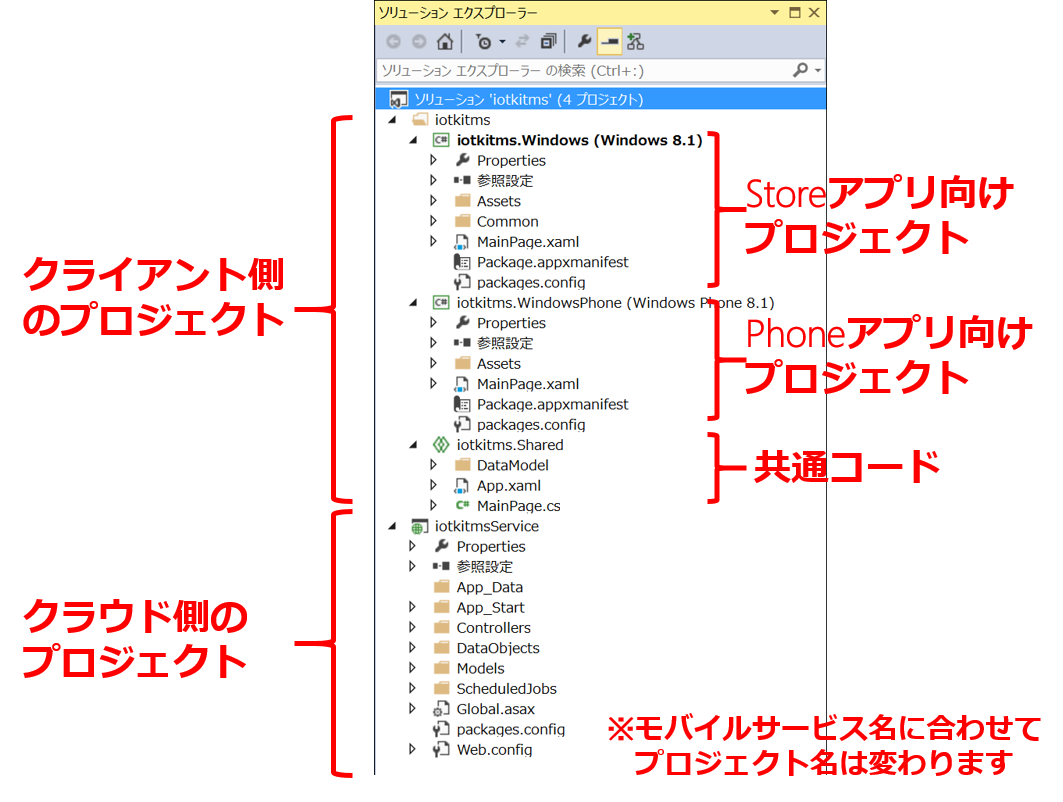
Visual Studioのソリューション一式が格納されたZIPファイルがダウンロードされるので、解凍し、SLNファイルをVisual Studioで開きます。



Visual Studioのソリューション一式が格納されたZIPファイルがダウンロードされるので、解凍し、SLNファイルをVisual Studioで開きます。

* + - ZIPファイルを展開するのを忘れないでください。ZIPファイルのままだと、ソリューションは正しく開けません。

ソリューションの構造は図のようになっています。



“*iotkitms*Service“という名前のプロジェクトがクラウド側で動くサービスの実装です。IoT Kitハードウェからセンサー計測値を受信するWebアプリが、このモバイルサービスのストレージに格納する際に利用するWeb APIはこのプロジェクトに追加します。”*iotkitms*”という名前のフォルダーに格納されているのは、モバイルクライアント側の実装です。ストアアプリでのIoT Kitハードウェアの状態表示はこちらのプロジェクトに実装していきます。作業に入る前に、一度、全体をリビルドしておいてください。

### 作成したMobile Serviceにデータモデルと転送用Web APIを追加する

“*iotkitms*Service”プロジェクトの”DataObjects”フォルダーに、“DeviceStatus.cs”というクラス向けのソースファイルを追加します。

クラスの定義は、

public class DeviceStatus : EntityData

{

public string DeviceId { get; set; }

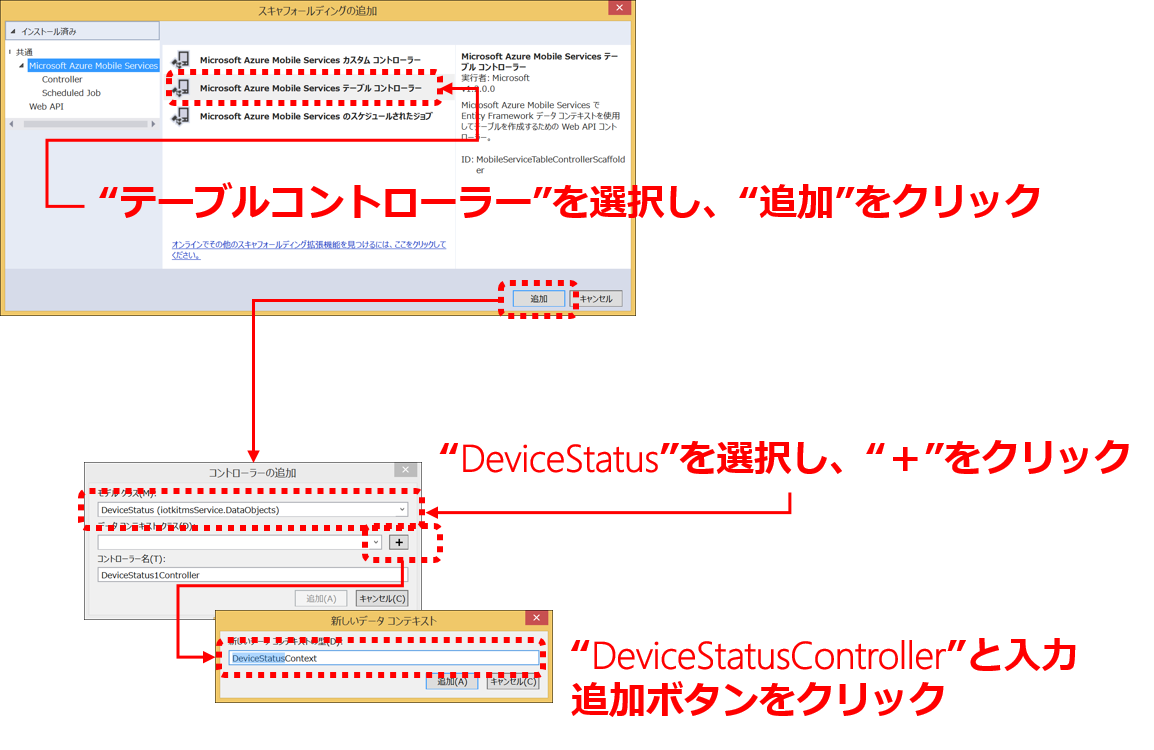
public string Status { get; set; }

}

とします。このクラスの2つのプロパティが、Mobile Serviceが管理するSQL Azure上のデータベースに用意されるテーブルのカラムになります。

※EntityDataに関連するusingの解決を行ってください。

次に、“ソリューションエクスプローラ”で、“*iotkitms*Service”プロジェクトを右クリックし、コンテキストメニューで、”追加“→”新しいスキャフォールディングアイテム“を選択します。



