ELOQUA

Cloud Connectors Sample Guide

Overview of Capability

Cloud Connectors are designed to allow you, as a third party, to create and operate your own step in Program Builder. It can be built to do any task that you choose, and runs as any Program Step does; as contacts, prospects, or companies flow into it, actions are executed against them.

There are three main aspects of a Cloud Connector:

Configuration of Program Builder

A step within Program Builder is set to have an action type of "Cloud Connector". The step is given an identity number.

Configuration of the Cloud Connector

In order to configure your Cloud Connector to do what it needs to do, a configuration window can be opened up from the "Configure" button on the Program Step. This passes key information in the URL:

- Program Builder StepID
- Eloqua Client Name

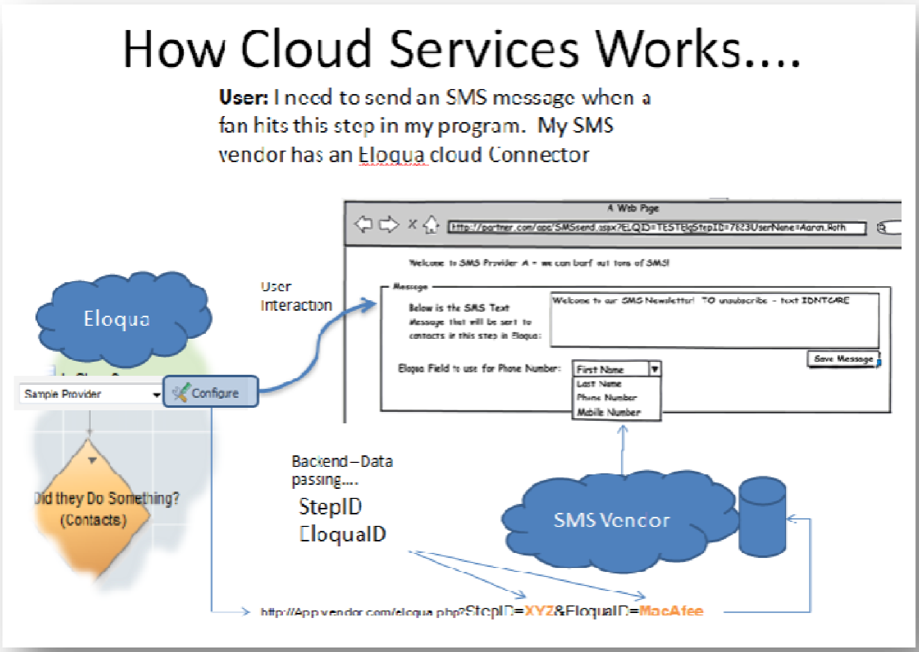
Note: in the initial building of your Cloud Connector, you don't need this configuration screen automatically linked; you can build a separate configuration screen that allows configuration of a Cloud Connector and this screen can be linked in once you have built and tested the Cloud Connector.

Running of the Cloud Connector

In order to build your Eloqua Cloud Connector, based on the step details that have been configured, you write and host a connector that polls the step looking for new members, reads the details of those members when in the step, and performs a required action.

The following diagram shows a conceptual view of this flow:

ELOQUA
THE POWER TO SUCCEED.



How Cloud Services Works....

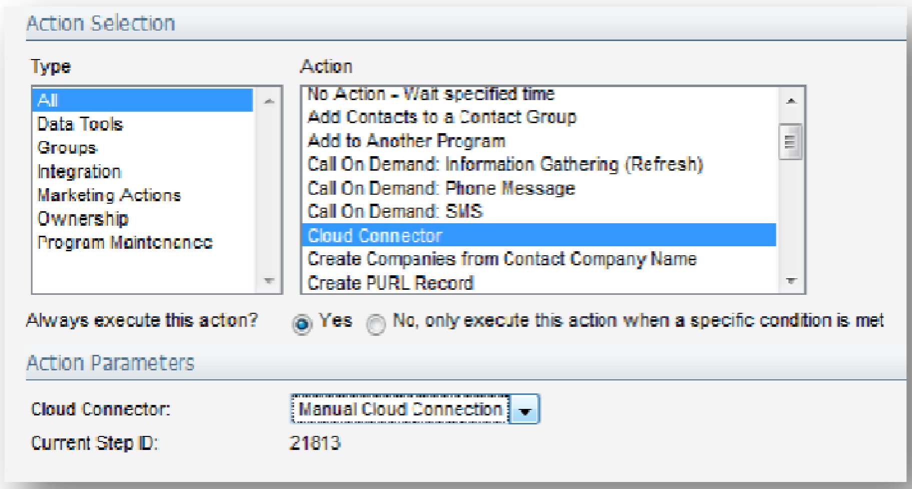
User: I need to send an SMS message when a fan hits this step in my program. My SMS vendor has an Eloqua cloud Connector

