**Assign a record to a Team using C# in Dynamics CRM 2015**

//Assign a record to a team

 private void AssignRecord(Entity TargetEntity, Guid TargetRecordID, Guid OwningTeamID, IOrganizationService orgService)

 {

     try

     {

         // Create the Request Object and Set the Request Object's Properties

         AssignRequest assign = new AssignRequest

         {

             Assignee = new EntityReference("team", OwningTeamID),

             Target = new EntityReference(TargetEntity.LogicalName, TargetRecordID)

         };

         // Execute the Request

         orgService.Execute(assign);

     }

     catch (Exception ex)

     {

         throw new Exception("An error occured while assinging Team to a record." + ex.Message);

     }

 }

**//Share a record Read,Write and Append privileges with Team**

private void SharePrivileges(string targetEntityName, Guid targetRecordID, Guid teamID, bool read\_Access, bool write\_Access, bool append\_Access, IOrganizationService orgService)

{

    try

    {

        //Get User or Team reference and Target Entity and record ID that needs to be shared.

        var recordRef = new EntityReference(targetEntityName, targetRecordID);

        var teamRef = new EntityReference("team", teamID);

        AccessRights Access\_Rights = new AccessRights();

        Access\_Rights = AccessRights.None;

        //Read Access

        if (read\_Access == true)

            Access\_Rights = AccessRights.ReadAccess;

        //Write Access (or) Read, Write Access

        if (write\_Access == true)

            if (Access\_Rights == AccessRights.None)

                Access\_Rights = AccessRights.WriteAccess;

            else

                Access\_Rights = Access\_Rights | AccessRights.WriteAccess;

        //Append Access or all or any two accesses

        if (append\_Access == true)

            if (Access\_Rights == AccessRights.None)

                Access\_Rights = AccessRights.AppendToAccess | AccessRights.AppendAccess;

            else

                Access\_Rights = Access\_Rights | AccessRights.AppendToAccess | AccessRights.AppendAccess;

        var grantAccess = new GrantAccessRequest

        {

            PrincipalAccess = new PrincipalAccess

            {

                AccessMask = Access\_Rights,

                Principal = teamRef

            },

            Target = recordRef

        };

        // Execute the Request

        orgService.Execute(grantAccess);

    }

    catch (Exception ex)

    {

        throw new Exception("An error occured while applying Sharing rules for the record." + ex.Message);

    }

}

//Code to modify privileges for the target record and team

private void ModifyAccess(string targetEntityName, Guid targetRecordID, Guid teamID, IOrganizationService orgService)

{

    try

    {

        //Get User or Team reference and Target Entity and record ID that needs to be shared.

        var RecordReference = new EntityReference(targetEntityName, targetRecordID);

        var teamRef = new EntityReference("team", teamID);

        AccessRights accessRights = new AccessRights();

        accessRights = AccessRights.DeleteAccess;

        var modifyAcess = new ModifyAccessRequest

        {

            PrincipalAccess = new PrincipalAccess

            {

                AccessMask = accessRights,

                Principal = teamRef

            },

            Target = RecordReference

        };

        // Execute the Request

        orgService.Execute(modifyAcess);

    }

    catch (Exception ex)

    {

        throw new Exception("An error occured in Modifying access." + ex.Message);

    }

}

//Code to remvove the sharing privileges for the target record and team

private void RevokeAccess(string targetEntityName, Guid targetRecordID, Guid teamID, IOrganizationService orgService)

{

    try

    {

        //Get User or Team reference and Target Entity and record ID that needs to be shared.

        var recordRef = new EntityReference(targetEntityName, targetRecordID);

        var teamRef = new EntityReference("team", teamID);

        var revokeAcess = new RevokeAccessRequest

        {

            Revokee = teamRef,

            Target = recordRef

        };

        // Execute the Request

        orgService.Execute(revokeAcess);

    }

    catch (Exception ex)

    {

        throw new Exception("An error occured in Revoking access." + ex.Message);

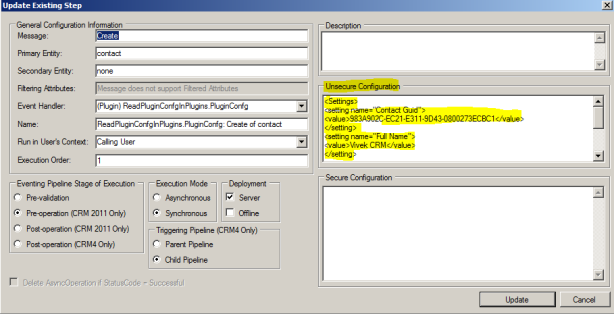
    }

}

**Pass parameters to Plugin Using Secure or Unsecure Configuration to Plugin:**

When you start developing plugins, you often need an input parameter or a configuration for the plugin execution which can be easily updated without having to re-compile and/or re-register the plugin.