図の要領で順に選択、入力していきます。この一連の作業で、“Controllers”、“Models”の両ホルダーに、“DeviceStatus”で始まる、Mobile Serviceが管理するストレージへのアクセス機能を提供する二つのソースファイルが追加されます。

次に、“Controllers”フォルダーの”DeviceStatusController.cs“に、IoT Kitハードウェアからセンサー計測情報を受信するWebアプリが使用する、Device Status転送用のAPIを追加します。

“DeviceStatusController.cs”ファイルを開き、“DeviceStatusController”クラスに以下のメソッドを追加します。

public Task<DeviceStatus> PostDeviceStatus(string deviceId, string status)

{

var entities = Query().Where(d => d.DeviceId == deviceId);

if (entities.Count() > 0)

{

var entity = entities.First();

var delta = new Delta<DeviceStatus>();

entity.Status = status;

delta.CopyChangedValues(entity);

return UpdateAsync(entity.Id, delta);

}

else

{

var entity = new DeviceStatus() { DeviceId = deviceId, Status = status };

return InsertAsync(entity);

}

}

このメソッドでは、引数で受け取ったdeviceIdを基に、SQL Azure内のテーブルに既に格納済みかを確認し、格納済みであればその状態（“Status”）を更新、未格納であれば新しくレコードを追加する、という処理を行っています。

このメソッドは、ASP.NETのフレームワークによって、HTTP REST形式のAPIとしてマップされます。

以上の作業が終わったら、“ソリューションエクスプローラー”で、“*iotkitms*Service”を右クリックし、コンテキストメニューで”発行“を選択して、Azure上に配置してください。配置後、PostDeviceStatusメソッドは、

[http://*iotkitms*.azure-mobile.net/tables/DeviceStatus?deviceId=[*deviceId*]&status=[*hot*|*cool*](http://iotkitms.azure-mobile.net/tables/DeviceStatus?deviceId=%5bdeviceId%5d&status=%5bhot|cool)]

という形式で、HTTPのメソッドが“POST”で利用できるようになります。また、DeviceStatusクラスで定義されたSQL Azure上のテーブルのレコードは、

[http://*iotkitms*.azure-mobile.net/tables/DeviceStatus](http://iotkitms.azure-mobile.net/tables/DeviceStatus)

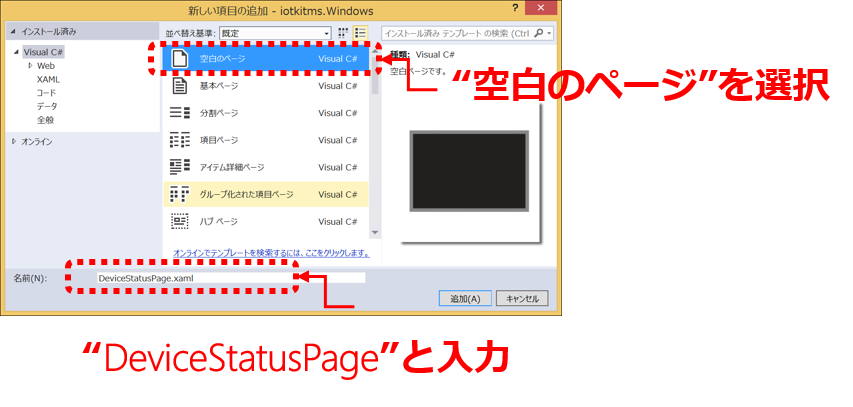
で、HTTPメソッドが“GET”で、JSON形式で取り出すことができます。

## Windows ストアアプリによる表示アプリ開発

前節の実習で、Microsoft Azureのポータルからダウンロードした、ZIPファイルに格納されているストアアプリ向けのプロジェクトをベースに、デバイス状態を表示するアプリを開発します。

※このステップを実行するには、Windows 8.1のPCが必要です。無い場合は、Microsoft Azureの“仮想マシン”で、開発環境を稼働し、試してみてください。

Visual Studioのソリューションエクスプローラーで、“iotkitms.Windows (Windows 8.1)のプロジェクトにデバイス状態を表示するためのページを追加します。プロジェクトを右クリックして、コンテキストメニューの”追加“→”新しい項目“を選択し、

図のように選択、入力し、右下の“追加”ボタンをクリックします。

出来上がったDeviceStatusPage.xamlで、IoT Kitハードウェアの状態をリスト表示するように改造します。

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<Grid.RowDefinitions>

<RowDefinition Height="180"/>

<RowDefinition/>

<RowDefinition Height="100"/>

</Grid.RowDefinitions>

<TextBlock Text="Device Status" FontSize="72" VerticalAlignment="Center" HorizontalAlignment="Center"/>

<ListView x:Name="deviceItems" Grid.Row="1" Margin="20">

<ListView.ItemTemplate>

<DataTemplate>

<Border BorderBrush="Blue" BorderThickness="1" Margin="1">

<Grid Height="80" >

<Grid.ColumnDefinitions>

<ColumnDefinition Width="200"/>

<ColumnDefinition Width="1000"/>

</Grid.ColumnDefinitions>

<TextBlock VerticalAlignment="Center" HorizontalAlignment="Center" FontSize="24" Text="{Binding DeviceId}"/>

<StackPanel Grid.Column="1" Orientation="Horizontal">

<Border BorderBrush="Blue" BorderThickness="1" Margin="1">

<Grid Width="100">

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<TextBlock Grid.Row="0" FontSize="16" Text="Status:" Margin="5,5,0,0"/>

<TextBlock Grid.Row="1" FontSize="16" Text="{Binding Status}" Margin="20,0"/>

</Grid>

</Border>

</StackPanel>

</Grid>

</Border>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

<Button x:Name="buttonRefresh" Content="Refresh" FontSize="32" Grid.Row="2" HorizontalAlignment="Center" VerticalAlignment="Center" Click="buttonRefresh\_Click"/>

</Grid>

<Grid…>…</Grid>の中身を上のリストのように書き換えます。

次に、“*iotkitms*.Shared”プロジェクトの”DataModel“フォルダーに、”DeviceStatus.cs“という名前でクラスのソースファイルを追加します。このファイルを次のように書き換えます。

namespace *iotkitms*

{

public class DeviceStatus

{

public string Id { get; set; }

[JsonProperty(PropertyName = "deviceid")]

public string DeviceId { get; set; }

[JsonProperty(PropertyName = "status")]

public string Status { get; set; }

}

}

このクラスは、Mobile Serviceからデバイスの状態リストを受信する際に使用するデータモデルです。

更に、“*iotkitms*.Shared”プロジェクトに、”DeviceStatusPage.cs“という名前でクラスのソースファイルを追加します。生成されたDeviceStatsuPageクラスを以下のように書き換えます。

sealed partial class DevicePage : Page

{

private MobileServiceCollection<DeviceStatus, DeviceStatus> devices;

private IMobileServiceTable<DeviceStatus> deviceTable = App.MobileService.GetTable<DeviceStatus>();

public DevicePage()

{

this.InitializeComponent();

}

async void RefreshDevices()

{

MobileServiceInvalidOperationException exc = null;

try {

devices = await deviceTable.ToCollectionAsync<DeviceStatus>();

}

catch (MobileServiceInvalidOperationException e)

{

exc = e;

}

if (exc != null)

{

var dialog = new MessageDialog(exc.Message, "Error loading items").ShowAsync();

}

else

{

deviceItems.ItemsSource = devices;