Eloqua
Sample Provider
Configure
Did they Do Something? (Contacts)
User Interaction
A Web Page
http://partner.com/app/SMSsend.aspx?ELQID=TESTEqStepID=7623UserName=Aaron.Roth
Welcome to SMS Provider A – we can barf out tons of SMS!
Message
Below is the SMS Text Message that will be sent to contacts in this step in Eloqua:
Welcome to our SMS Newsletter! TO unsubscribe – text IDNTCARE
Save Message
Eloqua Field to use for Phone Number:
First Name
Last Name
Phone Number
Mobile Number
Backend – Data passing....
StepID
EloquaID
SMS Vendor
http://App.vendor.com/eloqua.php?StepID=XYZ&EloquaID=MacAfee

Configuration of Program Builder

In Program Builder, add a new step for your example Cloud Connector. Set the action to “Cloud Connector”, and leave the Cloud Connector type of “Manual Cloud Connection” for now.

Make note of the Current Step ID (21813 in this example).


Action Selection

Type: All, Data Tools, Groups, Integration, Marketing Actions, Ownership, Program Maintenance

Action: No Action - Wait specified time; Add Contacts to a Contact Group; Add to Another Program; Call On Demand: Information Gathering (Refresh); Call On Demand: Phone Message; Call On Demand: SMS; Cloud Connector; Create Companies from Contact Company Name; Create PURL Record

Always execute this action? [✓ Yes] No, only execute this action when a specific condition is met

Action Parameters

Cloud Connector: Manual Cloud Connection

Current Step ID: 21813

Configuration of the Cloud Connector

The first step in building a Cloud Connector is building a configuration settings page that allows your user to tell you what they want to have happen when a person passes through the Program Step. In these examples, I will mainly be using C# on the Windows Azure platform to show how these can be built, but any technology environment can be used.

When we ultimately connect the configuration screen to Program Builder, it will have a format similar to the following:

http://App.vendor.com/eloqua.php?StepID=**XYZ**&EloquaID=**MacAfee**

Where:

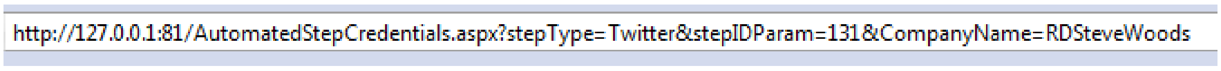
- **App.vendor.com** is the location where you have built your Cloud Connector software
- **Eloqua.php** is the page that you have built to allow the configuration of your Cloud Connector
- **StepID** is the parameter that holds the numeric identifier of your step
- **EloquaID** is the parameter that holds the name of the Eloqua instance that is calling for the step.

The structure of this URL string is configurable, but the same data will be passed when the user ultimately clicks on your "Configuration" link.

To build a Cloud Connector configuration page, accept these parameters, ask the user for Eloqua credentials that support an API call, and prompt the user for any required configuration parameters for the step.

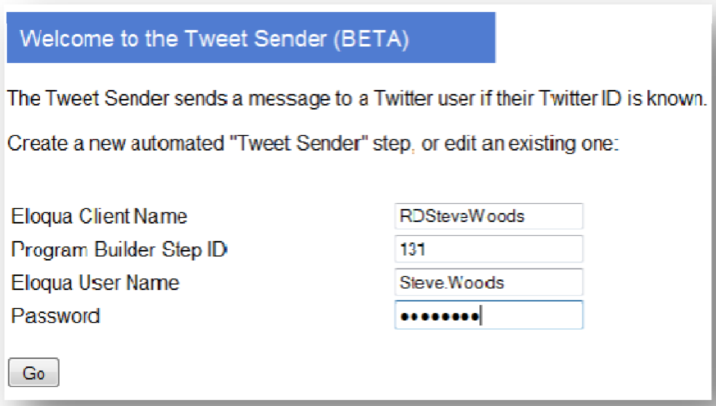
As a simple example, if we are going to build a very simple example step that sends out a Tweet to contacts in the step if they have a known TwitterID, we would build a Cloud Connector configuration page that did the following:

1) For a configuration URL (note that this URL has been configured from the default, and has additional parameters (hard coded) added to it:



```
http://127.0.0.1:81/AutomatedStepCredentials.aspx?stepType=Twitter&stepIDParam=131&CompanyName=RDSteveWoods
```

2) The following authentication step would be presented:



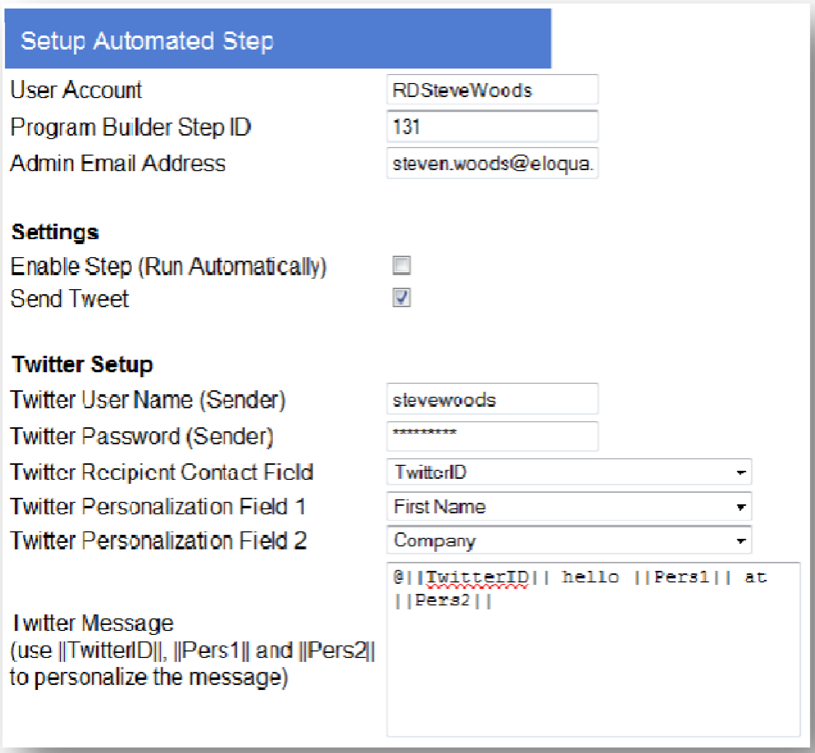
Welcome to the Tweet Sender (BETA)

The Tweet Sender sends a message to a Twitter user if their Twitter ID is known.

Create a new automated "Tweet Sender" step, or edit an existing one:

Eloqua Client Name	RDSteveWoods
Program Builder Step ID	131
Eloqua User Name	Steve.Woods
Password	•••••••••

3) This step would authenticate the user and then pass them on to the configuration screen for further setting and storage of the parameters required (in this case, the fields to use for the TwitterID, the message to send, and some personalization fields):



Setup Automated Step

User Account	RDSteveWoods
Program Builder Step ID	131
Admin Email Address	steven.woods@eloqua.

Settings

Enable Step (Run Automatically)	☐
Send Tweet	☑

Twitter Setup

Twitter User Name (Sender)	stevewoods
Twitter Password (Sender)	*********
Twitter Recipient Contact Field	TwitterID
Twitter Personalization Field 1	First Name
Twitter Personalization Field 2	Company
Twitter Message (use ||TwitterID||, ||Pers1|| and ||Pers2|| to personalize the message)	@||TwitterID|| hello ||Pers1|| at ||Pers2||

To do this, there are a couple of simple uses of the Eloqua API:

First, to set up a service proxy to Eloqua, add the following WSDL

https://secure.eloqua.com/API/1.2/Service.svc?wsdl

and give it a namespace (I have used EloquaService as the namespace for these examples)

