Remote Application サンプル コード説明

RICOH
imagine. change.

Remote Application とは？

Remote Application

MFP外部から操作するアプリ
（PCアプリ、スマホアプリ…）

・MultiLink-Panel Application
・操作部ブラウザーNX上で動くWebアプリ

Remote Application の特徴

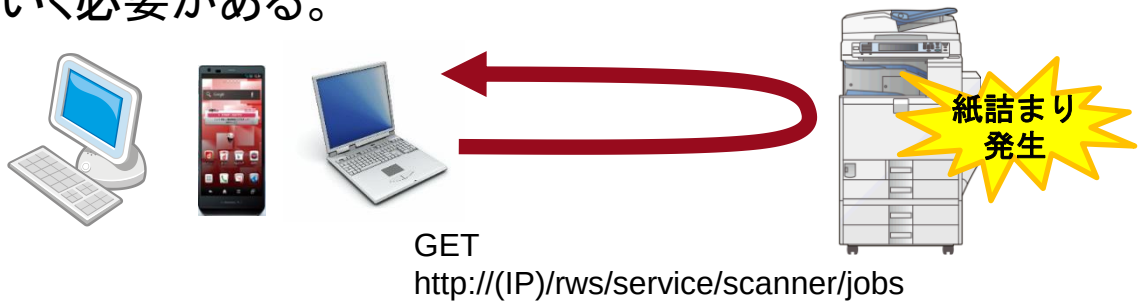
- Multi-Link Panel Application
- 操作部ブラウザNX上で動くWebアプリ

➡ イベントが通知される

ex) 紙詰まり が発生したら、
紙詰まりのイベント が通知される

- Remote Application

➡ イベントが通知されない。
自分で情報を取りにいく必要がある。



サンプルアプリ

RICOH
imagine. change.

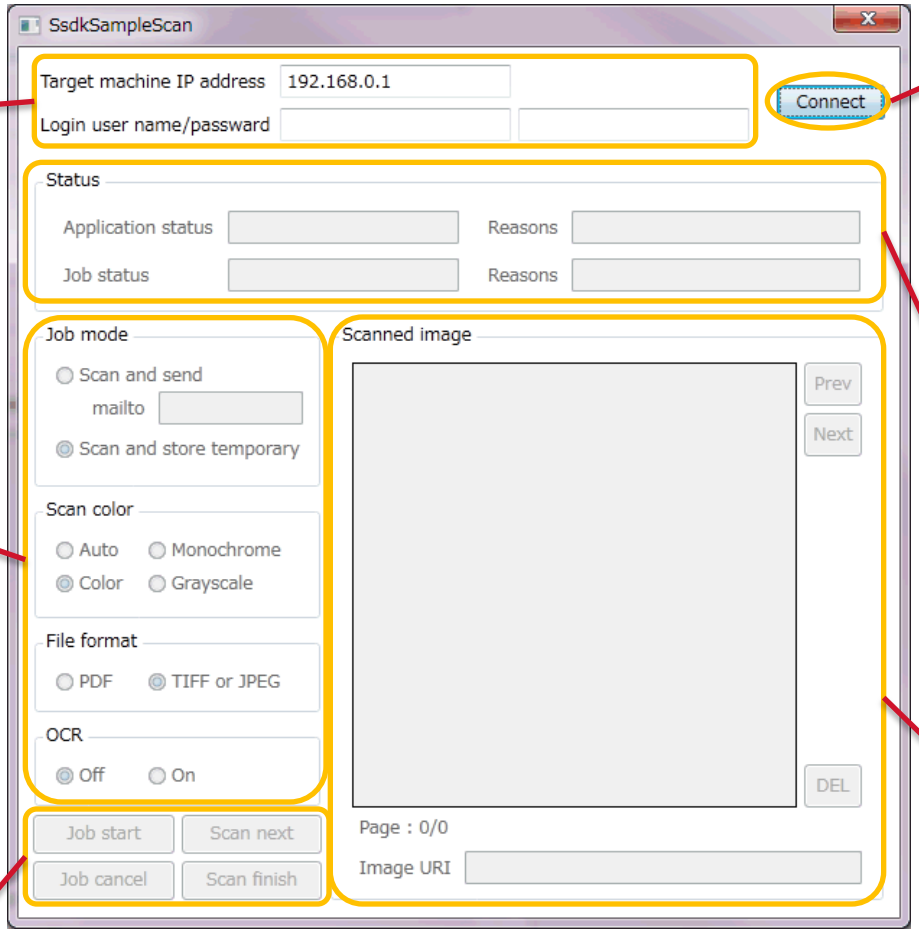
WindowsからWebAPIを用いてスキャンを実行するサンプル

- 使用言語
  - C#
- 使用ライブラリ
  - DynamicJson - @nuecc
    - http://dynamicjson.codeplex.com/
- 実行するには　.NetFramework4.5　が必要

サンプルアプリ GUI

RICOH
imagine. change.