}

}

private void buttonRefresh\_Click(object sender, RoutedEventArgs e)

{

RefreshDevices();

}

protected override void OnNavigatedTo(Windows.UI.Xaml.Navigation.NavigationEventArgs e)

{

RefreshDevices();

base.OnNavigatedTo(e);

}

}

このクラスはPageクラスを継承し、partial宣言されています。この宣言を付与することにより、“iotkitms.Windows”プロジェクトに追加したDeviceStatusPage.xamlのコードビハインドのクラスに、機能を追加できます。Microsoft Azureのポータルサイトからダウンロードしたプロジェクト一式には、Mobile Serviceにアクセスするためのライブラリが仕込まれているので、MobileServiceCollectionやIMobileServiceTable等のAPIを利用して、簡単にクラウド上のデータをハンドリング可能です。

RefreshDevicesメソッドの中で、deviceTableのToCollectionAsyncメソッドをコールしています。ここで、クラウド上のMobile Serviceが管理しているストレージからデバイスの状態のリストをダウンロードして、devicesに格納します。その後、DeviceStatusPage.xamlで定義されたListViewのdeviceItemsのItemsResouceにdevicesを代入することにより、UIへの表示が行われます。RefreshDevicesメソッドは、DeviceStatusPage.xamlにページが遷移されたときにコールされるOnNavigatedToメソッド内でコールされるので、このページに表示が切り替わるたびに表示が更新されます。また、DeviceStatusPage.xamlで定義されたボタン<Button x:Name=”buttonRefresh” Content=”Refresh”…>…</Button>のクリックイベントに関連付けられた、buttonRefresh\_Clickメソッド内でも、RefreshDevicesメソッドはコールされます。“Refresh”という表示があるボタンをクリックしても表示の更新がなされます。

最後に、“iotkitms.Shared”プロジェクトのApp.xamlの下に従属しているApp.xaml.csを変更します。ダウンロードしてきたZIPファイルに格納されているのは、ローカルテスト向けのソースコードになっています。

27~38行目の、MobileServiceという変数をnewしている行をコメントアウトし、Mobile ServiceのURLが引数になっているコメントアウトされた行の先頭の//を削除してください。

// This MobileServiceClient has been configured to communicate with your local

// test project for debugging purposes.

**//** public static MobileServiceClient MobileService = new MobileServiceClient(

**//** "http://localhost:51795"

**//** );

// This MobileServiceClient has been configured to communicate with your Mobile Service's url

// and application key. You're all set to start working with your Mobile Service!

**public static MobileServiceClient MobileService = new MobileServiceClient(**

**"https://*iotkitms*.azure-mobile.net/",**

**"*[Access Key]*"**

);

*[Access Key]*の部分は、既に正しい文字列が宣言されているので修正せずにそのまま使ってください。

更に、デフォルトでは、“MainPage.xaml”が表示される設定になっているので、96行目の

if (rootFrame.Content == null)

{

// When the navigation stack isn't restored navigate to the first page,

// configuring the new page by passing required information as a navigation

// parameter

rootFrame.Navigate(typeof(**DevicePage**), e.Arguments); // ←ここを修正

}

Navigateメソッドのtypeofの引数を“DeviceStatusPage”に変更してください。

以上で、ストアアプリの準備は万端です。

## WebアプリからMobile Serviceへのデータ転送

これまでの実習でクラウド側のMobile Service、表示用の一般端末向けストアアプリの準備が終わりました。

次は、1.2．で改造した*IoTCloud*のWebアプリに、Mobile Serviceへのデータ転送ロジックを組込みます。

ASP.NETのWebアプリプロジェクトのSensorController.csの、SensorControllerクラスにMobile Serviceにデータを転送するロジックを持つメソッドを追加します。

private void SendDataToMS(SensorReading sensorReading)

{

var client = new HttpClient();

client.DefaultRequestHeaders.Add("X-ZUMO-APPLICATION", "*[Access Key]*");

client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

var request = new HttpRequestMessage(new HttpMethod("POST"), "https://*iotkitms*.azure-mobile.net/tables/DeviceStatus?deviceId=" + sensorReading.DeviceId + "&status=" + sensorReading.DeviceStatus);

var result = client.SendAsync(request).Result;

}

ここでは、HttpClientというライブラリを使ってMobile Serviceにデータを送信しています。[Access Key]の部分は、2.2.節のApp.xaml.csの修正で出てきたMobileServiceをnewした時の2番目の引数をコピー＆ペーストしてください。また、HttpRequestMessageコンストラクターの二番目の引数のURLの“**iotkitms**”の部分は皆さんが作成したMobile ServiceのURLに適宜変更してください。

このメソッドの引数として渡されたsensorReadingにはDeviceIdとDeviceStatusが格納されており、2.1．で追加したHTTP REST APIを使ってデータを送信しています。

SendDataToMSメソッドのコールロジックを、Getメソッドに追加します。追加する場所はメソッドの最後の行です。

public async void Get()

{

… 中略 …

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

**SendDataToMS(sensorReading);**

}

以上で、一連の作業は完了です。“発行”を実施し、クラウド上のWebサイトのアプリを忘れずに更新してください。、1.1．で変更を加えたアプリをIoT Kitハードウェア上で実行した後、2.2で作成したストアアプリをF5でデバッグ実行してください。

IoT Kitの温度センサーを触ったり放したりして、ストアアプリの“Refresh”ボタンをクリックすると、デバイスのStatus表示が変わります。

## 表示用Webアプリの開発

ここでは、2.2．で作成したWindowsストアアプリと同じように、Mobile Serviceが管理するストレージからデータを取り出して、表示するWebアプリを開発し*IoTCloud*に加えます。

前の1.3．で似たような表示用Webアプリを開発しましたが、このケースでは、Mobie Serviceがホスティングされているサーバーと、*IoTCloud*をホストするサーバーが異なるため、CORS（クロスオリジンリソース共有）の設定が必要です。

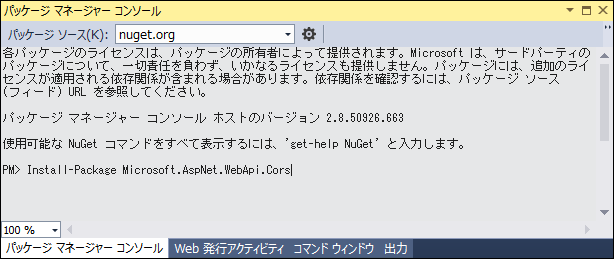
先ずは、Microsoft Azureの管理ポータルで、2.1．で作成したMobile Serviceを選択します。“構成”タブを選択し、



ページの下の方に移動して“クロスオリジンリソース共有（cors）”の”許可を与えるドメイン“の項目に*IoTCloud*をホストしているURLを入力します。

次に、表示用Webアプリを追加する*IoTCloud*側にもCORS用の修正を加えます。

メニューの“ツール”から、“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”を選択します。



上の図に示すように、“Install-Package Microsoft.AspNet.WebApi.Cors”と入力し、“Enter”を押します。

更に、“App\_Start”フォルダーの“WebApiConfig.cs”を開いて、以下の様に編集します。

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Web API の設定およびサービス

// ベアラ トークン認証のみを使用するように、Web API を設定します。

config.SuppressDefaultHostAuthentication();

config.Filters.Add(new HostAuthenticationFilter(OAuthDefaults.AuthenticationType));

