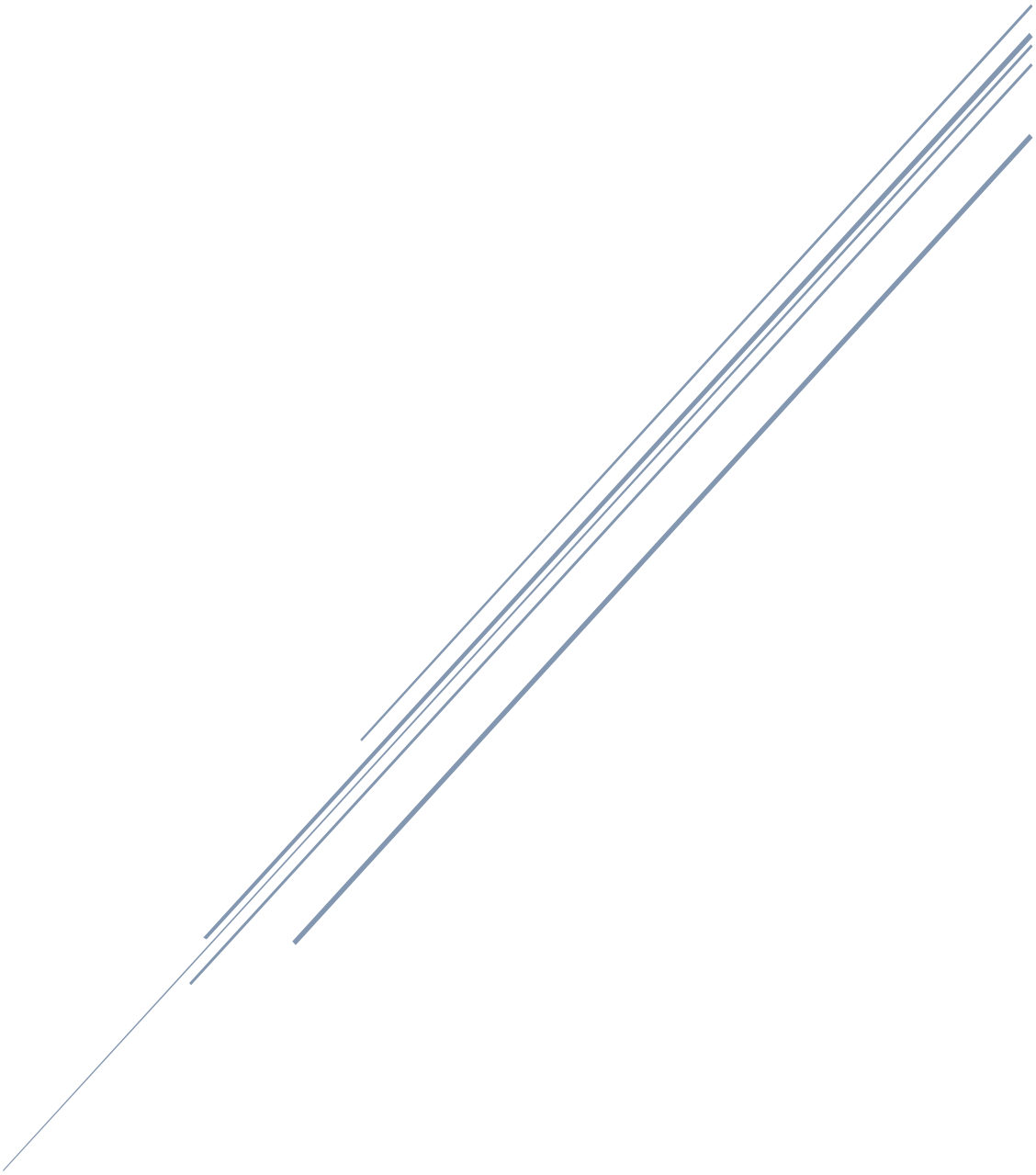
CIOF 開発者向けコントローラ作成スタートアップガイド

2022 年 7 月 1 日　Ver1.1



内容

1. 本書の目的

本書は、CIOF-SDK を用いた開発についてのスタートアップガイドです。
本書に従って開発すると、短いステップで CIOF-SDK を使って
データの送信または受信を行う Form アプリケーションを作成することができます。