To initiate the service proxy, provide it with the Instance Name, UserID and Password:

```
private EloquaService.EloquaServiceClient serviceProxy;
```

```
strInstanceName = InstanceName;
strUserID = UserID;
strUserPassword = UserPassword;
```

```
serviceProxy = new EloquaService.EloquaServiceClient();
serviceProxy.ClientCredentials.UserName.UserName = strInstanceName + "\\" + strUserID;
serviceProxy.ClientCredentials.UserName.Password = strUserPassword;
```

To authenticate, just run a simple query to verify access. If the query works, you have authenticated. In this case, we'll retrieve a single contact, with contact ID of 1:

```
try
{
    // Attempt to retrieve a single contact as an authentication test
    EloquaService.EntityType entityType = new EloquaService.EntityType();

    entityType.ID = 0;
    entityType.Name = "Contact";
    entityType.Type = "Base";

    // Set the ID of the Contact Entity
    int[] ids = new int[1];
    ids[0] = 1;

    // Create a new list containing the fields you want populated
    List<string> fieldList = new List<string>();
    // Add the Contact's relevant fields to the field list
    fieldList.Add("C_EmailAddress");
    // Build a Dynamic Entity array to store the results
    EloquaService.DynamicEntity[] retrievedEntities;
    // Execute the request and return only the selected Entity fields
    retrievedEntities = serviceProxy.Retrieve(entityType, ids, fieldList.ToArray());
    // If still processing, must have passed authentication
    blnAuthenticated = true;
}
catch (Exception ex)
{
    // Customize your own Error handling code.
    Console.WriteLine(String.Format("Exception Message: {0}", ex.Message.ToString()));
    blnAuthenticated = false;
}

return blnAuthenticated;
```

As part of setting up the configuration of your step, you often need to allow the user to configure one or more contact fields (to pull data from or push it back into). This can be quickly done through the API:

```
string[] tmpFields;
List<string[]> FieldList = new List<string[]>();

try
{
        // Execute the request
        EloquaService.EntityType entityType = new EloquaService.EntityType();

        entityType.ID = 0;
        entityType.Name = "Contact";
        entityType.Type = "Base";

        // Execute the request
        EloquaService.DescribeEntityResult result =
        serviceProxy.DescribeEntity(entityType);

        // Extract the ID, Name and Type of each Asset Type
        foreach (EloquaService.DynamicEntityFieldDefinition fieldDef in result.Fields)
        {
            tmpFields = new string[4];
            tmpFields[0] = fieldDef.DisplayName;
            tmpFields[1] = fieldDef.InternalName;
            tmpFields[2] = Convert.ToString(fieldDef.IsRequired);
            tmpFields[3] = Convert.ToString(fieldDef.IsWriteable);
            FieldList.Add(tmpFields);
        }
}
catch (Exception ex)
{
        // Customize your own Error handling code.
}
```

Any of the other configuration logic can be asked for in this configuration screen and saved against a record for that Eloqua Instance/Step ID.

Running of the Cloud Connector

Now, the third step is to build code to run the Cloud Connector for the step you have configured. This leverages a second API called the Cloud Services API to list, retrieve, and set the statuses of the contacts, prospects, or companies in each Program Builder Step.

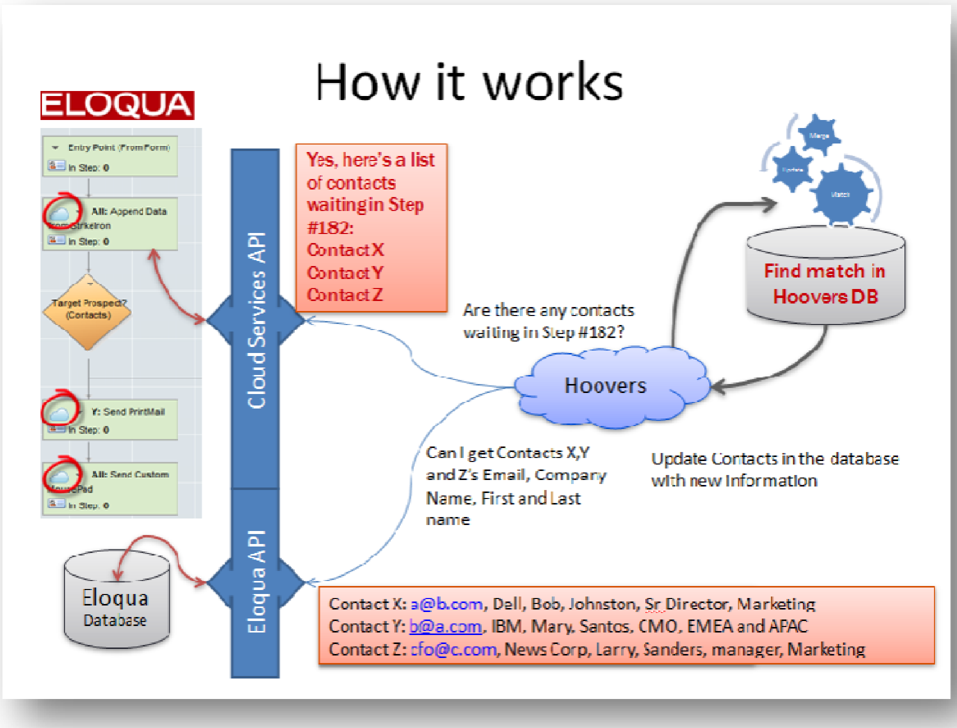
A simplified model of how a Cloud Connector is as follows:

- Every 5 minutes, your Cloud Connector polls the Cloud Services API to ask if there are Contacts in the selected step with a status of “Awaiting Action”
- If there are, those contacts are retrieved and set to “In Progress”
- The task of the Cloud Connector is performed
- The contacts are updated as needed (depending on the task of the Cloud Connector)
- The contacts in the step are set to “Complete”

From there, Program Builder picks up the contacts and moves them to the next step in the program.

That is all that is needed to create a Cloud Connector. What you build is entirely up to you.

Here is a quick example diagram of how a Cloud Connector might be built to connect to the Hoovers DB as a step in a Program:


How it works
ELOQUA
Entry Point (FromForm)
In Step: 0
All: Append Data from Harketron
In Step: 0
Target Prospect? (Contacts)
Y: Send PrintMail
In Step: 0
All: Send Custom Mousepad
In Step: 0
Eloqua Database
Cloud Services API
Eloqua API
Yes, here's a list of contacts waiting in Step #182: Contact X Contact Y Contact Z
Are there any contacts waiting in Step #182?
Hoovers
Find match in Hoovers DB
Can I get Contacts X,Y and Z's Email, Company Name, First and Last name
Update Contacts in the database with new information
Contact X: a@b.com, Dell, Bob, Johnston, Sr Director, Marketing
Contact Y: b@a.com, IBM, Mary, Santos, CMO, EMEA and APAC
Contact Z: cfo@c.com, News Corp, Larry, Sanders, manager, Marketing

Retrieving Contacts in Step

The first step in the code is to retrieve the contacts in the step. As mentioned, a second API is used for all Program Builder actions, so you will need to add another service reference to it. The WSDL for the Program Builder (Cloud Services) API is as follows:

https://secure.eloqua.com/api/1.2/ExternalActionService.svc?wsdl

In these code examples, I have referenced this as the EloquaProgramService namespace.

In a similar manner to the EloquaService, we need to create an instance of the service proxy and give it the appropriate user credentials:

```
private EloquaProgramService.ExternalActionServiceClient programServiceProxy;

strInstanceName = InstanceName;
strUserID = UserID;
strUserPassword = UserPassword;

programServiceProxy = new EloquaProgramService.ExternalActionServiceClient();
programServiceProxy.ClientCredentials.UserName.UserName = strInstanceName + "\\" +
strUserID;
programServiceProxy.ClientCredentials.UserName.Password = strUserPassword;
```

This can use the same set of user credentials as the previous EloquaService proxy, as long as the user credentials have the appropriate permissions for the API.

With this service proxy created, we can then list the contacts in the step:

```
List<EloquaContact> tmpContactsInStep = new List<EloquaContact>();
EloquaContact tmpContact;

EloquaProgramService.ExternalActionStatus status;
status = (EloquaProgramService.ExternalActionStatus)intStepStatus;

var result = programServiceProxy.ListMembersInStepByStatus(intPBStepID, status, 0, 1000);

foreach (var eam in result)
{
    if((int)eam.EntityType == 1)
    {
        tmpContact = new EloquaContact();
        tmpContact.ContactID = eam.EntityId;
        tmpContact.ExternalActionID = eam.Id;
        tmpContactsInStep.Add(tmpContact);
    }
}
```