IP、ユーザ名、パスワード入力



WebAPIのセッショントークン作成

アプリ状態（※）、ジョブ状態表示

※MFPのスキャナ自体の状態のこと

ジョブ設定項目

スキャン結果画像表示

ジョブ実行操作

プロジェクト構成

RICOH
imagine. change.

※今回の説明に関係のないファイルは省略

```
SsdkSampleScan
   │  DynamicJson_1.2.0.0.zip
   │  SsdkSampleScan.sln
   └─SsdkSampleScan
        │  DynamicJson.cs        //Jsonパース用ライブラリのクラス
        │  MainWindow.xaml.cs    //GUIイベントハンドラクラス
        │  SsdkScan.cs           //WebAPIアクセス用クラス
        ├─bin
        │  ├─Debug
        │  │      SsdkSampleScan.exe   //サンプルアプリexeファイル
        │  └─Release
        ├─obj
        │  └─Debug
        └─Properties
```

サンプルの処理の流れ

RICOH
imagine. change.

Connect押下

↓

セッショントークン取得

↓

スキャナの能力、
アプリ状態を取得

↓

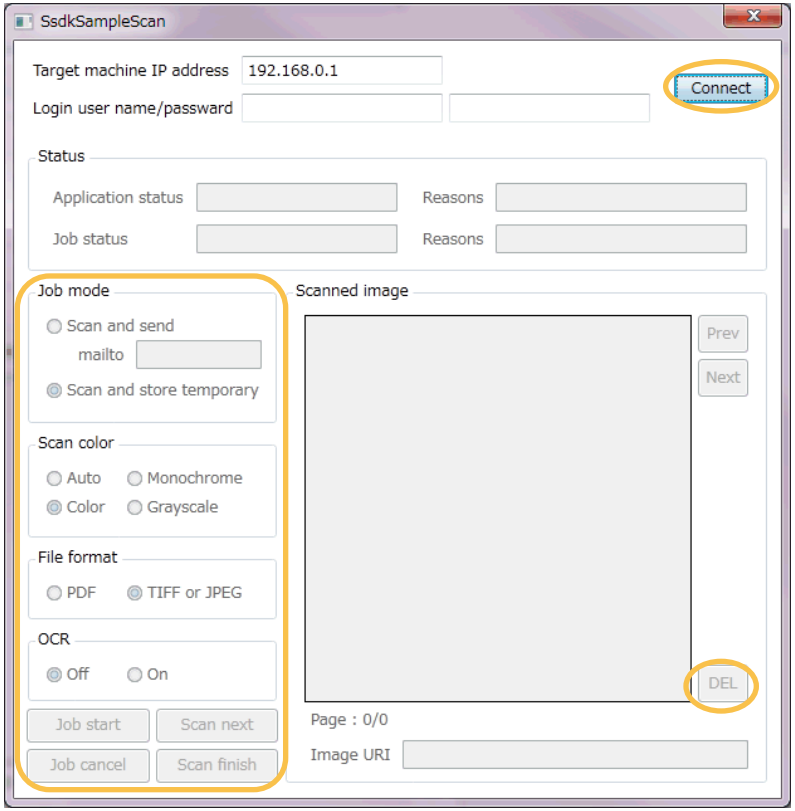
（ユーザが）ジョブ設定
＆ジョブ実行

↓

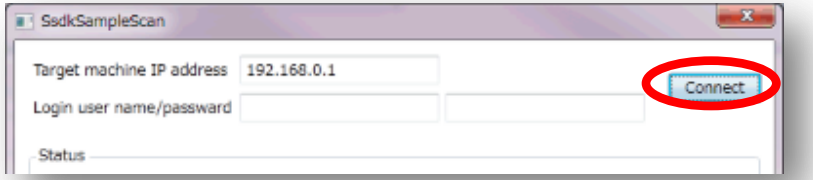
ジョブ開始＆画像を取得

↓

ジョブ完了　画像削除



セッショントークン取得



Connectボタン押下時の処理

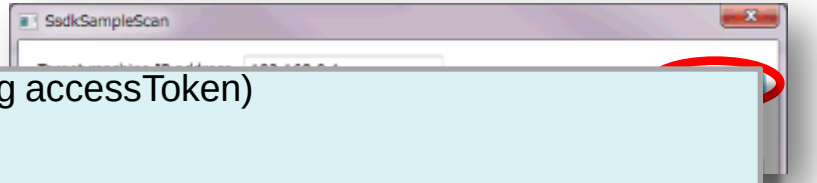
```
private async void btConnect_Click(object sender, RoutedEventArgs e)
{
 :
ssdkScan = new SsdkScan(tbIPaddr.Text, productId,
                                        tbLoginUser.Text, tbLoginPass.Text);
```

WebAPIを利用するインスタンスを生成

```
// セッショントークンを取得する
sessionToken = ssdkScan.GetSessionToken(accessToken);
if (sessionToken == null){
      MessageBox.Show("Get session token failed");
      return;
}
Task taskcheckCapability = checkCapability();
Task taskReloadStatus = getAplStatus();