// Web API ルート

config.MapHttpAttributeRoutes();

**config.EnableCors();**

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

}

MapHttpRouteメソッドをコールする行の上に、EnableCors()メソッドをコールするステートメントを追加しています。これで、こちら側でもCORSが有効になります。

参考までに、MapHttpRouteメソッドの引数で、routeTemplate: "api/{controller}/{id}"という記述があります。ASP.NETプロジェクトのControllersフォルダーに作成したコントローラクラスがなぜ、/api/… でアクセスできるかの秘密がここにあります。

準備が整ったところで、Mobile Serviceが管理するストレージからデバイスの状態を取得して表示するWebページを作成します。1.3．でやったのと同様に、Webページを“DeviceStatusPage.aspx”という名前で作成します。そして、<html>…</html>の部分を以下の様に編集します。

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

<title></title>

<script type="text/javascript" src="Scripts/jquery-1.10.2.min.js" ></script>

**<script src='http://ajax.aspnetcdn.com/ajax/mobileservices/MobileServices.Web-1.1.2.min.js'></script>**

<script type="text/javascript">

$(function () {

var xmlHttp = null;

function DeviceStatusModel() {

this.DeviceId = "";

this.DeviceStatus = "";

};

var rows = 0;

var table = document.getElementById("table1");

function AddRow(pId,pStatus) {

var row = table.insertRow(++rows);

var colName = row.insertCell(0);

colName.innerHTML = pId;

var colDesc = row.insertCell(1);

colDesc.innerHTML = pStatus;

}

AddRow("name"+rows,"desc"+rows);

**var client = new WindowsAzure.MobileServiceClient(  
"https://*[iotkitms]*.azure-mobile.net/", "*[Access Key]*");**

**var dsTable = client.getTable('DeviceStatus');**

**var read = dsTable.read();**

**read.done(function (result) {**

**for (var i = 0; i < result.length;i++) {**

**AddRow(result[i].deviceId, result[i].status);**

**}**

**}, function (error) {**

**alert(error);**

**});**

});

</script>

</head>

<body>

<p>Device Status</p>

<form id="form1" runat="server">

<div>

<table id="table1">

<tr><th>DeviceId</th><th>Status</th></tr>

</table>

</div>

</form>

</body>

</html>

構造は、1.3．の時とほぼ変わりません。異なるのはデータをMobile Serviceから取り出すところです。太字で示したのが特にMobile Serviceからデータを取り込むコードです。URLやAccess Keyはそれぞれの適切な値に変更してください。

修正が終わったASP.NETプロジェクトをAzure上に忘れずに発行して、このページをブラウザ上で開けば、Mobile Serviceが管理しているテーブル“DeviceStatus”に蓄積されたデータが表形式で表示されます。リモートデバッグも可能です。

セキュリティについて

IoTシナリオにおいて、セキュリティは、非常に大きな検討課題として考えられていることでしょう。しかし、この実習では特にセキュリティに関して明確な言及を行ってはいません。この一連のトレーニングで使用している、Mobile Serviceや、ASP.NETのWeb APIやWebForm等に対して、認証やアクセス権など、セキュリティを付与するのは簡単です。

* <http://www.asp.net/web-api/overview/security/authentication-and-authorization-in-aspnet-web-api>
* <http://msdn.microsoft.com/ja-jp/library/cc465514.aspx>
* <http://azure.microsoft.com/ja-jp/documentation/articles/mobile-services-html-get-started-users/>

もっと詳しく知りたい方は、このあたりの情報を参考にしてみてください。

|  |
| --- |
| 1. Notification Hubによるアプリ通知 |
| このSTEPでは、Mobile ServiceにNotification Hubを使ってPush通知を追加します。アプリに状態変更やアラートなどといった通知を送る方法を学びます。   * **Windows ストアアプリのPush通知設定** * **Mobile ServiceへのPush通知設定** |

## Mobile Service へのPush通知設定

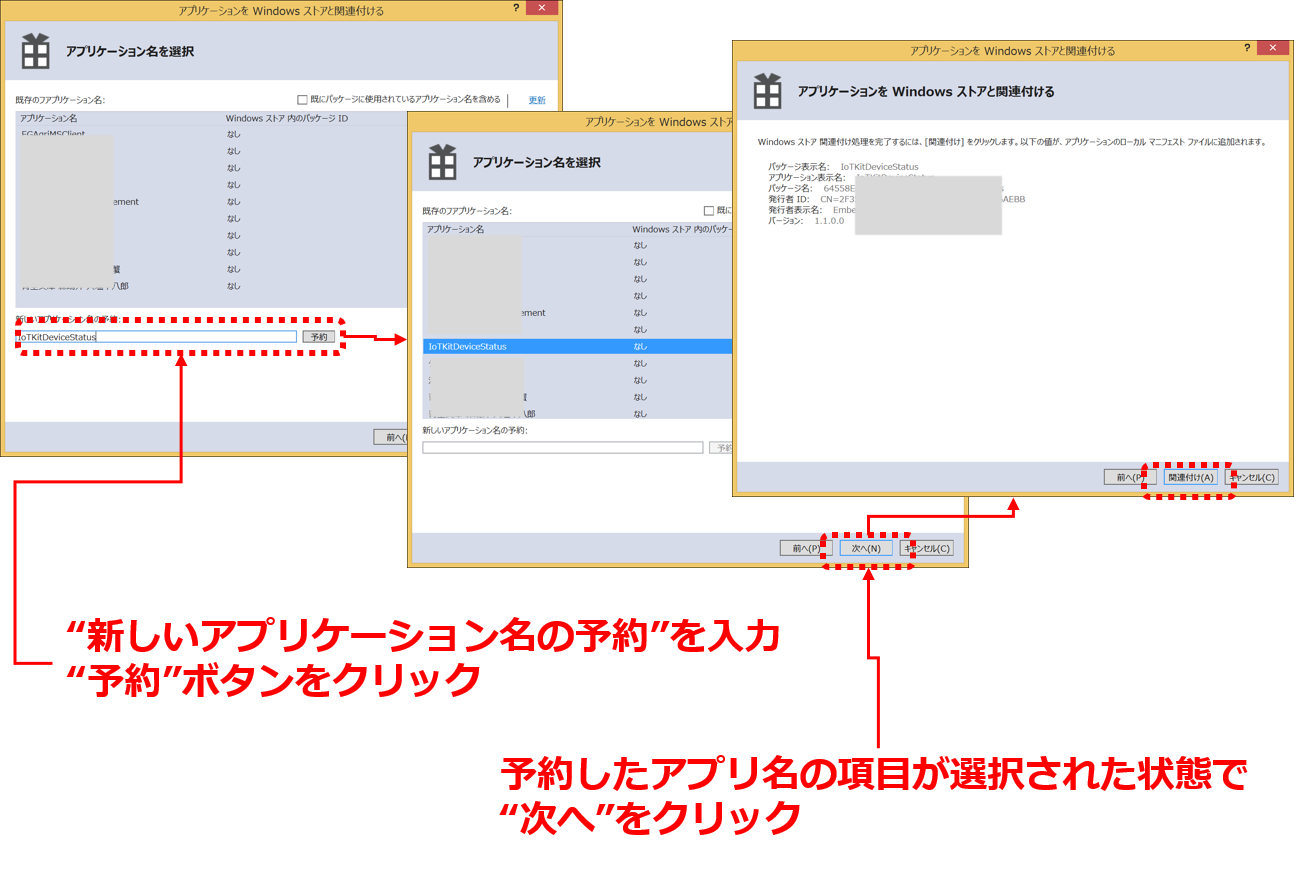
このステップでは、2．で作成したデバイス状態表示用のWindows ストアアプリが、Notification Hubの機能による、Push通知受信のための設定を行います。

