

Plug-In

Microsoft Dynamics CRM plug-ins are one of the most commonly used and powerful approaches to extending the application. A plug-in for CRM is custom code, written and compiled in .Net, that is “triggered” when a specific event takes place within a specified CRM entity. The objective of custom plug-in code is to enhance or modify the standard features/behavior of CRM by injecting custom business logic into the execution of nearly any task a user performs in CRM.

Plugin code can be triggered to run when a record is created or updated or perhaps even when a group of records are queried. The Microsoft Dynamics CRM system contains a wide array of these events, commonly referred to as system “Messages”, from which custom code can be triggered. The Messages (which are technically web service operations) are constantly available to run as CRM recognizes events within the platform and sends requests to the respective message for the custom logic to interact with your data.

The plug-in code can be written to run before or after, say, a save operation is completed, thus allowing the developer to work with the context of data when writing the new business logic. This allows a programmer to read and analyze the current data and environment to help design the custom business logic.

When plug-ins are triggered through the messaging system, they can be configured to run in various **Steps** of the execution pipeline within CRM: pre-event and pre-validation, pre-event and pre-operation, and post-event. The step is the “when” of the plug-in execution. These stages are important factors in designing how a plug-in can create the desired outcome for custom business logic.

Another critical concept behind plug-ins is the **Image**. An image, created when a plug-in is registered within CRM, is a “snapshot” of an entities data context before (pre-image) or after (post-image) the execution of the operation which triggered the plug-in. Again, access to the fields and values of an image provide the developer with valuable context needed when writing the custom business logic. While plug-ins are a .Net assembly, they are unique in that are “registered” within the Microsoft Dynamics CRM framework. There are a number of important concepts to understand when writing plug-ins

- Understanding data context that is available to plug-ins when run through different messages and at different stages.
- Using Steps and Images when registering plug-ins
- Exception handling within plug-ins ([link](#))
- Using Impersonation in plug-ins
- Challenges with debugging a plug-in

When to Use Plug-ins

Plug-ins can be used to achieve some of the similar goals as with workflows or client-side code such as JavaScript, Silverlight, or HTML.

Plug-in vs. Client-side Code

The differentiation is clear. Plug-in runs server-side with compiled code and should always be the preferred method of implementing business logic. Client-side code is vulnerable to different client

environment, browsers, browser settings, etc, and should not be relied on implementing business critical steps.

On the other hand, anything requiring UI or user interaction, must be done with client side code. You will not be able to modify UI or prompt the user during processing with the plug-in.

Synchronous vs. Asynchronous Plug-ins

Synchronous plug-ins are executed by the CRM Core System. Synchronous execution means that triggering event will wait until the plug-in finishes the execution. For example, if we have synchronous plug-in that triggers on the creation of account and user creates account record in CRM, the form will hang at save until the plugin has finished.

Asynchronous plug-ins are executed by asynchronous service. Asynchronous plug-ins allow triggering event to finish before plug-in code runs. Therefore these can never be used to prevent an action, validate data entry, or provide any error messages back to the user.

Event Pipeline

The event pipeline allows you to configure when in the event the plug-in code will execute. The event pipeline is divided into the following events and stages:

Pre-event/Pre-Validation

This stage executes before anything else, even before basic validation if the triggering action is even allowed based on security. Therefore, it would be possible to trigger the plug-in code even without actually having permission to do so and great consideration must be used when writing a pre-validation plug-in. Also, execution in this stage might not be part of the database transaction.

Example uses:

Some “delete” plug-ins. Deletion cascades happen prior to pre-operation, therefore if you need any information about the child records, the delete plugin must be pre-validation.

Pre-event/Pre-Operation

This stage executes after validation, but before the changes has been committed to database. This is one of the most commonly used stages.

Example uses:

If and “update” plug-in should update the same record, it is best practice to use the pre-operation stage and modify the properties. That way the plug-in update is done within same DB transaction without needing additional web service update call.

Platform Core Operation

This stage is the in-transaction main operation of the system, such as create, update, delete, and so on. No custom plug-ins can be registered in this stage. It is meant for internal use only.

Post-Event

This stage executed after changes have been committed to database. This is one of the most used stages.

Example uses:

Most of the “Create” plugins are post-event. This allows access to the created GUID and creation of relationships to newly created record.

Post-Event/Post-Operation (deprecated)

This stage only supports Microsoft Dynamics CRM 4.0 based plug-ins.