In this example, I have created a very simple class of EloquaContact to store them in, but this class is not part of the API, I have defined it in this project as follows:

```
public class EloquaContact
{
    public int ContactID { get; set; }
    public string EmailAddress { get; set; }
    public string TwitterID { get; set; }
    public string PersonalizationField1 { get; set; }
    public string PersonalizationField2 { get; set; }
    public string TwitterMessage { get; set; }
    public int ExternalActionID { get; set; }
}
```

The code to retrieve the contacts uses an `EloquaProgramService.ExternalActionStatus` which has a value of 1 for Awaiting Action, 2 for In Progress, and 3 for Complete. Up to 1000 contacts at a time can be listed, this code starts from contact 0 and pulls 1000 at a time.

As I'm only interested in contacts, I check to capture only the contacts (EntityType of 1):

```
 if((int)eam.EntityType == 1)
```

We will want to set the status of the contacts in the step to "In Progress" so that it is known that we are processing the contacts.

```
public List<EloquaContact> SetStatusOfContactsInStep(int intProgramStepID, int
intOldStepStatus, int intNewStepStatus, List<EloquaContact> tmpContactsInStep)
{

    List<EloquaContact> tmpUpdatedContacts = new List<EloquaContact>();
    EloquaContact tmpContact;
    EloquaProgramService.ExternalActionStatus OldStatus;
    EloquaProgramService.ExternalActionStatus NewStatus;
    EloquaProgramService.Member[] stepMembers;
    OldStatus = (EloquaProgramService.ExternalActionStatus)intOldStepStatus;
    NewStatus = (EloquaProgramService.ExternalActionStatus)intNewStepStatus;
    EloquaProgramService.Member tmpMember;

    stepMembers = new EloquaProgramService.Member[tmpContactsInStep.Count()];

    for(int index = 0; index < tmpContactsInStep.Count(); index++)
    {
        tmpMember = new NameAnalyzerRole.EloquaProgramService.Member();

        tmpMember.EntityId =  tmpContactsInStep.ElementAt(index).ContactID;
        tmpMember.EntityType = (EloquaProgramService.EntityType)1;
        tmpMember.StepId = intProgramStepID;
        tmpMember.Status = OldStatus;
        tmpMember.Id = tmpContactsInStep.ElementAt(index).ExternalActionID;
        stepMembers[index] = tmpMember;
    }

    var results = programServiceProxy.SetMemberStatus(stepMembers, NewStatus);

    foreach (Member tmpUpdatedMember in results)
    {
        tmpContact = new EloquaContact();
        tmpContact.ContactID = tmpUpdatedMember.EntityId;
        tmpContact.ExternalActionID = tmpUpdatedMember.Id;
        tmpUpdatedContacts.Add(tmpContact);
    }

    return tmpUpdatedContacts;

}
```

Again, we use the status variable, but this time we use it both as a starting point and an ending point. If the status of any members of the step has changed since we began the process, setting a starting point of "Awaiting Action" will ensure they are missed (as they are likely being processed already). This allows a parallel environment where multiple servers are executing against the same step.

The results that are returned only show the contacts whose status was successfully changed. This is the list we can work with.