※このステップを実施するには、Windowsストアアプリ開発者登録が必要です。予め、<http://dev.windows.com/ja-jp/dashboard> から、アプリ開発者登録を行ってください。

WindowsストアアプリでPush通知を受信するためには、Windowsストアの開発者ダッシュボードに関連付けを行う必要があります。

2．でMicrosoft Azureのポータルからダウンロードし、2.1．、2.2で作業を行ったソリューションをVisual Studioで開きます。ソリューションエクスプローラーで、“*iotkitms*.Windows”プロジェクトを右クリックし、コンテキストメニューで、”ストア“→”アプリケーションをストアと関連付ける“を選択します。

表示されたダイアログで一連の認証の手続きを踏んだ後、Windowsストア上のアプリ名を予約します。ここでは、“*IoTKitDeviceStatus*”という名前にしています。この名前は、Microsoft Azureのサービス名等が唯一なのと同じように、Windowsストア上で唯一でなければなりません。各自適宜適当名前を考えて使ってください。

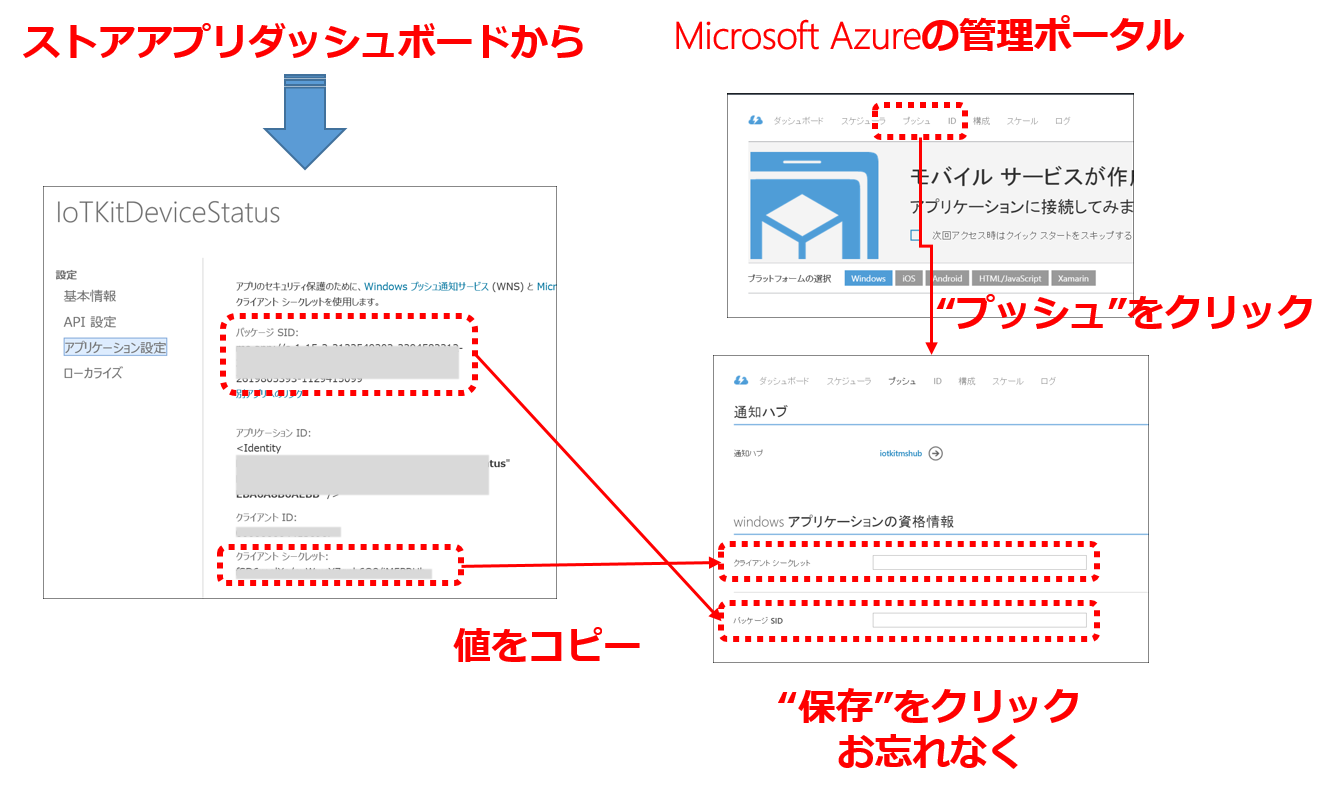


“新しいアプリケーション名の予約”に適宜名前を入力し、“予約”ボタンをクリックし、予約した名前のアプリがリスト上で選択されていることを確認して、“次へ”をクリックし、“関連付け”をクリックします。

次に、Visual Studioのメニューの”プロジェクト“→”ストア“→”開発者アカウントを開く“を選択し、ブラウザが起動されたら、表示されたページの右上隅の”サインイン“をクリックして、開発者登録したアカウントでサインインし、ダッシュボードを開きます。



予約したアプリのページから上の図に従ってページを開いていきます。“Liveサービスサイトで表示された値を下図の様に、Microsoft AzureのポータルのMobile Serviceのプッシュの設定画面にコピーします。



Microsoft Azureの管理ポータルに値をコピーした後、保存を忘れずにクリックしてください。以上で、Push通知に関する設定は完了です。

## WindowsストアアプリへのPush通知ロジック追加

Push通知に関する設定が完了したので、次は、2.2．で作成した、Windows ストアアプリにPush通知に関するロジックを追加します。Push通知を受信するには、Push通知受信用のチャネルを開き、WNS（Windows Notification Service）にチャネルのURLの登録が必要です。

2.2．で作成した”*iotkitms*.Shared”プロジェクトのApp.xaml.csのAppクラスに、Push通知を初期化するコードを追加します。

#if !WINDOWS\_PHONE\_APP

private async void InitNotificationsAsync()

{

// Request a push notification channel.

var channel = await PushNotificationChannelManager

.CreatePushNotificationChannelForApplicationAsync();

// Register for notifications using the new channel

System.Exception exception = null;

try

{

await MobileService.GetPush().RegisterNativeAsync(channel.Uri);

}

catch (System.Exception ex)

{

exception = ex;

}

if (exception != null)

{

var dialog = new MessageDialog(exception.Message, "Registering Channel URI");

dialog.Commands.Add(new UICommand("OK"));

await dialog.ShowAsync();

}

}

#endif

”*iotkitms*.Shared”プロジェクトは、Windowsストアアプリと、Windows Phoneアプリの共有ライブラリです。WNSはWindowsストアアプリ向けのみのサービスなので、WINDOWS\_PHONE\_APPマクロで、Windowsストアアプリで使われる場合のみ有効にします。

そして、OnLaunchedメソッドに、InitNotificationAsyncメソッドをコールするコードを追加します。

protected override void OnLaunched(LaunchActivatedEventArgs e)

{

・・・　中略　・・・

if (rootFrame.Content == null)