Plug-in Messages

Plug-in Message is the triggering event, such as **Update** or **Create**. There are hundreds of these. The CRM SDK has a list of them all in SDK/message-entity support for plug-ins.xlsx. Here are some of the most commonly used messages:

Create

Triggered when the record is created. All set attribute values are included in the target entity.

Update

Triggered when the record is updated. You can define which attribute updates trigger the plug-in, by defining the filtering attributes. Also, only the updated attribute values are included in the target entity (see images)

Delete

Triggered when the record is deleted.

Retrieve

Triggered when record is retrieved, for example when user opens a record on a form.

RetrieveMultiple

Triggered when a record set is retrieved using RetrieveMultiple, for example showing a view.

Plug-in Images (Pre vs. Post)

Images are snapshots of the entity's attributes, before and after the core system operation. Following table shows when in the event pipeline different images are available:

Message	Stage	Pre-Image	Post-Image
Create	PRE	No	No
Create	POST	No	Yes
Update	PRE	Yes	No
Update	POST	Yes	Yes
Delete	PRE	Yes	No
Delete	POST	Yes	No

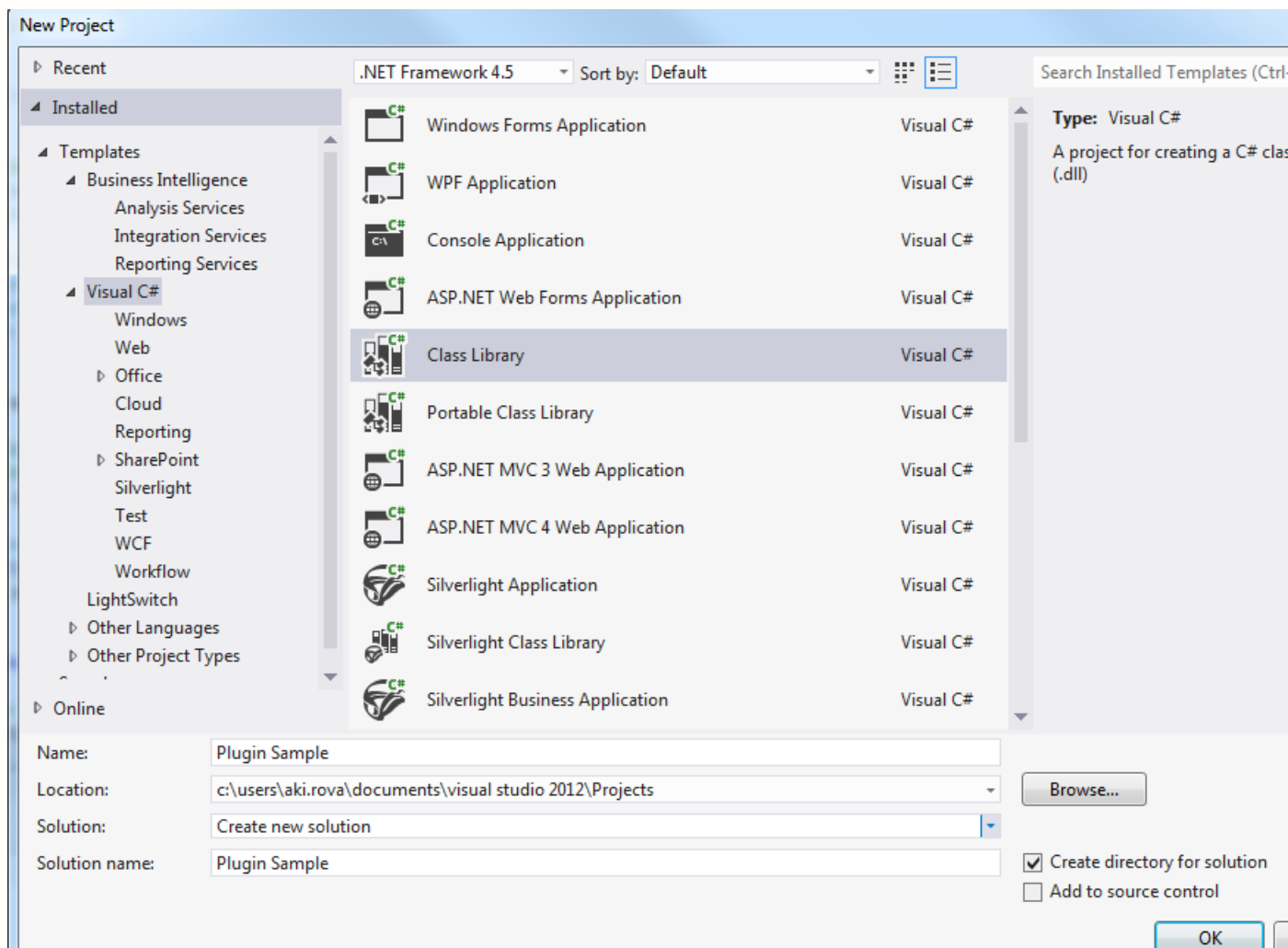
The benefits of images

- One of the best uses for this is in update plug-ins. As mentioned before, update plug-in target entity only contains the updated attributes. However, often the plug-in will require information from other attributes as well. Instead of issuing a retrieve, the best practice is to push the required data in an image instead.
- Comparison of data before and after. This allows for various audit-type plugins, that logs what the value was before and after, or calculating the time spent in a stage or status.

Developing a Plug-in

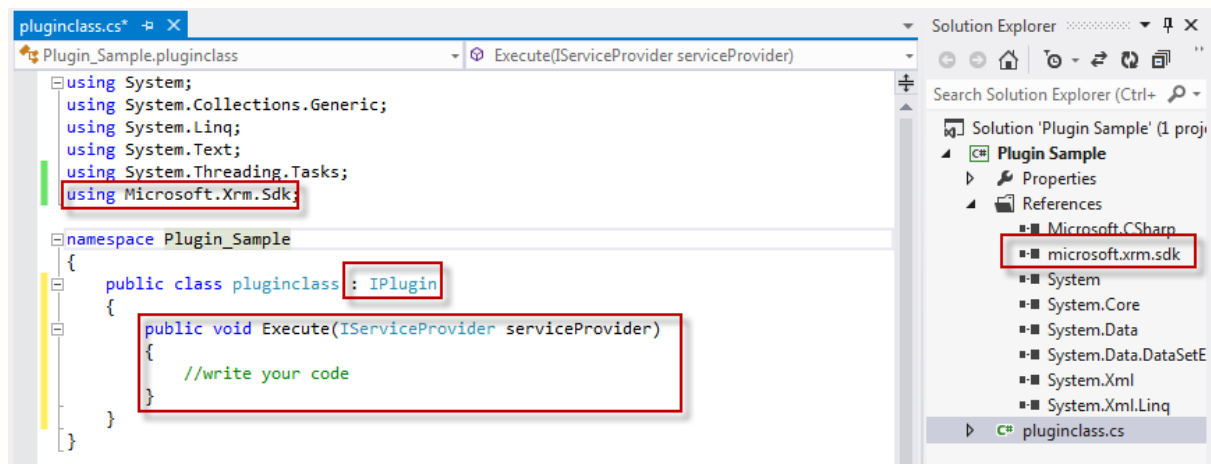
How to Start Developing a CRM Plug-in

1. Download CRM SDK. This will provide you all the information, required SDK assemblies and many helpful samples.
2. Set up your plug-in project in Visual Studio. This will be a .NET class library.