2. 前提条件

・本書で説明するサンプルの開発には以下が必要です。

開発環境

・Visual Studio 2015 以降

フレームワーク

・.Net Framework 4.6.2 以降

言語

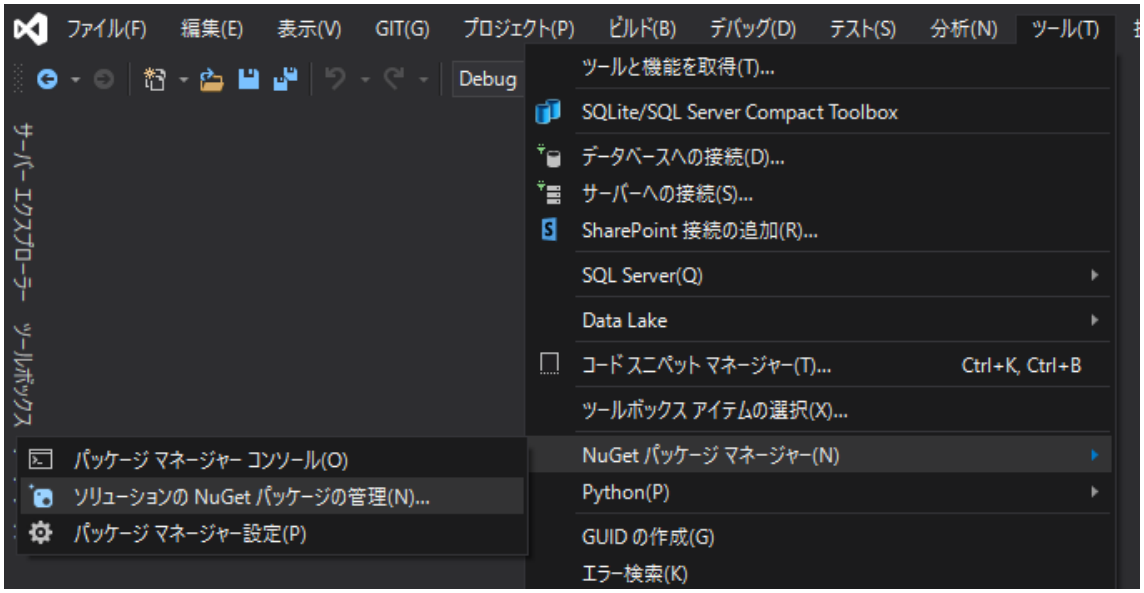
・C#

・連携ターミナル起動用の docker-compose.yml また docker については、
各自でダウンロードをお願いします。

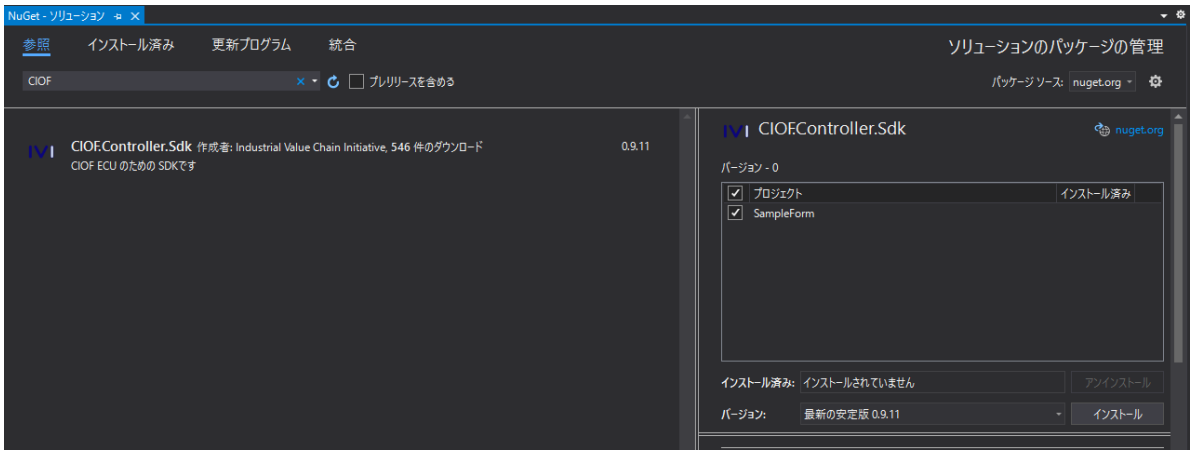
・辞書や取引については各ユーザーで設定していただくようお願いします。

3. CIOF-SDK のインストール

Visual Studio のメニューから、NuGet パッケージマネージャー
→ソリューションの NuGet パッケージの管理を選択します。



“CIOF”で検索すると、CIOF.Controller.Sdk が表示されます。
CIOF.Controller.Sdk を選択して、インストール対象のプロジェクトにチェックを入れます。
バージョンは最新を選択して、インストールボタンをクリックします。