{

// When the navigation stack isn't restored navigate to the first page,

// configuring the new page by passing required information as a navigation

// parameter

rootFrame.Navigate(typeof(DeviceStatusPage), e.Arguments);

}

#if !WINDOWS\_PHONE\_APP

InitNotificationsAsync();

#endif

// Ensure the current window is active

Window.Current.Activate();

}

次に、“*iotkitms*.Windows”プロジェクトのPackage.appxmanifestを開き、”通知：“の”トースト対応“を”はい“に設定します。

これで、Windowsストアアプリ側の設定は完了です。

## Mobile ServiceへのPush通知送信ロジック追加

Push通知に関する設定が完了したので、次は、2.1．で作成した、Mobile ServiceのDeviceStatusControllerクラスのPostDeviceStatusメソッドにPush通知を送信するロジックを追加します。

public Task<DeviceStatus> PostDeviceStatus(string deviceId, string status)

{

var entities = Query().Where(d => d.DeviceId == deviceId);

if (entities.Count() > 0)

{

var entity = entities.First();

if (status != entity.Status)

{

WindowsPushMessage message = new WindowsPushMessage();

// Define the XML paylod for a WNS native toast notification

// that contains the text of the inserted item.

message.XmlPayload = @"<?xml version=""1.0"" encoding=""utf-8""?>" +

@"<toast><visual><binding template=""ToastText02"">" +

@"<text id=""1"">" + deviceId + @"</text>" +

@"<text id=""2"">Status changed to " + status + @"</text>" +

@"</binding></visual></toast>";

try

{

var result = Services.Push.SendAsync(message).Result;

Services.Log.Info(result.State.ToString());

}

catch (System.Exception ex)

{

Services.Log.Error(ex.Message, null, "Push.SendAsync Error");

}

}

var delta = new Delta<DeviceStatus>();

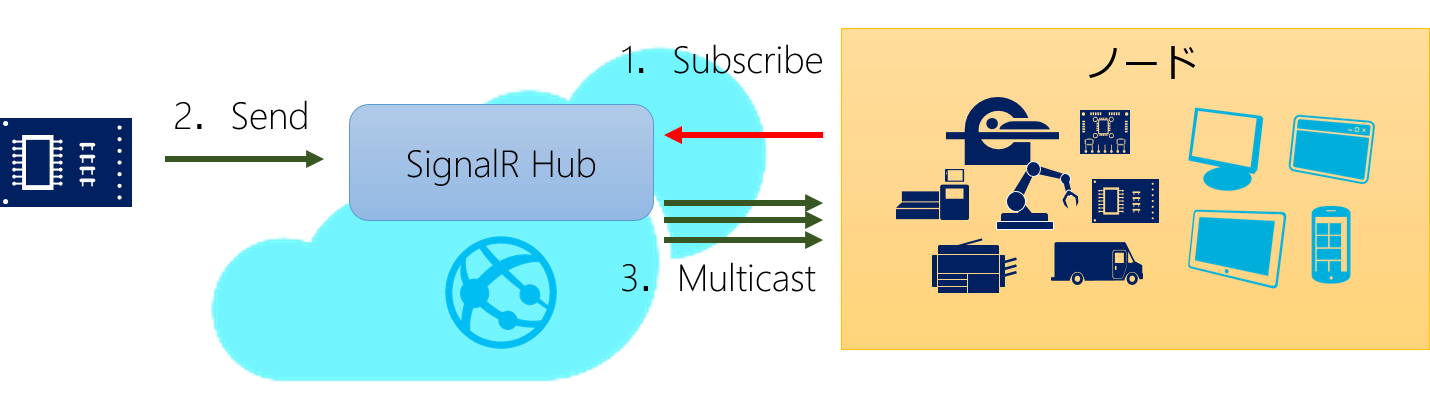
デバイスの状態が“hot”でかつ、Mobile Serviceが管理するストレージに記録されている状態が”hot”でない場合だけ、Push通知を送るロジックです。このコードではToastText02というテンプレートを使用しています。他のテンプレートについては、<http://msdn.microsoft.com/ja-jp/library/windows/apps/hh761494.aspx> を参考にしてください。

これで準備は整いました。3.2で修正したストアアプリをデバッグ実行し、一旦デバッグを終了して、IoT Kitハードウェアのアプリを実行してください。全てきちんと作られていたら、開発用PCでトーストによる通知が表示され、そのトースト通知をタップすると、ストアアプリが起動します。

|  |
| --- |
| 1. SignalRによるリアルタイム通信 |
| このSTEPでは、SignalRを使ってセンサー計測データやIoT Kitの状態をリアルタイムに送信し、Webアプリやストアアプリで受信・表示する基礎を学びます。   * **SignalRハブの作成** * **WebアプリへのSignalR送信データロジック追加** * **表示用Web アプリでのデータ受信ロジック追加** * **ストアアプリでのデータ受信ロジック追加** |

## SignalRハブの作成

SignalRは、SignalRをホストする“ハブ”に、予めサブスクライブしたノードにメッセージをマルチキャストします。



ここではまず、2.4．で改造したWebアプリに、更に改造を加えていきます。まず、SignalRでデータを送信するためのデータクラスを定義します。

ソリューションエクスプローラーで、プロジェクトの“Models”フォルダーを右クリックし、“SignalRPackets.cs”という名前でクラスを追加し、以下の様に修正します。

using Newtonsoft.Json;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace *IoTCloud*.Models

{

public class DeviceStatusPacket

{

[JsonProperty("deviceid")]

public string DeviceId { get; set; }

[JsonProperty("status")]

public string Status { get; set; }

}

public class TemperaturePacket

{

[JsonProperty("deviceid")]

public string DeviceId { get; set; }

[JsonProperty("temperature")]

public double Temperature { get; set; }

}

public class AccelerometerPacket

{

[JsonProperty("deviceid")]

public string DeviceId { get; set; }

[JsonProperty("accelx")]

public double X { get; set; }

[JsonProperty("accely")]

public double Y { get; set; }

[JsonProperty("accelz")]

public double Z { get; set; }

}

}

これらのデータクラスはSignalRの送信で使用します。

次にSignalRのハブを作成します。

ソリューションエクスプローラーで、プロジェクトを右クリックし、コンテキストメニューで“追加”→“新しいフォルダー”を選択し、“Hubs”という名前のフォルダーを作成します。

そして、作成した“Hubs”フォルダーを右クリックしてコンテキストメニューで“追加”→“SignalRハブクラス”を選択します。表示されたダイアログで“DeviceMonitorHub”という名前でハブクラスを作成します。

この操作で出来上がったDeviceMonitorHub.csのDeviceMonitorHubクラスが、SignalRのハブです。

このクラスを以下の様に改造します。

public class DeviceMonitorHub : Hub

{

public void Hello()

{

Clients.All.hello();

}

/// <summary>

/// 特定のIoT Kitハードウェアに対する通知を受けるグループにJoinする

/// </summary>

/// <param name="deviceId">通知を受けたいIoT Kitハードウェアの識別子</param>

public void Join(string deviceId)

{

Groups.Add(Context.ConnectionId, deviceId);

}

/// <summary>

/// 特定のIoT Kitハードウェアに対する通知を受けるグループから離れる

/// </summary>

/// <param name="deviceId"></param>

public void Leave(string deviceId)

{

Groups.Remove(Context.ConnectionId, deviceId);