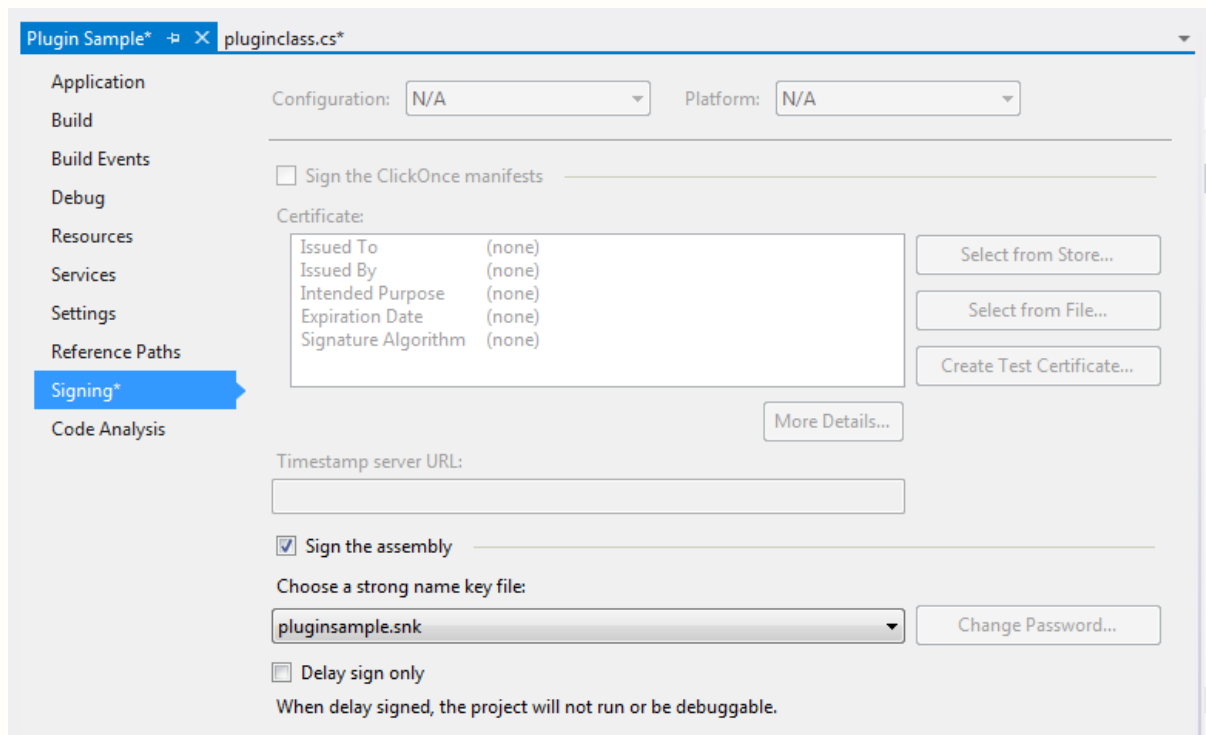


3. Add References. At minimum, you will need Microsoft.Xrm.Sdk, obtained from CRM SDK.
4. Extend the class from Microsoft.Xrm.Sdk.IPlugin

5. Write your code



6. At the project, sign the assembly. This is required in order to be able to deploy the plugin.



7. Compile the assembly and deploy using Plugin Registration Tool.

Example 1: Update parent record based on multiple fields

Here's an extremely simplified example of a plug-in. We have parent record with a field for total, and a single child with unit and rate. When either unit or rate changes, we want to multiply these and update to the total of the parent.

Design Notes

- Because user may only update units and not update rate, we cannot use target which would only include the attributes that were actually updated. Therefore, we use image to make sure we get the necessary attributes, without having to do additional data retrieve.

- The plug-in is post-update, so we will have access to post-image, showing all values like they are after update.
- When registering the plugin, we set filtering attributes to include only rate and units. Therefore, plugin will not trigger needlessly if something unrelated updates.
- When setting up the Image, we only need rate, units and parent ID.

Plug-in Code

```
using System;
using Microsoft.Xrm.Sdk;
namespace CRMBook
{
    public class Update : IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            // Obtain the execution context from the service provider.
            IPluginExecutionContext context =
                (IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));
            // Get a reference to the Organization service.
            IOrganizationServiceFactory factory =
                (IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));
            IOrganizationService service = factory.CreateOrganizationService(context.UserId);
            if (context.InputParameters != null)
            {
                //entity = (Entity)context.InputParameters["Target"];
                //Instead of getting entity from Target, we use the Image
                Entity entity = context.PostEntityImages["PostImage"];
                Money rate = (Money)entity.Attributes["po_rate"];
                int units = (int)entity.Attributes["po_units"];
                EntityReference parent = (EntityReference)entity.Attributes["po_parentid"];
                //Multiply
                Money total = new Money(rate.Value * units);
                //Set the update entity
                Entity parententity = new Entity("po_parententity");
                parententity.Id = parent.Id;
                parententity.Attributes["po_total"] = total;
                //Update
                service.Update(parententity);
            }
        }
    }
}
```

```

    }
}
}

```

Example 2: Update same record based on multiple fields

This example is similar to the one above, but let's assume our total is on same entity.

- In order to change the total on the fly, the plug-in needs to be pre-update
- This means we do not have access to post image, and have to use the pre-image instead. Therefore, we need to get all the changed values from the target, and unchanged values from the image.