4. サンプルプログラム作成手順

本章では、CIOF-SDK を使って簡単なデータの送受信を行うサンプルアプリケーションの開発手順を説明します。

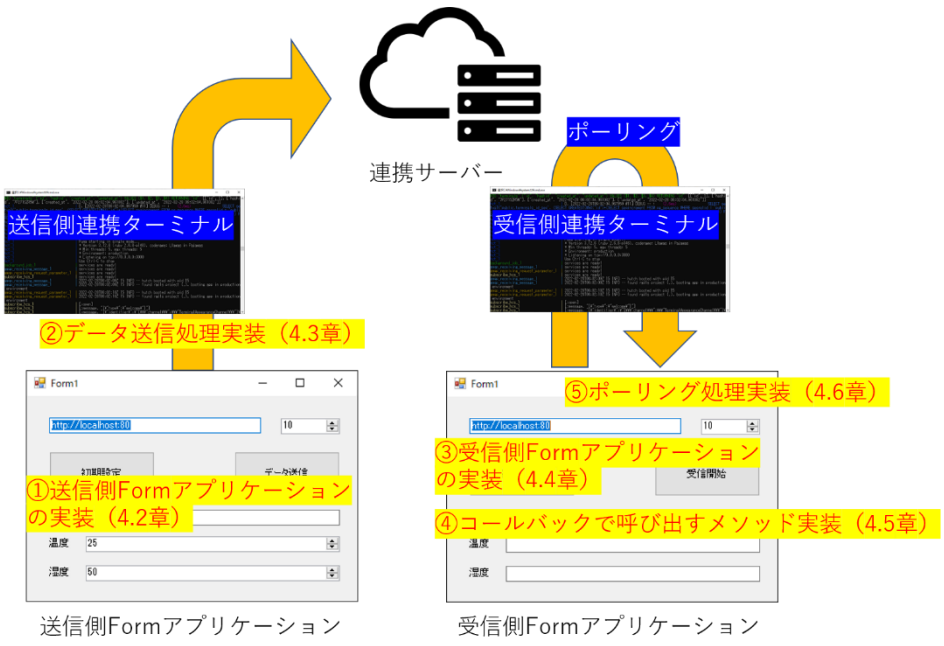
4.1.　アプリケーションの概要

以下の図は、今回実装するサンプルアプリケーションを使ったデータ送受信の概要です。

送信側では、連携ターミナルを通して、連携サーバーにデータを送信します。
4.2 ではアプリケーションに必要なボタン等を配置して
アプリケーションを作成する方法を説明します。
4.3 では、連携サーバーへのデータ送信処理方法を説明します。

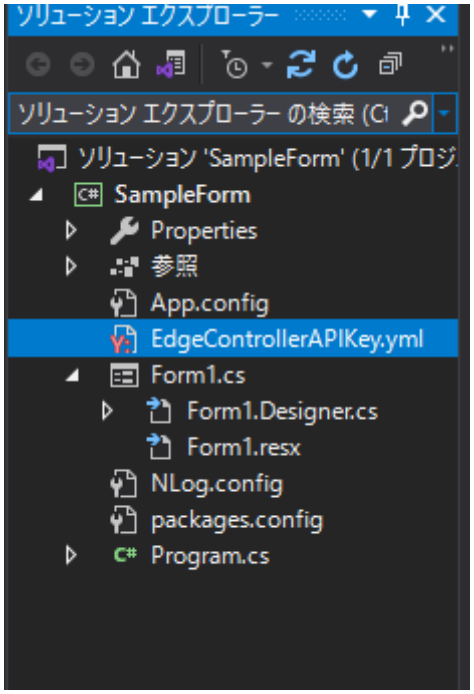
受信側では、連携ターミナルを通して、設定した周期でポーリングを行い、
連携サーバーにデータが送信されているか確認します。
データが送信されている場合は、予め登録したメソッドをコールバックで呼び出します。

4.4 ではアプリケーションに必要なボタン等を配置して
アプリケーションを作成する方法を説明します。
4.5 ではコールバックで呼び出すメソッドの実装と登録の方法を説明します。
4.6 では、ポーリング処理の方法を説明します。

