Wieldy – Addin-SDK – Tutorial
29.05.11

Introduction

Wieldy V0.2.6 introduced a simple API to develop add-ins. Those add-ins can be used to synchronize the items stored in Wieldy with external files or services. This tutorial will give a short introduction on how to use the SDK to write your own synchronizer. The tutorial will be a simple csv importer for the user contacts.

Note: This tutorial is based on the beta version 0.2.7.871. The interface may change in future versions of Wieldy.

Prerequisites

Wieldy is written in C# and build with the Microsoft .NET Framework 3.5. It is recommended to use Visual Studio to develop the add-in. Within Visual Studio you have to create a new "Class Library"-project and add a reference the assembly `Wieldy.Addins.Sdk.dll`. For this tutorial I used the project name `Wieldy.Synchronizer.CsvTutorial`.

Next you have to create two classes. The first class is called `CsvTutorialSynchronizerSettings` and is used to allow the user to configure your synchronizer. The second class is called `CsvTutorialSynchronizer`. This class contains the logic to execute the synchronization. To make this class work you have to implement the interface `ISynchronizer` and it additionally has to be marked by the attribute `Addin`.

The `Addin`-Attribute has two parameters. The first parameter is used to set the name which is displayed to the user. The second is used to configure the settings class which should be used.

Here is the source code:

```
using Wieldy.Addins.Sdk;

namespace Wieldy.Synchronizer.CsvTutorial
{
    public class CsvTutorialSynchronizerSettings : SynchronizerSettings
    {
    }

    [Addin("CSV Synchronizer (Tutorial)", typeof(CsvTutorialSynchronizerSettings))]
    public class CsvTutorialSynchronizer : ISynchronizer
    {
    }
}
```

The settings

Before we start implementing the synchronization logic, we want to finish the settings class. In this example the user can only configure the filename/location of the csv file. For every setting we want to provide, we have to add a property. Each property must be marked with a `SynchronizeSettings`-Attribute. Else it will be ignored. The parameters of this attribute are following:

- orderId, set the order in which the option is shown
- `type`, decide if the attribute should be editable or hidden from the user. The `EditableEx` option can be used to make a textfield which can be enabled/disabled by a checkbox.

- `title`, set the title for the option
- `tooltip`, set the tooltip description for this option
- `hint`, this text is shown as a watermark when the textfield is empty

Here is the complete code for our settings class:

```
    public class CsvTutorialSynchronizerSettings : SynchronizerSettings
    {
        [SynchronizeSettings(1, SynchronizeSettingsAttributeType.Editable,
         "Filename:", "Enter the filename/location of the .csv file ", "Filename")]
        public string Path { get; set; }

        // This synchronizer does not support password encryption
        [SynchronizeSettings]
        public override bool EnableEncryption
        {
            get { return false; }
            set { }
        }

        public override string GetDescriptionKey()
        {
            return "This is a sample contact synchronizer with csv files.";
        }
    }
```

The Synchronization Basics

When implementing the interface (`ISynchronizer`) for the synchronizer the following three properties are forced to further customize your synchronizer:

```
    [Addin("CSV Synchronizer (Tutorial)", typeof(CsvTutorialSynchronizerSettings))]
    public class CsvTutorialSynchronizer : ISynchronizer
    {
        // This is shown in the synchronisation window
        public string RemoteTranslationKey
        {
            get { return "CSV"; }
        }

        // Set the direction which is supported by this synchronizer
        public SynchronizerDirectionType SupportedSyncDirection
        {
            get { return SynchronizerDirectionType.Both;  }
        }

        // Configure if deleted items which are missing should still
        // be synchronized. In most cases set this to true.
        public bool IgnoreMissingDeleted
        {
            get { return true; }
        }

        ...
    }
```

When synchronization is started the method `Configure()` is called, following by a call of the method `StartSynchronisation()`. After the synchronization is completed the method `StopSynchronisation()` is called. A sample implementation of these three methods for our csv synchronizer is listed next. This implementation simple read the csv file when required and stores it in a list for later usage.

```
        ...

        private string csvPath;
        private List<CsvContact> contacts;

        public void Configure(SynchronizerSettings
          synchronizerSettings, System.Func<string, bool>
          continueHandler, System.Guid id)
        {
            var settings = (CsvTutorialSynchronizerSettings)
                                 synchronizerSettings;
            csvPath = settings.Path;
        }

        public void StartSynchronisation(bool readRequired,
          System.Security.SecureString readPassword)
        {
            contacts = new List<CsvContact>();
            if (readRequired)
            {
                var lines = File.ReadAllLines(csvPath);
                foreach (var line in lines)
                {
                    var elements = line.Split(',');
                    var contact = new CsvContact();
                    contact.Lastname = elements[0];
                    contact.Firstname = elements[1];
                    contact.EMail = elements[2];
                    contact.PhoneNumber = elements[3];
                    contacts.Add(contact);
                }
            }
        }

        public void StopSynchronisation(bool abort)
        {
        }

        ...
```

The `CsvContact` class looks like:

```
namespace Wieldy.Synchronizer.CsvTutorial
{
    public class CsvContact
    {
        public string Firstname
        {
            get;
            set;
        }

        public string Lastname
        {
            get;
            set;
        }

        public string EMail
        {
            get;
            set;
        }
        public string PhoneNumber
        {
            get;
            set;
        }
    }
}
```

Returning the content to Wieldy

After the data has been read by the method `StartSynchronisation()` we have to convert this data to a structure Wieldy understands. When doing this, we also have to match our items with existing items. Currently Wieldy does not provide any helper methods for this. This means each Add-In has to do this on its own. This add-in will check the lastname and the E-Mail address to match the contacts. You can also store unique ids on an item, so you can identify them later. This can be done by adding the id to the `SynchronizerIDs`-List of the item.