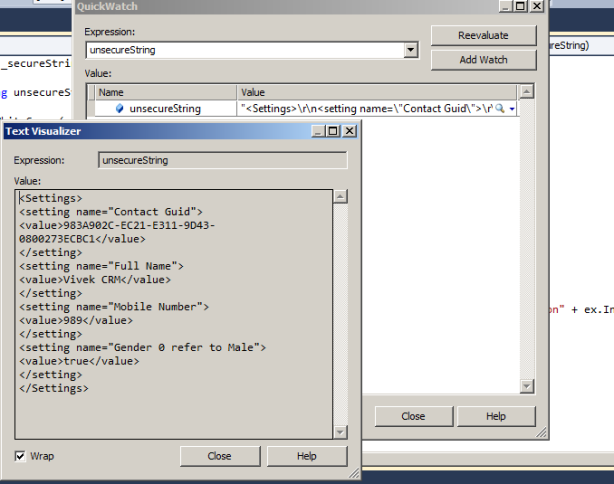
For example , i want to read some of the data which may change every time. Rather than creating the custom entity to read the Configuration kind of thing better to use this method to read the Parameters.

**Use the plugin step “Configuration”**  
When you register a plugin step, there is a field where you can specify some configuration parameters for the plugin execution as below:[](https://srmscrm.files.wordpress.com/2013/10/image.png)

Then in the Constructor of your plugin class you will get the configuration value which you can use later in the Execute method:

[](https://srmscrm.files.wordpress.com/2013/10/image2.png)

In the Quickwatch you can watch the total configuration as follows:

[](https://srmscrm.files.wordpress.com/2013/10/image1.png)

have a glance below for plugin code:

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Microsoft.Xrm.Sdk;  
using System.Xml;

namespace ReadPluginConfgInPlugins  
{  
public class PluginConfg : IPlugin  
{  
private readonly string \_unsecureString;  
private readonly string \_secureString;

public PluginConfg(string unsecureString, string secureString)  
{  
if (String.IsNullOrWhiteSpace(unsecureString) || String.IsNullOrWhiteSpace(secureString))  
{  
try  
{  
XmlDocument doc = new XmlDocument();  
doc.LoadXml(unsecureString);  
Guid DefaultQueueGuid = PluginConfiguration.GetConfigDataGuid(doc, “Contact Guid”);  
string ContactFullName = PluginConfiguration.GetConfigDataString(doc, “Full Name”);  
int MobileNumber = PluginConfiguration.GetConfigDataInt(doc, “Mobile Number”);  
bool Gender = PluginConfiguration.GetConfigDataBool(doc, “Gender 0 refer to Male”);  
}  
catch (Exception ex)  
{  
throw new Exception(“SoapException” + ex.Message + “########” + ex.StackTrace + “$$$$Inner Exception” + ex.InnerException);  
}  
}   
}

public void Execute(IServiceProvider serviceProvider)  
{  
//Extract the tracing service for use in debugging sandboxed plug-ins.  
ITracingService tracingService =  
(ITracingService)serviceProvider.GetService(typeof(ITracingService));

// Obtain the execution context from the service provider.  
IPluginExecutionContext context = (IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));

// For this sample, execute the plug-in code only while the client is online.   
tracingService.Trace(“AdvancedPlugin: Verifying the client is not offline.”);  
if (context.IsExecutingOffline || context.IsOfflinePlayback)  
return;

// The InputParameters collection contains all the data passed   
// in the message request.  
if (context.InputParameters.Contains(“Target”) &&  
context.InputParameters[“Target”] is Entity)  
{  
Entity entity = (Entity)context.InputParameters[“Target”];

}

}  
}

}

For this approach also have a Pros and Cons.

PROS:

* The step configuration is solution-aware so it will be automatically transported with the plugin step.

CONS:

* You need to use the plugin registration tool or another application to update the step configuration.
* The configuration is step-specific so you have to provide it and/or update it for every step even if the value is the same for all the steps (the configuration is on each step instead of the assembly or plugin type).
* the configuration is just an attribute of the plugin step so you cannot control privileges on the configuration independently from privileges on plugin step entity.