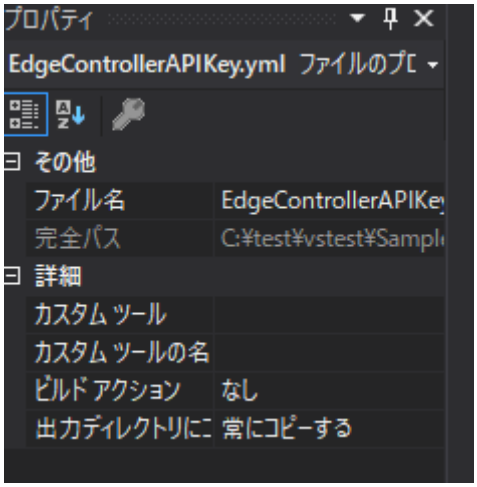


4.2.　送信側 Form アプリケーションの実装方法説明

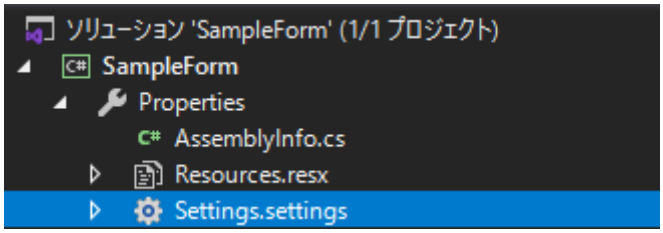
プロジェクトに、連携サイトから入手した EdgeControllerAPIKey.yml を配置します。



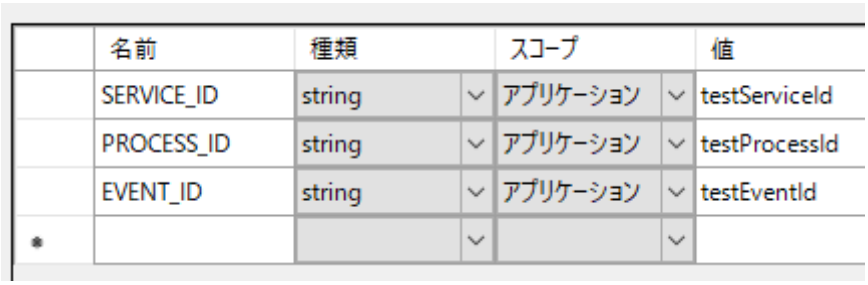
ファイルは出力ディレクトリにコピーできるように設定しておきます。



プロジェクトの Settings.settings ファイルを開きます。

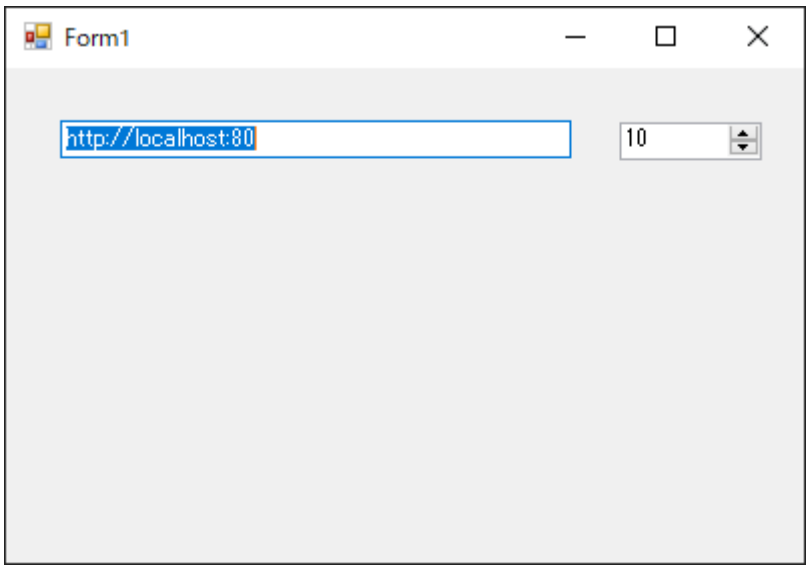


Settings.settings ファイルに、
内部サービス ID、内部プロセス ID、内部イベント ID など、
データ取引に必要な情報を定義しておきます。
スコープがアプリケーションの場合、App.config に設定した値が保存されるので、
内部 ID の変更があった場合でも、プログラムを書き換えることなく処理を実行できます。