```

セッショントークン取得

RICOH
imagine. change.



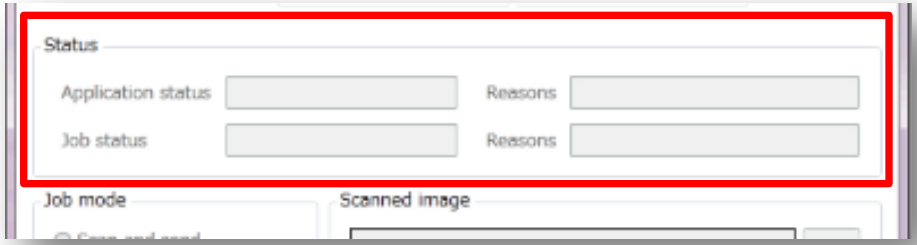

```
public string GetSessionToken(string accessToken)
{
    string sessionToken = null;
    HttpResponseMessage hr = null;
    string output;
    dynamic jsonSessionToken;

    bool ret = DoHttpRequest(HttpMethod.Post , accessToken, urlToken, null, out
hr);
    if (!ret){
        return null;
    }
    output = hr.Content.ReadAsStringAsync().Result;
    if (output == "" || !(hr.StatusCode == HttpStatusCode.Created)){
        return null;
    }

    jsonSessionToken = DynamicJson.Parse(output);
    if (jsonSessionToken.sessionToken()){
        sessionToken = jsonSessionToken.sessionToken;
    }else{
        return null;
    }
return sessionToken;    }
```

HTTPリクエストでWebAPIを利用

レスポンスを確認

レスポンスを
テキスト → オブジェクト

スキャナの能力&アプリ状態取得

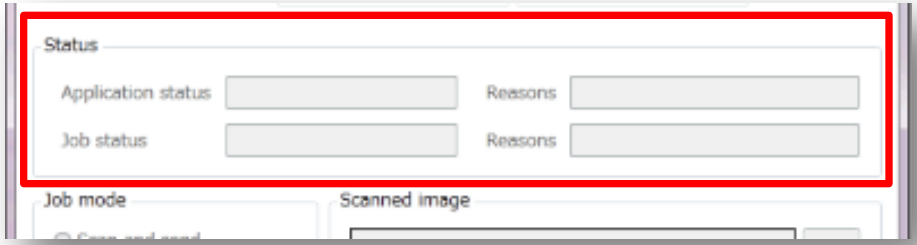



```
private async void btConnect_Click(object sender, RoutedEventArgs e)
{
 :
ssdkScan = new SsdkScan(tbIPaddr.Text, productId,
                                        tbLoginUser.Text, tbLoginPass.Text);

// セッショントークンを取得する
sessionToken = ssdkScan.GetSessionToken(accessToken);
if (sessionToken == null){
      MessageBox.Show("Get session token failed");
      return;
}
Task taskcheckCapability = checkCapability();
Task taskReloadStatus = getAplStatus();
```