Now we have a list of contacts in the step, but they are only known by their ID, so the next step is to populate the key information on those contacts. The following basic procedure loops through the IDs in the list and populate them by batches:

```
public List<EloquaContact> GetContactsInList(List<int> tmpContactIDs)
{
    int intTotalContacts;
    int intCurrentContactCount;
    int intCurrentBatchSize;
    int intRetrieveBatchSize = 5;
    int intCurrentContactIndex = 0;
    int intCurrentArrayIndex = 0;

    List<EloquaContact> tmpFullContactList = new List<EloquaContact>();
    EloquaContact tmpContact;

    try
    {
        // Build a Contact Entity Type object
        EloquaService.EntityType entityType = new EloquaService.EntityType();

        entityType.ID = 0;
        entityType.Name = "Contact";
        entityType.Type = "Base";

        // Create a new list containing the fields you want populated
        List<string> fieldList = new List<string>();

        // Add the Contact's relevant fields to the field list

        fieldList.Add(strEmailAddressField);
        fieldList.Add(strFirstNameField);
        fieldList.Add(strTwitterIDField);
        fieldList.Add(strPersonalizationField1);
        fieldList.Add(strPersonalizationField2);

        // Build a Dynamic Entity array to store the results
        EloquaService.DynamicEntity[] retrievedEntities;

        // If the field list is empty - the request will return all Entity Fields
        // Otherwise, only fields defined in the field list are returned

        if (fieldList.Count > 0)
        {
            // Execute the request and return only the selected Entity fields
            // Set the ID of the Contact Entity
            intTotalContacts = tmpContactIDs.Count();
            intCurrentContactCount = 0;


            int[] ids;
            intCurrentContactCount = 0;
            intCurrentBatchSize = Math.Min((intTotalContacts - intCurrentContactCount),
intRetrieveBatchSize);

            while(intCurrentBatchSize > 0)
            {
                ids = new int[intCurrentBatchSize];
                intCurrentArrayIndex = 0;
                for (intCurrentContactIndex = intCurrentContactCount;
intCurrentContactIndex < (intCurrentContactCount + intCurrentBatchSize);
intCurrentContactIndex++)
                {
                    ids[intCurrentArrayIndex] =
tmpContactIDs.ElementAt(intCurrentContactIndex);
                    intCurrentArrayIndex++;
                }
                retrievedEntities = serviceProxy.Retrieve(entityType, ids,
fieldList.ToArray());

                if (retrievedEntities.Length > 0)
                {
                    foreach (EloquaService.DynamicEntity dynamicEntity in
retrievedEntities)
                    {
                        tmpContact = new EloquaContact();
                        tmpContact.ContactID = dynamicEntity.Id;
                        foreach (KeyValuePair<string, string> keyValPair in
dynamicEntity.FieldValueCollection)
                        {
                            if (keyValPair.Key == strEmailAddressField)
                                tmpContact.EmailAddress = keyValPair.Value;
                            if (keyValPair.Key == strFirstNameField)
                                tmpContact.FirstName = keyValPair.Value;
                            if (keyValPair.Key == strTwitterIDField)
```

```
                                tmpContact.TwitterID = keyValPair.Value;
                            if (keyValPair.Key == strPersonalizationField1)
                                tmpContact.PersonalizationField1 = keyValPair.Value;
                            if (keyValPair.Key == strPersonalizationField2)
                                tmpContact.PersonalizationField2 = keyValPair.Value;
                        }

                        tmpFullContactList.Add(tmpContact);
                    }
                }

                intCurrentContactCount += intCurrentBatchSize;
                intCurrentBatchSize = Math.Min((intTotalContacts - 
intCurrentContactCount), intRetrieveBatchSize);
            }//end batch retrieve while loop
        }
    }

    catch (System.ServiceModel.FaultException ex)
    {
        // Customize your own Error handling code.
        Console.WriteLine(String.Format("Reson: {0}", ex.Reason.ToString()));
        Console.WriteLine(String.Format("Fault Type: {0}", ex.GetType().ToString()));
        Console.WriteLine(String.Format("Fault Code: {0}", ex.Code.Name.ToString()));
    }
    catch (Exception ex)
    {
        // Customize your own Error handling code.
        Console.WriteLine(String.Format("Exception Message: {0}", 
ex.Message.ToString()));
    }

    return tmpFullContactList;
}
```

Now we have a fully populated (with the field we specified) list of contacts who are in the step. With this list we can then process the step actions and do what we need to do. In our case, we would send an appropriately formatted Tweet to the person in the step.

Once we have completed the task we wanted to complete, the only thing remaining is to then set the status of the contacts in the step to complete. The code to do this is identical to what was used above, only that we are now changing the status from In Progress (2) to Complete (3).

That is all that is needed. We have now created a fully functional Cloud Connector.