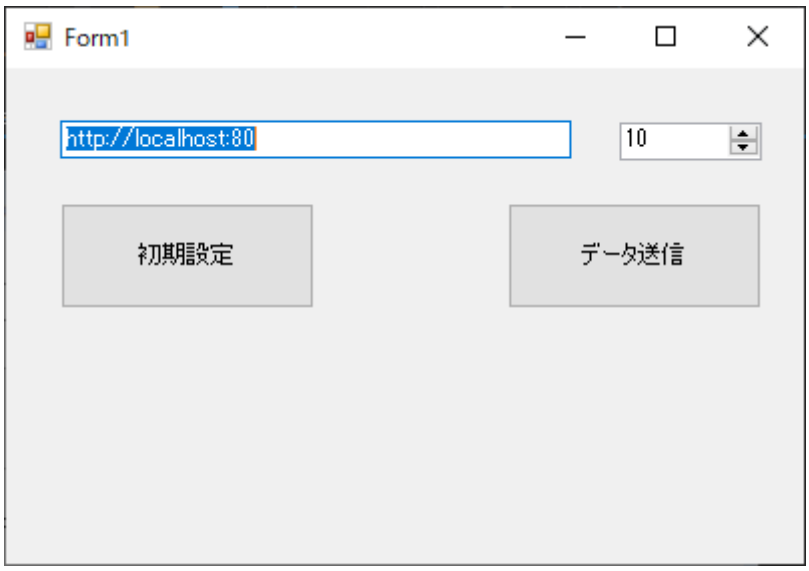


	名前	種類	スコープ	値
	SERVICE_ID	string	アプリケーション	testServiceId
	PROCESS_ID	string	アプリケーション	testProcessId
	EVENT_ID	string	アプリケーション	testEventId
*				

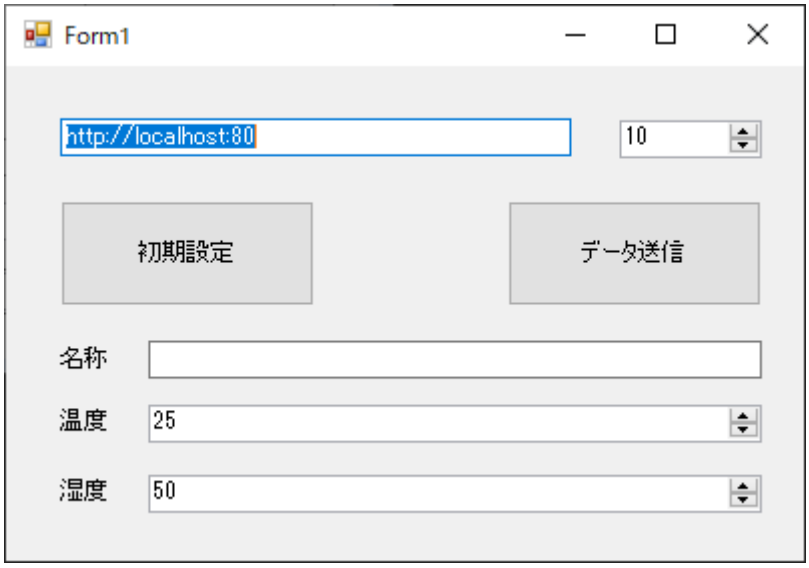
フォームのデザイナー上で、以下のものを配置します。
・URI 入力用の textbox（初期値は http://localhost:80 とします。）
・ポーリング周期入力用の numericUpDown（初期値は 10 とします。）



Form に初期設定ボタン、データ送信ボタンを配置します。



Form に名称、温度、湿度のラベルと入力用の TextBox、NumericUpDown を配置します。



4.3.　データ送信処理の実装方法説明

4.2 で配置した各ボタンのイベント、データの送信処理について説明します。

まずFormクラスにusing CIOF_SDK;を追加して、SDKのメソッドが使えるようにします。

```
using System;
using System.Windows.Forms;
using CIOF_SDK;
```

メンバ変数として controller を宣言します。

コンストラクタ内で、controller の初期化を行います。

コンストラクタの引数には、EdgeControllerAPIKey.yml を配置したパスを設定します。

```
ControllerModel controllerModel;
1 個の参照
public Form1()
{
    InitializeComponent();
    var currentDirectory = Directory.GetCurrentDirectory();
    controllerModel = new ControllerModel(currentDirectory);
}
```

初期設定ボタンクリック時のイベントで、controller の初期化メソッドである
InitialSetting を呼び出します。

第一引数には、uri 用の textBox の値、

第二引数にはポーリング周期用の numericUpDown の値を設定します。

```
/// <summary>
/// 初期設定ボタンクリック時のイベント
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 個の参照
private void btnInitial_Click(object sender, EventArgs e)
{
    controller.InitialSetting(tbxURI.Text, (int)nudPolling.Value);
}
```

これで、初期設定ボタンをクリックすると、コントローラの初期設定を行うことができます。

以上で、送信側アプリケーションの実装は完了です。

次に、データ送信ボタンクリック時のイベントを実装します。

using CIOF_SDK.Util;を追加して、SDK のメソッドが使えるようにします。
また、Properties についても追加します。

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using CIOF_SDK;
using CIOF_SDK.Util;
using SampleForm.Properties;
```

TypeUtil,CreateSendDataList、TypeUtil,CreateSendData メソッドを使って、
送信データ用のオブジェクトを作成します。