}

/// <summary>

/// ジョインしている全てのクライアントに通知

/// </summary>

/// <param name="packet"></param>

public void UpdateStatus(Models.DeviceStatusPacket packet)

{

Clients.All.UpdateStatus(packet);

}

/// <summary>

/// 温度をグループにジョインしているクライアントに通知

/// </summary>

/// <param name="packet"></param>

public void UpdateTemperatureSensor(Models.TemperaturePacket packet)

{

Clients.Group(packet.DeviceId).UpdateTemperatureSensor(packet);

}

/// <summary>

/// 加速度をグループにジョインしているクライアントに通知

/// </summary>

/// <param name="packet"></param>

public void UpdateAccelerometerSensor(Models.AccelerometerPacket packet)

{

Clients.Group(packet.DeviceId).UpdateAccelerometerSensor(packet);

}

}

最後に、*IoTCloud*プロジェクト直下の“Startup.cs”ファイルを以下の様に変更します。

public partial class Startup

{

public void Configuration(IAppBuilder app)

{

ConfigureAuth(app);

**app.MapSignalR();**

}

}

StartupクラスのConfigurationメソッドは、Webアプリ初期化時にコールされます。太字で示した、MapSignalRメソッドをコールして、SignalRの機能の初期化を行います。

SignalRのハブはこれで完成です。UpdateStatusメソッドはデバイスの状態を、このハブにサブスクライブしている全てのクライアントに通知します。UpdateTemperatureSensorメソッドは温度センサーの計測値を、UpdateAccelerometerSensorメソッドは加速度センサーの計測値を、Joinメソッドでデバイス名のグループにジョインしているクライアントのみに通知します。

## WebアプリへのSignalR送信ロジック追加

前のステップ4.1．で作成したSignalRのハブに送信するロジックを、ステップ2.3．で改造したWebアプリに追加します。

SensorControllerクラスにSignalRハブへの送信用メソッドを追加します。

private void SendToSignalRHub(SensorReading sensorReading)

{

var context = GlobalHost.ConnectionManager.GetHubContext<Hubs.DeviceMonitorHub>();

context.Clients.Group(sensorReading.DeviceId).

UpdateStatus(

new Models.DeviceStatusPacket()

{

DeviceId = sensorReading.DeviceId,

Status = sensorReading.DeviceStatus

});

if (sensorReading.SensorType == "temperature")

{

context.Clients.Group(sensorReading.DeviceId).

UpdateTemperatureSensor(

new Models.TemperaturePacket()

{

DeviceId = sensorReading.DeviceId,

Temperature = Double.Parse(sensorReading.SensorValue)

});

}

}

ここに示すように、SignalRハブへの送信は、

1. GlobalHostクラスのConnectionManagerプロパティのGetHubContextメソッドを使って、ハブのコンテキストを取得
2. ハブクラスに定義したメッセージ送信用メソッドを送信する

というパターンで実現可能です。

このメソッドをコールするコードを、Getメソッドの最後に追加します。

public async void Get()

{

・・・ 中略　・・・

SendDataToMS(sensorReading);

**SendToSignalRHub(sensorReading);**

}

これで、Webアプリ側の作業は完了です。

## 表示WebアプリでのSignalR受信ロジック追加

ここでは、2.4．で作成した、DeviceStatusPage.aspxに、SignalR受信ロジックと、受信した際の表示更新ロジックを追加します。追加するコードは、太字で示した部分です。

<head runat="server">

<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

<title></title>

<script type="text/javascript" src="Scripts/jquery-1.10.2.min.js" ></script>

**<script type="text/javascript" src="Scripts/jquery.signalR-2.0.2.min.js"></script>**

**<script type="text/javascript" src="SignalR/hubs"></script>**

<script src='http://ajax.aspnetcdn.com/ajax/mobileservices/MobileServices.Web-1.1.2.min.js'></script>

<script type="text/javascript">

$(function () {

var xmlHttp = null;

function DeviceStatusModel() {

this.DeviceId = "";

this.DeviceStatus = "";

};

var rows = 0;

var table = document.getElementById("table1");

function AddRow(pId,pStatus) {

var row = table.insertRow(++rows);

var colName = row.insertCell(0);

colName.innerHTML = pId;

var colDesc = row.insertCell(1);

colDesc.innerHTML = pStatus;

}

AddRow("name"+rows,"desc"+rows);

var client = new WindowsAzure.MobileServiceClient(  
"https://*[iotkitms]*.azure-mobile.net/", "*[Access Key]*");

var dsTable = client.getTable('DeviceStatus');

var read = dsTable.read();

read.done(function (result) {

for (var i = 0; i < result.length;i++) {

AddRow(result[i].deviceId, result[i].status);

}

}, function (error) {

alert(error);

});

**var deviceStatusHub = $.connection.DeviceMonitorHub;**

**deviceStatusHub.on("UpdateStatus", function (packet) {**

**for (var i = 1; i < table.rows.length; i++) {**

**var candidate = table.rows[i].cells[0];**

**if (candidate.innerHTML == packet.deviceid) {**

**table.rows[i].cells[1].innerHTML = packet.status;**

**break;**

**}**

**}**

**});**

**$.connection.hub.start();**

});

</script>

</head>

ここでは、SignalRを利用するためのライブラリ参照追加と、DeviceMonitorHubに接続し、UpdateStatusメッセージが発信されたときのコールバックロジックを定義しています。JavaScriptでSignalRを使用するときは、このパターンを利用します。Azureに発行して、DeviceStatusPage.aspxをブラウザで開き、IoT Kitハードウェアのアプリを実行して、センサーを温めて表示が変わるか確認してください。

見栄えについて

この一連のトレーニングでは、見栄えは考慮していません。Webの表示もそっけないものばかりです。Coolな感じのスタイルに変更するのは皆さんへの課題としておきます。

## ストアアプリでのSignalR受信ロジック追加

このステップでは、3.2．で改造したWindows ストアアプリにSignalRの受信ロジックを追加します。

先ずは、“*iotkitms*.Windows”プロジェクトに、SignalRにアクセスするためのSDKをインストールします。

ソリューションエクスプローラーで“参照”を右クリックし、”NuGetパッケージの管理“を選択します。



これで準備完了です。これからのステップで、全IoT Kitハードウェアの状態のSignalRによる受信と表示更新と、特定のIoT Kitハードウェアの温度センサーのSignalRによる受信と表示を追加していきます。

### デバイスの状態を受信する

“*iotkitms*.Windows”の”DeviceStatusPage.xaml.cs“のDeviceStatusPageクラスに、以下のメソッドを追加します。

IHubProxy deviceStatusHub;

async void MakeConnection()

{

Exception ex = null;

try

{

var hubConnection = new HubConnection("http://*iotcloud*.azurewebsites.net");

deviceStatusHub = hubConnection.CreateHubProxy("DeviceMonitorHub");

await hubConnection.Start();

var subscribe = deviceStatusHub.Subscribe("UpdateStatus");

subscribe.Received += subscribe\_Received;

}

catch (Exception e)

{

ex = e;

}

if (ex != null)

{

var dialog = new MessageDialog(ex.Message, "SignalR Error");

await dialog.ShowAsync();

}

}

async void subscribe\_Received(IList<Newtonsoft.Json.Linq.JToken> obj)

{

await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

string deviceId = "";

string status = "";

foreach (var packet in obj)