**\*. Use the plugin step “Secure Configuration”**  
This is similar to the step Configuration except that the configuration data is stored in a separate entity which can be secured.  
PROS:

* The configuration data can be secured as any other entity using the CRM security model. This is useful when the configuration contains sensitive information such as passwords.

CONS:

* Secure configuration is not solution aware so you will need to configure it for each environment.
* You need to use the plugin registration tool or another application to update the step configuration.
* The configuration is step-specific so you have to provide it and/or update it for every step even if the value is the same for all the steps (the configuration is on each step instead of the assembly or plugin type).

**\*. Use a Web Resource**  
You can store configuration information in web resources, for example you might have some XML configuration stored in a web resource and have your plugin read the web resource each time it executes.

PROS:

* Web resources are solution aware and the GUID is preserved across environments so you can hardcode the web resource GUID in your plugin code. You can transport the web resource and the plugin in the same solution.
* Can be easily updated using the CRM UI.

CONS:

* You cannot secure the configuration since it depends on the web resource access privileges and most users will need at least read access to web resources.

Hope this may help you.

Regards,

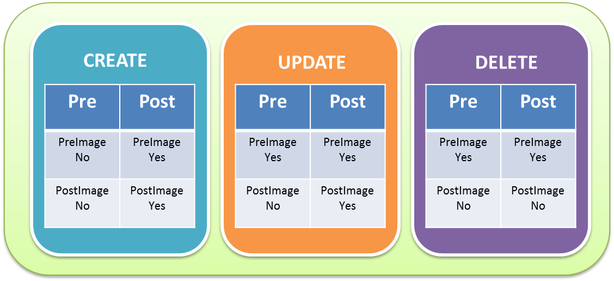
[**Plugin Images Support – MS CRM 2011Plugins**](https://srmscrm.wordpress.com/2013/05/30/plugin-images-support-ms-crm-2011plugins/)

Posted: May 30, 2013 in [Ms Crm 2011](https://srmscrm.wordpress.com/category/ms-crm-2011/), [Ms Crm 2011 PlugIns](https://srmscrm.wordpress.com/category/ms-crm-2011/ms-crm-2011-plugins/)

[0](https://srmscrm.wordpress.com/2013/05/30/plugin-images-support-ms-crm-2011plugins/#respond)

**Plugin Images**

Have a glance below Image for quick reference .

[](https://srmscrm.files.wordpress.com/2013/05/entity-plugin-images.png)

**PreEntityImages** :(Gets the properties of the primary entity before the core platform operation has begins)

The **PostEntityImages** contains the attributes value which are finally changed. We can capture the changed data before the database operation takes place. And can do any kind of validation based on the changed data.

# Diff between context.userid and context.initiatinguserid In MS CRM 2011 Plugin

Situation: A plugin running from a impersonated context

Aim: Get the GUID of the user who has actually fired the plugin and not the GUID of the user under whose context the plugin is registered.

The context gives us both the GUIDs

* context.initiatinguserid: gets the systemuser GUID who actually fired the plugin
* context.userid : gets the  impersonated systemuser GUID
* For a plugin that is executed the IExecutionContext consists of 2 id's: UserId and InitiatingUserId

UserId - "Gets the global unique identifier of the system user for whom the plug-in invokes Web service methods on behalf of"

InitiatingUserId - "Gets the global unique identifier of the system user account under which the current pipeline is executing. "

Suppose I have an Update operation (pre or post) and that is registered with the following setting:

 "Run in User's Context" as "Calling User"

When the plugin is fired the value of the UserId seems to be that of the "SYSTEM" user. Is this account that is related to the application pool identity on the IIS namely the "NETWORK SERVICE" account?

How do we change this to another account? I am confused as to what is meant by impersonation..

If in your plugin step registration "Run in User's Context" field has the value "Calling User" then the UserId and InitiatingUserId will match.

If you specify a user in the "Run in User's Context" field, then the UserId field will contain the user ID of the person you specify and the InitiatingUserId will be the actual [CRM](http://www.toplinestrategies.com/blogs/sales-management/crm) user whose action triggered the plugin.

My bet is that most people are usually looking for the InitiatingUserId, so it probably makes sense to use that property even if you specify "Calling User" for the "Run in User's Context" field. That way, if the "Run in User's Context" value ever changes, InitiatingUserId will still have the correct value.

If you use "UserID", it's possible you'll get a different value than what you expect. Hopefully this helps someone figure out which value is the correct one for your situation.