```
/// <summary>
/// データ送信ボタンクリック時のイベント
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 個の参照
private void btnSend_Click(object sender, EventArgs e)
{
    // 送信データ用のオブジェクトを作成
    var contentsList = TypeUtil.CreateSendDataList();
    var contents = TypeUtil.CreateSendData();
```

contents オブジェクトに、個別辞書で定義した値と送信するデータを追加します。

```
/// <summary>
/// データ送信ボタンクリック時のイベント
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 個の参照
private void btnSend_Click(object sender, EventArgs e)
{
    // 送信データ用のオブジェクトを作成
    var contentsList = TypeUtil.CreateSendDataList();
    var contents = TypeUtil.CreateSendData();
    contents.Add("名称", tbxName.Text);
    contents.Add("温度", (int)nudTemperature.Value);
    contents.Add("湿度", (int)nudHumidity.Value);
```

contents を contentsList に add します。
controller の PostTradeDataByServiceId メソッドを使ってデータを送信します。
第一引数には、サービス内部 ID、第二引数には contentsList を指定します。
PostTradeDataByServiceId 戻り値としてデータ ID のリストが返ってくるので、
List<string>の形で受けます。

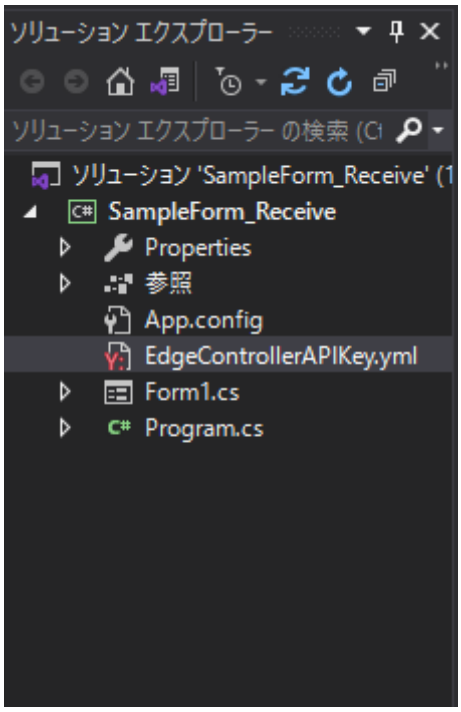
```
/// <summary>
/// データ送信ボタンクリック時のイベント
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 個の参照
private void btnSend_Click(object sender, EventArgs e)
{
    // 送信データ用のオブジェクトを作成
    var contentsList = TypeUtil.CreateSendDataList();
    var contents = TypeUtil.CreateSendData();
    contents.Add("名称", tbxName.Text);
    contents.Add("温度", (int)nudTemperature.Value);
    contents.Add("湿度", (int)nudHumidity.Value);
    contentsList.Add(contents);