能力取得＆アプリ状態(※) 取得

※MFPのスキャナ自体の状態のこと

スキャナの能力&アプリ状態取得



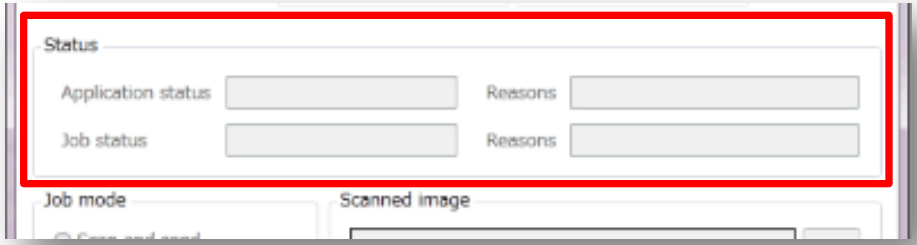

```
pr
{
 :
ss


// 
se
if (

}
Task taskcheckCapability = checkCapability();
Task taskReloadStatus = getAplStatus();
```

```
private Task checkCapability()
 {
        return Task.Run(() =>
        {
            // capabilityを取得する
            jsonCapability = ssdkScan.GetCapability(sessionToken);
            if (jsonCapability == null){
                MessageBox.Show("Get capability failed");
                return;
            }
    :           //capabilityの値で表示更新
```

能力取得＆アプリ状態(※) 取得

※MFPのスキャナ自体の状態のこと

能力値、アプリ状態を取得




```
public dynamic GetCapability(string sessionToken)
        {
            return GetRequestAndParse(sessionToken, uriCapability);
        }
```

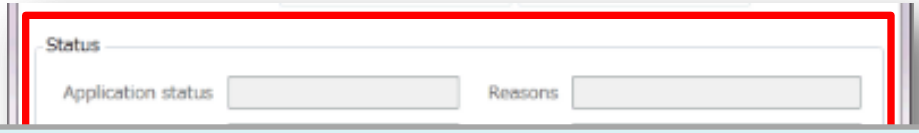
```
Task taskcheckCapability = checkCapability();
Task taskReloadStatus = getAplStatus();
```

能力取得＆アプリ状態(※) 取得

※MFPのスキャナ自体の状態のこと

スキャナの能力&アプリ状態取得

RICOH
imagine. change.



Connect押下
セッショントークン取得
スキャナの能力、
アプリ状態を取得
（ユーザが）ジョブ設定
＆ジョブ実行
ジョブ開始＆画像を取得
ジョブ完了　画像削除

```
public dynamic GetRequestAndParse(string sessionToken, string uri)
    {
        HttpResponseMessage hr = null;
        string output = "";
        dynamic json = null;

        bool ret = DoHttpRequest(HttpMethod.Get, sessionToken, uri, out hr);
        if (!ret){
            return null;
        }

        try{
            output = hr.Content.ReadAsStringAsync().Result;
            if (output == "" || !(hr.StatusCode == HttpStatusCode.OK)){
                return null;
            }
            // JSONをパースする
            json = DynamicJson.Parse(output);
        }catch (Exception){
        }

        return json;
    }
```

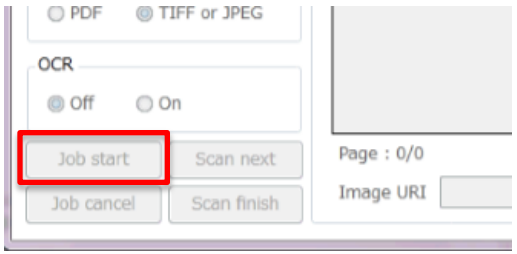
HTTPリクエストでWebAPIを利用

レスポンスを確認

レスポンスを
テキスト → オブジェクト

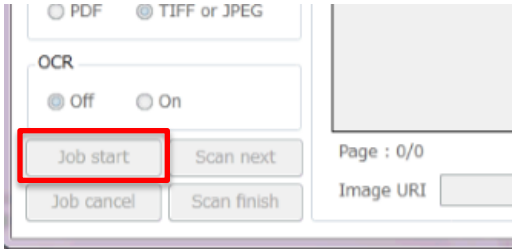
※アプリ状態も同様にWebAPIを利用し取得している

ジョブ開始




```
private void btJobStart_Click(object sender, RoutedEventArgs e){
        createJob();
}
```