Note: Do not modify the items which are provided as the first parameter (`knownItems`) by the `ReadItems()`-method call!

```
        public IEnumerable<IItemBase> ReadItems(IEnumerable<IItemBase> knownItems,
bool buildValidSubEntries)
        {
            // Note: Do not modify the knownItems, as those are the real items
            // stored in Wieldy!!

            var items = new List<IItemBase>();
            foreach (var csvContact in contacts)
            {
                IContact contact = null;

                // Try to find existing user in Wieldy
                foreach (var knownItem in knownItems)
                {
                    var knownContact = knownItem as IContact;
                    if (knownContact!=null)
                    {
                        if (IsContactMatch(knownContact, csvContact))
                        {
                            contact = (IContact)knownContact.Clone();
                        }
                    }
                }

                // If no contact is known, create a new one
                if (contact==null)
                {
                    contact = InstanceCreator.Create<IContact>();
                }

                // Now set the data from the csv-File to the contact
                contact.Firstname = csvContact.Firstname;
                contact.Lastname = csvContact.Lastname;
                if(!contact.EMails.Contains(csvContact.EMail))
                {
                    contact.EMails.Add(csvContact.EMail);
                }
                contact.PrivatePhone = csvContact.PhoneNumber;
                // Set the item to valid and add it to the list
                contact.ValidEntry = true;
                items.Add(contact);
            }
            return items;
        }

        private static bool IsContactMatch(IContact knownContact, CsvContact 
csvContact)
        {
            return knownContact.Lastname == csvContact.Lastname &&
                   knownContact.EMails.Contains(csvContact.EMail);
        }
```

Acknowledging conflicts

Wieldy will now look for differences, after we have returned the list of items from the csv-File back to Wieldy. During this process, Wieldy may ask the add-in, if a conflict may be processed during this synchronization. For example, this synchronizer only returns contacts to Wieldy, but Wieldy may have other items like actions. For each action Wieldy will now ask the synchronization add-in if this item has been deleted or it is just not supported by the synchronizer. This is done by implementing the method `CanProcessConflict()`.

```
        public bool CanProcessConflict(ISynchronisationConflict conflict)
        {
            // We only support IContacts and items which are newly created by the 
addin
            return conflict.LocalItem == null || conflict.LocalItem is IContact;
        }
```

Processing conflicts

After the user has decided how the conflicts should be processed. The add-in may now receive calls to the methods AddItem(), UpdateItem() or RemoveItem().

```
        public void AddItem(IItemBase item)
        {
            var newContact = new CsvContact();
            var contact = (IContact) item;
            newContact.Firstname = contact.Firstname;
            newContact.Lastname = contact.Lastname;
            newContact.PhoneNumber = contact.PrivatePhone;
            if (contact.EMails.Count > 0)
            {
                newContact.EMail = contact.EMails[0];
            }
            contacts.Add(newContact);
        }

        public void RemoveItem(IItemBase item)
        {
            var contact = (IContact) item;
            foreach (var csvContact in contacts)
            {
                if (IsContactMatch(contact, csvContact))
                {
                    contacts.Remove(csvContact);
                    break;
                }
            }
        }
        public void UpdateItem(IItemBase newItem, IItemBase oldItem)
        {
            var oldContact = (IContact) oldItem;
            var newContact = (IContact) newItem;
            foreach (var csvContact in contacts)
            {
                if (IsContactMatch(oldContact, csvContact))
                {
                    csvContact.Firstname = newContact.Firstname;
                    csvContact.Lastname = newContact.Lastname;
                    csvContact.PhoneNumber = newContact.PrivatePhone;
                    if (newContact.EMails.Count > 0)
                    {
                        csvContact.EMail = newContact.EMails[0];
                    }
                    else
                    {
                        csvContact.EMail = string.Empty;
                    }
                    break;
                }
            }
        }
```

Saving the file

Now we only have to save the changes to the csv-file. This can be done in the method `StopSynchronisation()`. The code looks like:

```
        public void StopSynchronisation(bool abort)
        {
            if (abort == false)
            {
                using (var stream = new StreamWriter(csvPath))
                {
                    foreach (var csvContact in contacts)
                    {
                        stream.WriteLine("{0},{1},{2},{3}", csvContact.Lastname,
csvContact.Firstname, csvContact.EMail, csvContact.PhoneNumber);
                    }
                }
            }
        }
```

Final notes

That's it. We have written a simple csv-File-Synchronizer which can import/export contact-data into a csv-file. Beside the `IContact` interface there are a few more interfaces defining the different item types. Those are mainly the following: `IAction`, `IActionTemplate`, `IContact`, `IContactGroup`, `IProject`, `IReference` and `ITag`. Processing them is similar to the `IContact` interface shown here.

The source code for this sample can be downloaded here: `CsvTutorial.zip`

Please contact me if you need help.