    List<string> dataIdList = controller.PostTradeDataByServiceId(Settings.Default.SERVICE_ID, contentsList);
}
```

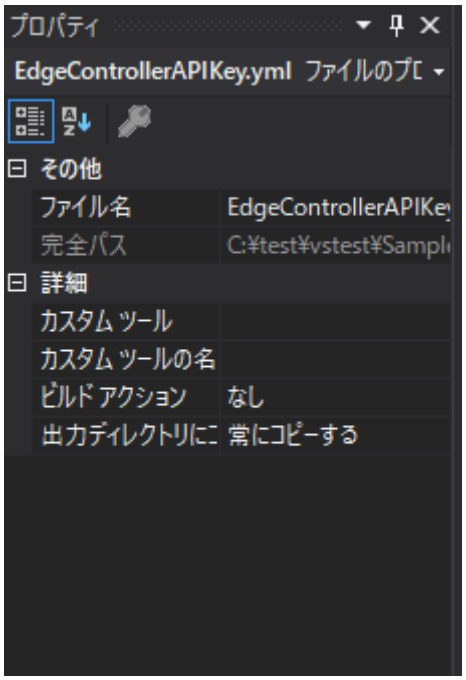
4.4.　受信側 Form アプリケーションの実装方法説明

ここからは、受信側 Form アプリケーションの実装について説明します。

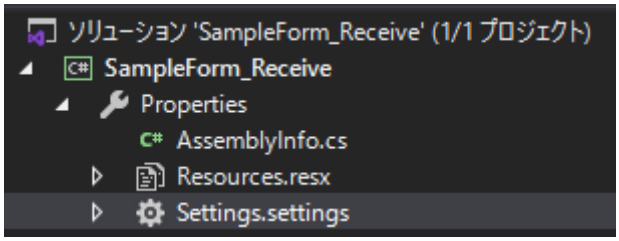
プロジェクトに、連携サイトから入手した EdgeControllerAPIKey.yml を配置します。



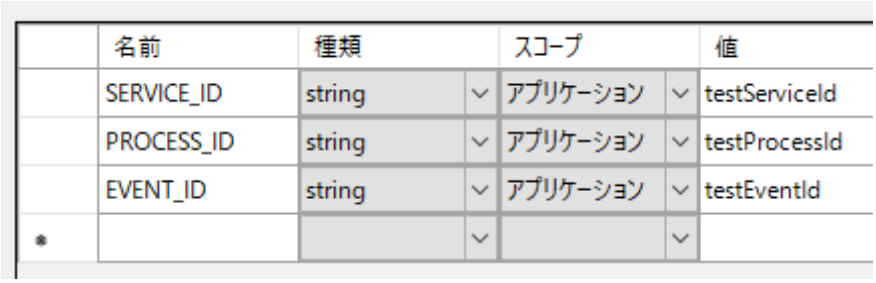
ファイルは出力ディレクトリにコピーできるように設定しておきます。



プロジェクトの Settings.settings ファイルを開きます。

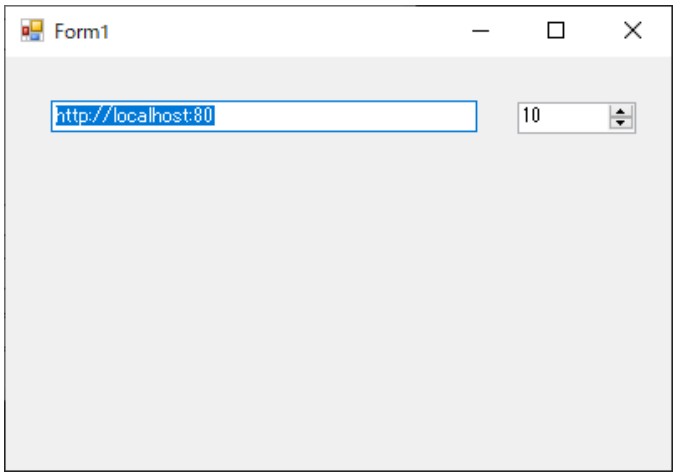


Settings.settings ファイルに、
内部サービス ID、内部プロセス ID、内部イベント ID など、
データ取引に必要な情報を定義しておきます。
スコープがアプリケーションの場合、App.config に設定した値が保存されるので、
内部 ID の変更があった場合でも、プログラムを書き換えることなく処理を実行できます。

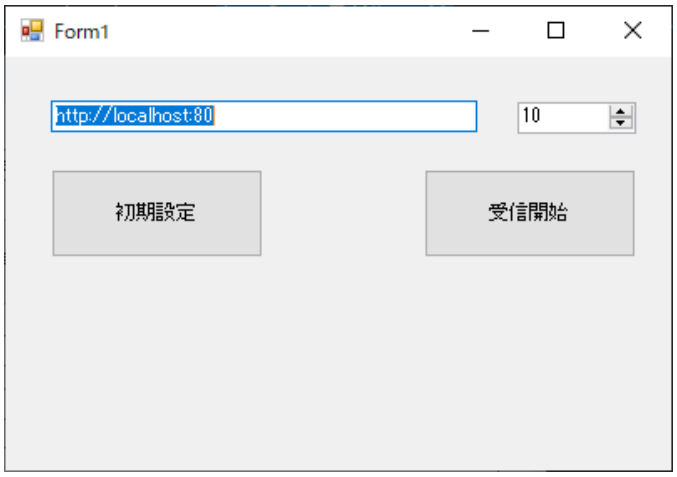


	名前	種類	スコープ	値
	SERVICE_ID	string	アプリケーション	testServiceId
	PROCESS_ID	string	アプリケーション	testProcessId
	EVENT_ID	string	アプリケーション	testEventId
*				

フォームのデザイナー上で、以下のものを配置します。
・URI 入力用の textbox（初期値は http://localhost:80 とします。）
・ポーリング周期入力用の numericUpDown（初期値は 10 とします。）



Form に初期設定ボタン、受信開始ボタンを配置します。



Form に名称、温度、湿度のラベルと Textbox を配置します。

4.5.　コールバックで呼び出すメソッドの実装方法説明

データ受信時に実行したいメソッドを実装して、SDK を使用してメソッドを登録しておくと、データを受信した際に、登録したメソッドがコールバックで呼び出されます。

ここでは、GetDataMethod というメソッドを作成します。

データ受信時に実行するメソッドは、以下の引数を持つ必要があります。

第一引数　string 取引契約 ID

第二引数　Dictionary<string, List<Dictionary<string, object>>> 受信したデータ

第一引数を id、第二引数を contentDict とします。