Plug-in Code

```

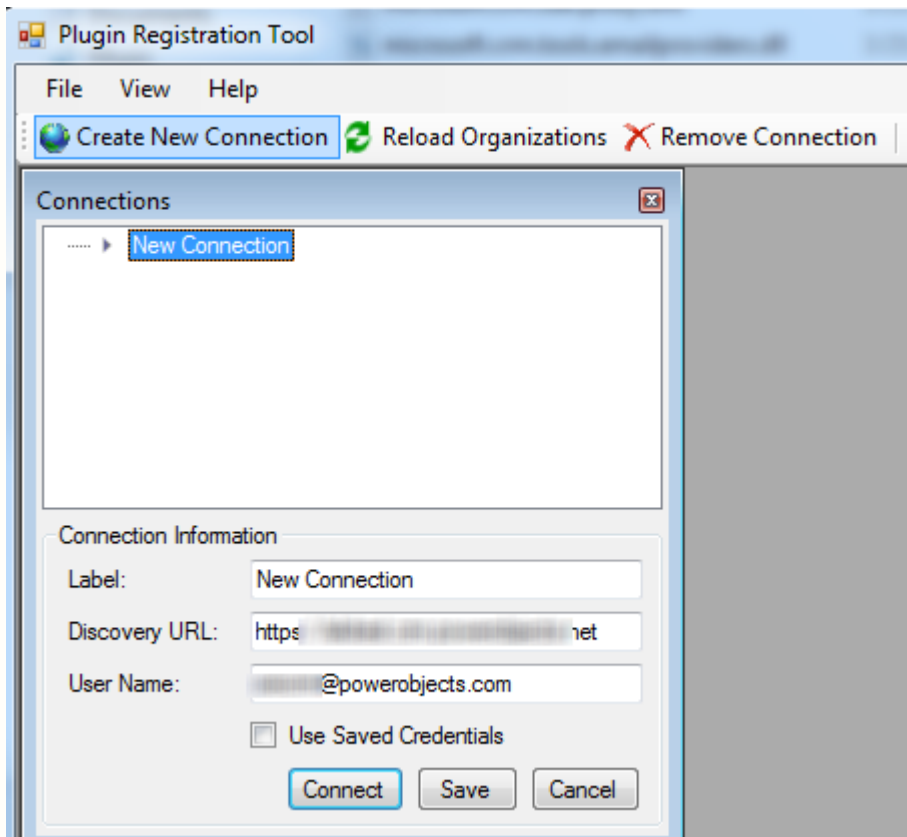
if (context.InputParameters != null)
{
    //Get Target - includes everything changed
    Entity entity = (Entity)context.InputParameters["Target"];
    //Get Pre Image
    Entity image = context.PreEntityImages["PreImage"];
    //If value was changed, get it from target.
    //Else, get the value from preImage
    Money rate = null;
    if(entity.Attributes.Contains("po_rate"))
        rate = (Money)entity.Attributes["po_rate"];
    else
        rate = (Money)image.Attributes["po_rate"];
    int units = 0;
    if (entity.Attributes.Contains("po_units"))
        units = (int)entity.Attributes["po_units"];
    else
        units = (int)image.Attributes["po_units"];
    //Multiply
    Money total = new Money(rate.Value * units);
    //Set the value to target
    entity.Attributes.Add("po_total",total);
    //No need to issue additional update
}