{

deviceId = packet.Value<string>("deviceid");

status = packet.Value<string>("status");

}

var items = devices.Where((ds) => (ds.DeviceId == deviceId));

if (items.Count() > 0)

{

items.First().Status = status;

}

});

}

MakeConnectionメソッドの中で、4.1．で追加したSignalRハブへの接続とUpdateStatusへのサブスクライブを行っています。例外のハンドリングが入って少々煩雑ですが、コードを読み込んでみてください。

subscribe\_Receivedメソッドは、SignalRハブにUpdateStatusが送信されたときにコールされるコールバックメソッドです。このメソッドのコールは、UIを管理するスレッドとは別のスレッドからコールされるので、DispatcherプロパティのRunAsyncメソッドをコールすることによりUIスレッドに処理を委譲しています。これもパターンとして覚えてください。このメソッドの中で、SignalRハブから送信されたデータパケット（4.2で定義したModels.SignalRPackets.cs内のDeviceStatusPacket。JSONフォーマットで送信されてくる）から、デバイスのIDと状態を取り出しています。更に、”*iotkitms*.Shared”側のDeviceStatusPage.csで定義されているMobileServiceCollection型のコンテナ変数devicesから、該当する識別子を持つアイテムを探し、Statusを更新しています。

MobileServiceCollectionはObservableCollectionを継承していて、かつ、“*iotkitms*.Windows”プロジェクト側のDeviceStatusPage.xamlで定義されているListVeiwのdeviceItemsのItemsSourceにdevicesが代入されているので、このコードを実行することにより、UI上の表示が更新されます。

後はMakeConnectionメソッドをコールするコードを“*iotkitms*.Shared”側のDeviceStatusPage.csのOnNavigatedToメソッドに追加します。

protected override void OnNavigatedTo(Windows.UI.Xaml.Navigation.NavigationEventArgs e)

{

RefreshDevices();

**MakeConnection();**

base.OnNavigatedTo(e);

}

これで、DeviceStatusPageが表示されたときに、SignalRハブへの接続が行われ、SignalRハブに対して、UpdateStatusが通知されるたびに、DeviceStatusPage上の状態表示が変わります。

※注意  
同名のファイルが複数プロジェクトにあるので、修正はどのプロジェクトに対するものか慎重に判断して作業を進めてください。

### 温度センサー計測値を受信する

ここでは、DeviceStatusPageに表示されたIoT Kitハードウェアの状態一覧リストの項目が、どれか一つ選択されたときに、特定ハードウェアの温度表示用のページを表示するように“*iotkitms*.Windows”プロジェクトに変更を加え、温度表示用のページでSignalRハブのUpdateTemperatureSensorメッセージを受信し、温度の表示を変える、という変更を加えます。

先ず、“*iotkitms*.Windows”プロジェクトに、”TemperaturePage.xaml“という名前で、”空白のページ“を追加し、デバイスIDと温度を表示するよう修正します。

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

**<Grid.RowDefinitions>**

**<RowDefinition/>**

**<RowDefinition/>**

**</Grid.RowDefinitions>**

**<TextBlock x:Name="tbDeviceId" FontSize="48" HorizontalAlignment="Center" VerticalAlignment="Center"/>**

**<TextBlock x:Name="tbTemperature" FontSize="54" HorizontalAlignment="Center" VerticalAlignment="Center"/>**

</Grid>

</Page>

次に、TemperaturePage.xaml.csファイルを開き、TemperaturePageクラスに、以下のメソッドを追加します。

HubConnection hubConnection;

IHubProxy deviceStatusHub;

async void MakeConnection(string deviceId)

{

Exception ex = null;

try

{

hubConnection = new HubConnection("http://*iotcloud*.azurewebsites.net");

deviceStatusHub = hubConnection.CreateHubProxy("DeviceMonitorHub");

await hubConnection.Start();

await deviceStatusHub.Invoke("Join", new object[] { deviceId });

var subscribe = deviceStatusHub.Subscribe("UpdateTemperatureSensor");

subscribe.Received += subscribe\_Received;

}

catch (Exception e)

{

ex = e;

}

if (ex != null)

{

var dialog = new MessageDialog(ex.Message, "SignalR Error");

await dialog.ShowAsync();

}

}

async void subscribe\_Received(IList<Newtonsoft.Json.Linq.JToken> obj)

{

await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

string deviceId = "";

double temperature = 0;

foreach (var packet in obj)

{

deviceId = packet.Value<string>("deviceid");

temperature = packet.Value<double>("temperature");

}

tbTemperature.Text = temperature.ToString();

});

}

protected override void OnNavigatedTo(NavigationEventArgs e)

{

DeviceStatus targetDevice = e.Parameter as DeviceStatus;

tbDeviceId.Text = targetDevice.DeviceId;

MakeConnection(targetDevice.DeviceId);

base.OnNavigatedTo(e);

}

protected override async void OnNavigatedFrom(NavigationEventArgs e)

{

await deviceStatusHub.Invoke("Leave", new object[] { tbDeviceId.Text });

hubConnection.Stop();

base.OnNavigatedFrom(e);

}

このページが表示されるとき、OnNavigateToメソッドがコールされます。このページへの遷移で引数として表示対象のIoT KitハードウェアのDeviceIdが渡されるものとしています。MakeConnectionメソッドは、デバイスの状態を受信するときとほぼ同じ形をしていますが、温度情報通知は、SignalRハブが管理するグループに送信されるよう定義されているので、Joinメッセージをコールして、先ずはグループに所属するようなコードが追加されています。逆にこのページから別のページに遷移する際には、SignalRによる受信は止めないといけないので、OnNavigatedFromメソッドの中で、Leaveメッセージを送ってグループから脱退し、更には、ハブ接続のhubConnectionのStopメソッドをコールして通信を止めています。

最後に、“iotkitms.Windows”のDeviceStatusPage.xamlのListView型のdeviceItemのSelectionChangedイベントにハンドラを追加し、

<TextBlock Text="Device Status" FontSize="72" VerticalAlignment="Center" HorizontalAlignment="Center"/>

<ListView x:Name="deviceItems" Grid.Row="1" Margin="20"   
**SelectionChanged="deviceItems\_SelectionChanged"**>

<ListView.ItemTemplate>

DeviceStatusPage.xaml.cs内に生成された”deviceItems\_SelectionChanged”メソッドに、TemperaturePageのページに遷移するコードを追加して、リストからIoT Kitを選択した時に選択されたIoT Kitハードウェアの温度を表示するページが表示するように変更を加えます。

private void deviceItems\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

**Frame.Navigate(typeof(TemperaturePage),deviceItems.SelectedItem);**

}

以上で修正作業は完了です。クラウド側のWebアプリは発行しておいてください。

IoT Kitハードウェア上でアプリを動作させ、温度センサーを指等で温めて、プッシュ通知が飛んできたらトーストをタップし、ストアアプリを起動します。リストから該当する項目を選択すると、TemperaturePageが表示され、IoT Kitハードウェアの温度センサーで計測した値が、ストアアプリ上で表示されます。

以上で、Step4 表示編の実習は終了です。

|  |
| --- |
| 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、で蓄えられたデータの表示ができるようになりました。  FAや医療機器、交通量、ビル管理など様々なシナリオを想定して、IoTシステムの構築にトライしてみてください。  複数のIoT Kitを接続したり、計測するセンサーを増やしたりするのも面白いでしょう。  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。 |