ジョブ開始

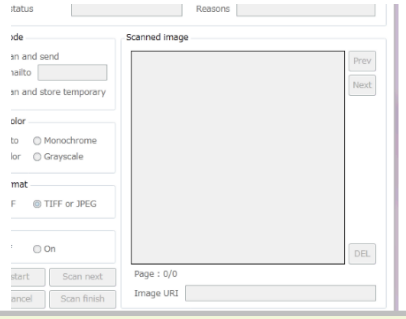



```
private Task createJob()
{
    return Task.Run(() =>
    {
        displayedTempImagePage = scannedCount = 0;    // ページ数を初期化

        jsonJobSetting = createJobSettingJson();    // ジョブ条件を生成

        jobId = ssdkScan.CreateJob(sessionToken, jsonJobSetting);   // ジョブ生
成


        getJobStatus();    // ジョブの状態を取得

        getAplStatus();    // スキャナ状態を取得

        timerJobStatus.Enabled = true;    // ジョブ状態を取得するためのタイマー起動

        setEnableUiControls();    // 画面更新
    });
}
```

これまでの説明と同様にWebAPIを利用し
ジョブの実行＆状態の取得を行っている

2秒ごとにジョブの状態を取得するための
タイマーを起動している。
タイマー自体はコンストラクタで生成。

画像を取得



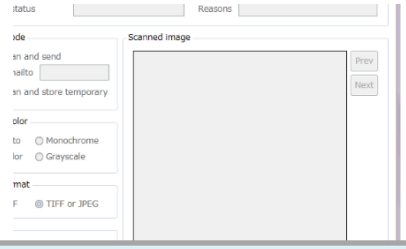

```
private void OnReloadJobStatus(object source, ElapsedEventArgs e)
{
        getJobStatus();
}

private Task getJobStatus()
{
        return Task.Run(() =>
          {
            dynamic json = ssdkScan.GetJobStatus(sessionToken, jobId);
            :
            displayTemporaryImage();    // 画像を取得して表示
 :
```

タイマーにより2秒ごとに呼び出される

ジョブ状態を取得

ジョブ状態を確認し原稿が読み取れていたら、
画像を取得＆表示

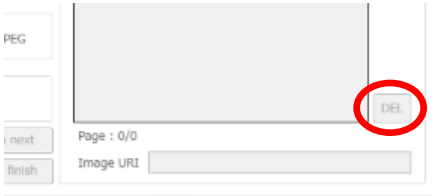
画像を取得




```
private Task displayTemporaryImage()
{
    return Task.Run(() =>
    {
        imImage.Dispatcher.Invoke(DispatcherPriority.Normal, new Action(() =>
        {
            tbImageUri.Text = fileUri + "?pageNo=" +
displayedTempImagePage.ToString();

            ImageSource img = ssdkScan.GetImageAsBitmap(sessionToken,
jobId

displayedTempImagePage);
            if (img == null)
            {
                imImage.Source = null;
                return;
            }

            imImage.Stretch = Stretch.Uniform;
            imImage.Source = img;
        }));
    });
}
```

WebAPIを利用し、画像を取得する

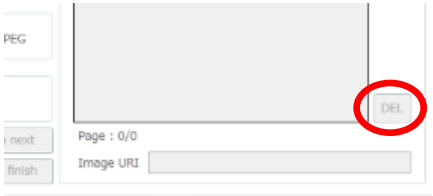
ジョブ完了、画像削除



ジョブ状態が完了になったら、
次のジョブを実行する前に、DELボタン押下しMFPに残っている画像を削除する

```
private void btDel_Click(object sender, RoutedEventArgs e)
{
        deleteTemporaryImage();
}
```

ジョブ完了、画像削除

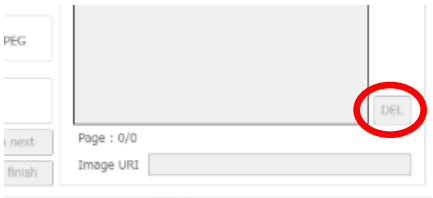


ジョブ状態が完了になったら、
次のジョブを実行する前に、DELボタン押下しMFPに残っている画像を削除する

```
private void btDel_Click(object sender, RoutedEventArgs e)
{
        de
}
```

```
private Task deleteTemporaryImage(){
   :
   ssdkScan.DeleteImage(sessionToken, jobId);
   :
}
```

ジョブ完了、画像削除



ジョブ状態が完了になったら、
次のジョブを実行する前に、DELボタン押下しMFPに残っている画像を削除する

```
private void btDel_Click(object sender, RoutedEventArgs e)
{
        de
}
```

```
public bool DeleteImage(string sessionToken, string
jobId)
{
    HttpResponseMessage responseMessage;

    bool ret =
DoHttpRequest(HttpMethod.Delete,sessionToken,
                    uriJobId + jobId + "/file", out
responseMessage);

    return ret;
}
```

WebAPIでMFP内の画像を削除

RICOH

imagine. change.