```

Registering and Deploying Plug-ins

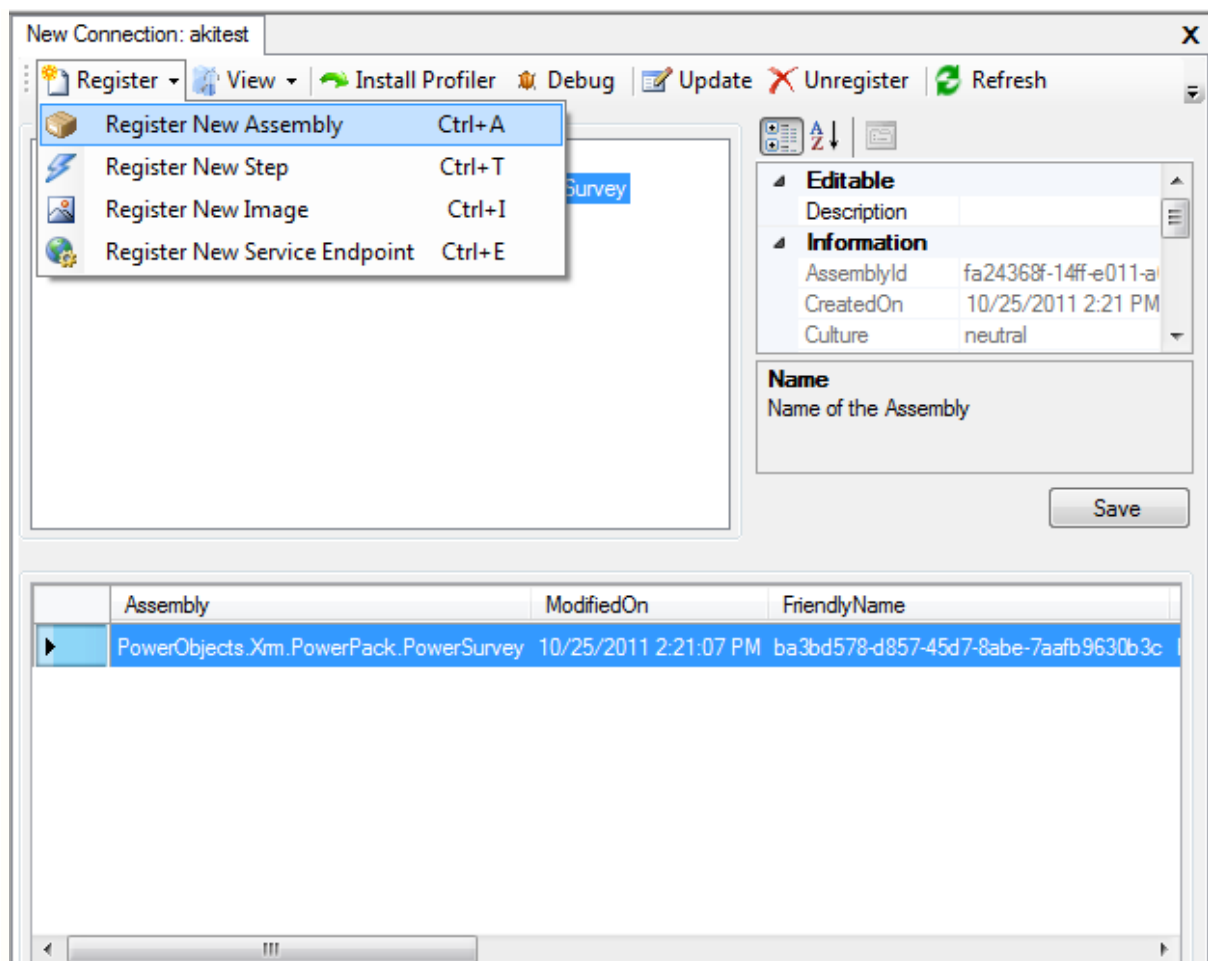
Registering and deploying plug-ins can be done using the plug-in registration tool. The tool is available in CRM SDK. Here's how to do it:

1. Connect to your organization.



If you have access to multiple organizations in the server, choose the one to connect to.

2. Register a new assembly.



3. Browse the assembly file, select **Isolation Mode** and specify where the assembly is stored.

Register New Plugin

Step #1: Specify the Location of the Assembly to Analyze

Load Assembly

Step #2: Select the Plugins & Workflow Activities to Register

☒ Select All / Deselect All

Step #3: Specify the Isolation Mode

☒ **Sandbox**
All code in this assembly will be run in a secure sandbox (reduced functionality)

☐ **None**

Step #4: Specify the Location where the Assembly should be stored

☒ **Database**
Assembly is stored and loaded from the database. For debugging purposes, the Symbols (.PDB files) must be in \Server\bin\assembly of the main installation folder for each server that needs to be debugged.

☐ **Disk**
Assembly is stored and loaded from \Server\bin\assembly in the main installation directory for each server. For debugging purposes, the Symbols (.PDB files) must be located in the same place as the assembly.