```
/// <summary>
/// データ受信用メソッド
/// </summary>
/// <param name="id">取引契約ID</param>
/// <param name="contentDict"></param>
1 個の参照
private void GetDataMethod(string id, Dictionary<string, List<Dictionary<string, object>>> contentDict)
{
```

データを受信した場合、contentDict は以下のような組を持った構造になっています。

Key：データ ID

Value：次の組についての List（Key：データ名、Value：データの値）

データを受信した場合、4.4 で配置した各テキストボックスに値を表示するようにします。

```
/// <summary>
/// データ受信用メソッド
/// </summary>
/// <param name="id">取引契約ID</param>
/// <param name="contentDict"></param>
1 個の参照
private void GetDataMethod(string id, Dictionary<string, List<Dictionary<string, object>>> contentDict)
{
    foreach (var dict in contentDict)
    {
        string dataId = dict.Key;
        foreach (var contentInfo in dict.Value)
        {
            if (contentInfo.ContainsKey("名称"))
            {
                tbxHumidity.Text = (string)contentInfo["名称"];
            }
            if (contentInfo.ContainsKey("温度"))
            {
                tbxTemperature.Text = (string)contentInfo["温度"];
            }
            if (contentInfo.ContainsKey("湿度"))
            {
                tbxTemperature.Text = (string)contentInfo["湿度"];
            }
        }
    }
}
```

4.6.　ポーリングによるデータ受信処理の実装方法説明

まず、送信側と同様に、初期設定を行います。
初期設定ボタンをダブルクリックすると、初期設定ボタンクリック時のイベントを
実装することができます。

using CIOF_SDK;を追加して、SDK のメソッドが使えるようにします。

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using CIOF_SDK;
```

メンバ変数として controller を宣言します。
コンストラクタ内で、controller の初期化を行います。

```
ControllerModel controller;

1 個の参照
public Form1()
{
    InitializeComponent();
    controller = new ControllerModel();
}
```

まず、4.5 で実装したメソッドをコールバックで呼び出せるように登録します。

登録用の Dictionary オブジェクトを作成します。Dictionary の型は、
Dictonary<string, Action<string, Dictionary<string, List<Dictionary<string, object>>>>>
です。

```
の参照
ate void btnInitial_Click(object sender, System.EventArgs e)

// サービス実装の登録
var serviceDict = new Dictionary<string, Action<string, Dictionary<string, List<Dictionary<string, object>>>>>();
```

作成したオブジェクトに、サービス内部 ID と、サービス実装メソッドの組を add します。

```
// サービス実装の登録
var serviceDict = new Dictionary<string, Action<string, Dicti
serviceDict.Add(Settings.Default.SERVICE_ID, GetDataMethod);
```

controller の SetServiceMethod でサービスを登録します。
第一引数にサービス実装とサービス内部 ID の組を add した Dictionary をセットします。
第二引数と第三引数は null で OK です。(リクエストに応じたサービス実装メソッドを疎くしたい場合は、第二引数または第三引数に同様の手順でメソッドを登録します。)

```
// サービス実装の登録
var serviceDict = new Dictionary<string, Action<string
serviceDict.Add(Settings.Default.SERVICE_ID, GetDataMe
controller.SetServiceMethod(serviceDict, null, null);
```

メソッド登録後は、送信側と同様に、controller の初期化メソッドである
InitialSetting を呼び出します。
第一引数には、textBox の値、第二引数には numericUpDown の値を設定します。

```
// サービス実装の登録
var serviceDict = new Dictionary<string, Action<string, Dictior
serviceDict.Add(Settings.Default.SERVICE_ID, GetDataMethod);
controller.SetServiceMethod(serviceDict, null, null);
controller.InitialSetting(tbxURI.Text, (int)nudPolling.Value);
```

これで、初期設定ボタンをクリックすると、コントローラの初期設定を行うことができます。

次に受信開始ボタンクリック時のイベントを実装します。

controller の PollingStart メソッドを呼び出します。

これで、受信開始ボタンクリックでポーリングが開始されます。
ポーリング中にデータを受信すると、
初期設定で登録したメソッドが呼び出されます。

```
1 個の参照
private void btnStart_Click(object sender, EventArgs e)
{
    controller.PollingStart();
}
```

受信した場合は、4.5 で実装したメソッドに、サービス利用通知用のメソッド
PostServiceRecordByDataId でデータ利用した旨を通知します。

```
controller.PostServiceRecordByDataId(dataId,
                                     "use",
                                     "データを表示しました。",
                                     "1",
                                     Settings.Default.SERVICE_ID,
                                     Settings.Default.PROCESS_ID,
                                     Settings.Default.EVENT_ID);
```