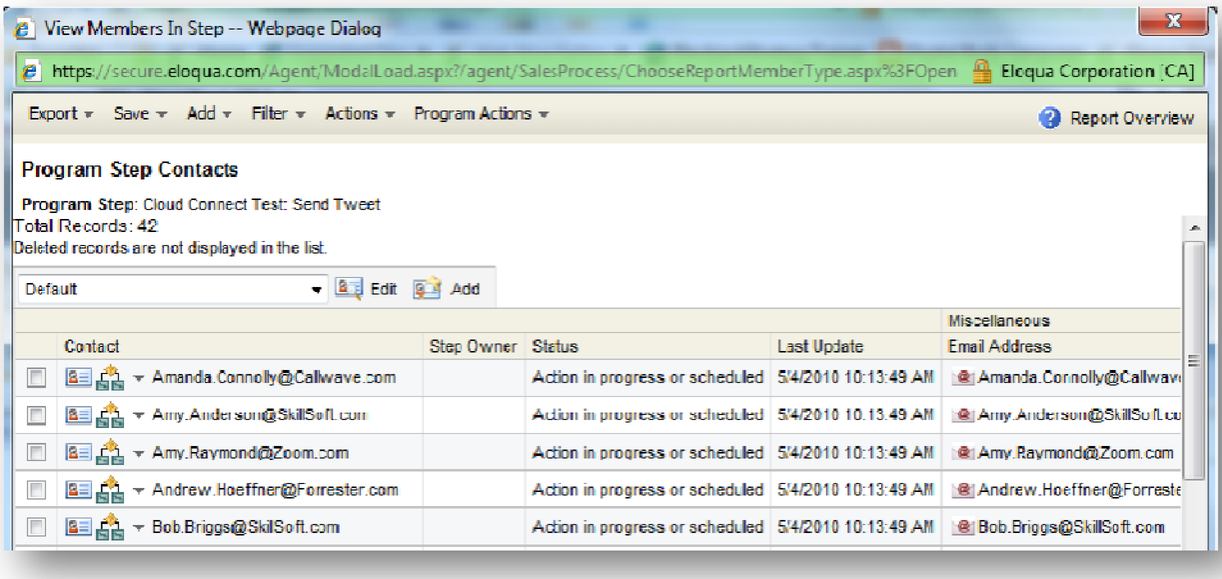
Tips and Tricks

Live Layer, and Viewing Contacts

When developing and testing your Cloud Connector, you will likely spend some time adding contacts to your sample Program, and running your Cloud Connector against the Program Step. As you do, you may notice that the list your code retrieves is not the same as the list that is displayed in Program Builder's "View Contacts In Step". The reason for this is simple.

Program Builder, when running, loads the contacts to be processed into a "Live" view, in memory, in order to allow faster processing. This happens every time Program Builder executes. If you have just added contacts to the step manually, but your code cannot see them, chances are Program Builder has not yet run and loaded them into the Live layer.

In Normal Mode, this will happen every 15 minutes, in Priority Mode, this will happen every 5 minutes.


View Members In Step -- Webpage Dialog
https://secure.eloqua.com/Agent/ModalLoad.aspx?/agent/SalesProcess/ChooseReportMemberType.aspx%3FOpen
Eloqua Corporation [CA]
Export
Save
Add
Filter
Actions
Program Actions
Report Overview
Default
Edit
Add

Program Step Contacts

Program Step: Cloud Connect Test: Send Tweet
Total Records: 42
Deleted records are not displayed in the list.

Contact	Step Owner	Status	Last Update	Miscellaneous Email Address
Amanda.Connolly@Callwave.com		Action in progress or scheduled	5/4/2010 10:13:49 AM	Amanda.Connolly@Callwav
Amy.Anderson@SkillSoft.com		Action in progress or scheduled	5/4/2010 10:13:49 AM	Amy.Anderson@SkillSoft.co
Amy.Raymond@Zoom.com		Action in progress or scheduled	5/4/2010 10:13:49 AM	Amy.Raymond@Zoom.com
Andrew.Hoeffner@Forrester.com		Action in progress or scheduled	5/4/2010 10:13:49 AM	Andrew.Hoeffner@Forreste
Bob.Briggs@SkillSoft.com		Action in progress or scheduled	5/4/2010 10:13:49 AM	Bob.Briggs@SkillSoft.com

Enabled Programs

As a related point, the Program must be running in order to have your Cloud Connector be able to see the contacts. If you do not find that you are able to see the contacts in the step, ensure that the Program is enabled.

Program Step Configuration

Once you have a completed Cloud Connector, and wish to have its configuration step connected to the user interface of Program Builder, contact Miles Thibault at Eloqua (miles.thibault@eloqua.com) with the URL and Cloud Connector details.

Conclusion

This guide walks through a simple version of a Cloud Connector, and provides the basic functionality needed to build your own. Please provide feedback as to any areas you found difficult or unclear in this document, or an future enhancements to the Connector infrastructure you would like to see.