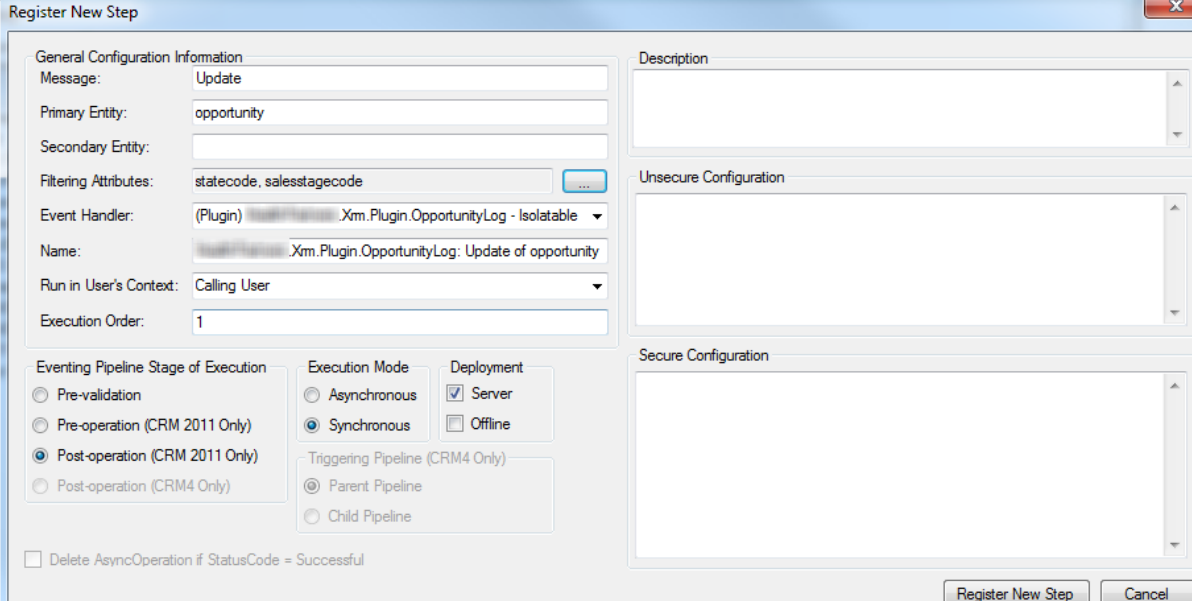
File Name on Server:

☐ **GAC**
File is placed in the GAC of each server where it will used.

Log

Register Selected Plugins Close

4. Next you'll need to select the registered assembly. It can contain multiple plug-ins. Select the plugin you are adding steps to, and register one or more steps.



The 'Register New Step' dialog box is used to configure a new step in a CRM pipeline. It contains several sections:

- General Configuration Information:**
 - Message: Update
 - Primary Entity: opportunity
 - Secondary Entity:
 - Filtering Attributes: statecode, salesstagecode
 - Event Handler: (Plugin)Xml.Plugin.OpportunityLog - Isolatable
 - Name:Xml.Plugin.OpportunityLog: Update of opportunity
 - Run in User's Context: Calling User
 - Execution Order: 1
- Event Pipeline Stage of Execution:**
 - ☐ Pre-validation
 - ☐ Pre-operation (CRM 2011 Only)
 - ☒ Post-operation (CRM 2011 Only)
 - ☐ Post-operation (CRM 4 Only)
- Execution Mode:**
 - ☐ Asynchronous
 - ☒ Synchronous
- Deployment:**
 - ☒ Server
 - ☐ Offline
- Triggering Pipeline (CRM 4 Only):**
 - ☒ Parent Pipeline
 - ☐ Child Pipeline
- ☐ Delete AsyncOperation if StatusCode = Successful
- Description:** (Empty text area)
- Unsecure Configuration:** (Empty text area)
- Secure Configuration:** (Empty text area)
- Buttons: Register New Step, Cancel

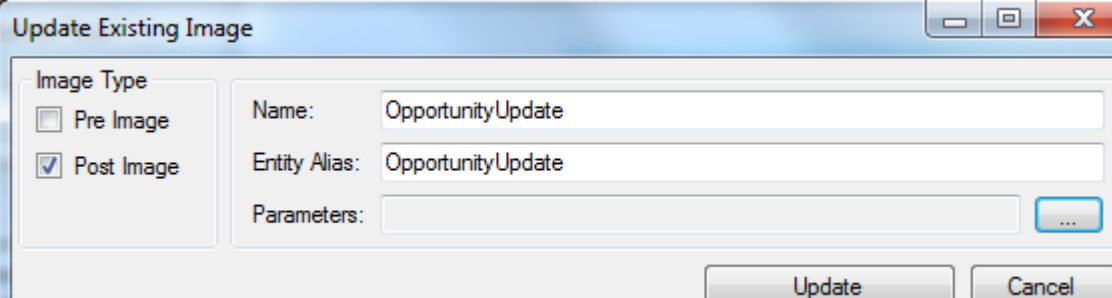
5. Fill in the following information for the steps:

- Message
- Entity
- Filtering Attributes if applicable. In above example, the plugin will only trigger for statecode or salesstagecode updates. Selecting the attributes will prevent plugin triggering accidentally or needlessly when unrelated field is updated.
- Event Pipeline
- Execution Mode

6. Fill in the **Unsecure Configuration/Secure Configuration** sections. These sections can be used to pass configuration information to the plug-in, such as user credentials or URLs. The difference between secure and unsecure configuration is as follows:

- Secure Configuration does not move with solutions. It has to be re-configured for each environment.
- Secure Configuration can be viewed only by CRM administrators.

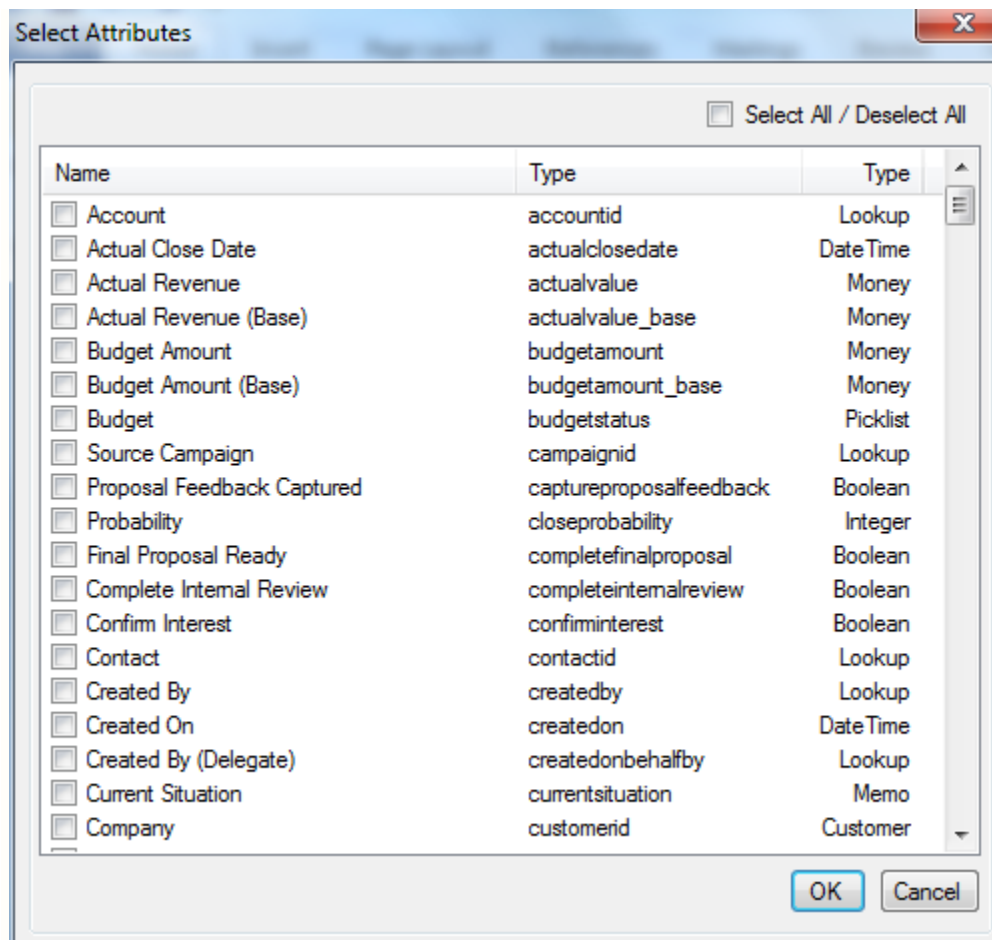
7. If applicable, select step and register an image. Choose whether it's a **Pre Image** or **Post Image**.



The 'Update Existing Image' dialog box is used to update an existing image in a CRM pipeline. It contains the following fields:

- Image Type:**
 - ☐ Pre Image
 - ☒ Post Image
- Name:** OpportunityUpdate
- Entity Alias:** OpportunityUpdate
- Parameters:** (Empty text area with a button to select parameters)
- Buttons: Update, Cancel

You'll also need to select attributes that you would like to be contained within the image.



8. After the plug-in and the steps have been registered, they can now be included in CRM solutions and deployed with unmanaged or managed solutions.