**UserId & InitiatingUserId properties in Plugin of CRM**

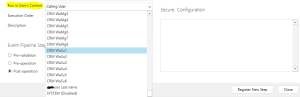
August 19, 2012[Rajeev Pentyala](https://rajeevpentyala.com/author/rajeevpentyala/)[Leave a comment](https://rajeevpentyala.com/2012/08/19/userid-initiatinguserid-properties-in-plugin-of-crm/#respond)[Go to comments](https://rajeevpentyala.com/2012/08/19/userid-initiatinguserid-properties-in-plugin-of-crm/#comments)

In CRM plugin, “*IExecutionContext*” contains 2 properties

* UserId
  + *Gets the GUID of the user for whom the plug-in invokes “on behalf of”.*
* InitiatingUserId
  + *Gets the GUID of the user under which the current pipeline is executing.*

Consider a scenario

* You have a user “RAJ” with “Sales Person” role with only “User Level” “Read” privilege on ‘Contact’
* You have a plugin on Post Deletion of ‘Contact’ with name “PostContactDelete”
* Assume in one particular scenario user “RAJ” should be able to delete a ‘Contact’
* So you can run the “PostContactDelete” plugin in the user with “SystemAdministrator” role
  + (i.e., Set “Run in User’s Context” to User with admin role; In sample screen shot below I chosen my admin user whose name is  ‘CRM WaSu1)

[](https://rajeevpentyala.files.wordpress.com/2012/08/run-in-user-context.png)

Run in User Context

* When User “RAJ” logs in and try to delete ‘Contact’ the plug-in “PostContactDelete” fires. When you debug
  + IExecutionContext.UserId = GUID of  SystemAdministrator (i.e., OnBehalfOf User ‘RAJ’)
  + IExecutionContext. InitiatingUserId =GUID of  RAJ   (i.e., Actual User)

**Reading LinkEntity attributes using Query Expression in CRM 2011**

October 25, 2012[Rajeev Pentyala](https://rajeevpentyala.com/author/rajeevpentyala/)[Leave a comment](https://rajeevpentyala.com/2012/10/25/reading-linkentity-attributes-using-query-expression-in-crm-2011/#respond)[Go to comments](https://rajeevpentyala.com/2012/10/25/reading-linkentity-attributes-using-query-expression-in-crm-2011/#comments)

Assume you have to retrieve

* All the “Contact” records with fields  “Full Name, Address1\_City”
* Along with ‘Account Number’ field of ‘Account’ whom the contacts are associated with.

**Relationship between Contact & Account**

In CRM, there is out of the box 1:N relationship between Account & Contact.

* Contact has ‘ParentCustomerId’ field as foreignkey of ‘Account’ entity
* ‘Account’ entity’s primary field is ‘AccountId’

**Preparing Query Expression**

* To get the ‘Contact’ attributes along with LinkEntity (i.e.,Account) attributes (i.e., Account Number),  we will write a Query Expression with (Entity as ‘Contact’ and LinkEntity as ‘Account’)
* *To read LinkEntity attribute’s we have to read the attribute along with ‘EntityAlias’ of LinkEntity (i.e.,* **EntityAlias.AttributeName***)*

Below is the query expression

 var query = new QueryExpression(“contact”);

var columnNames = new[] { “fullname”,”address1\_city” };

query.ColumnSet = new ColumnSet(columnNames);

// ‘Account’ as LinkEntity

var colsAccount = new[] { “accountnumber” };

LinkEntity linkEntityAccount = new LinkEntity() {

LinkFromEntityName = “contact”,

LinkFromAttributeName = “parentcustomerid”,

LinkToEntityName = “account”,

LinkToAttributeName = “accountid”,

JoinOperator = JoinOperator.Inner,

Columns = new ColumnSet(colsAccount),

EntityAlias = “aliasAccount”

};

query.LinkEntities.Add(linkEntityAccount);

// Execute Query using RetrieveMultiple

EntityCollection contacts = this.service.*RetrieveMultiple*(query);

if (contacts != null) {

foreach (var targetEntity in contacts.Entities) {

// Read “Account Number” along with Alias

var accountNumber = getAttributeValue(targetEntity, “aliasAccount.accountnumber“);

var contactFullname = getAttributeValue(targetEntity, “fullname”);

}

}

/// <summary>

/// Generic function to get value from attribute

/// </summary>

private string **getAttributeValue**(Entity targetEntity, string attributeName) {

if (string.IsNullOrEmpty(attributeName)) {

return string.Empty;

}

if (targetEntity[attributeName] is AliasedValue) {

return (targetEntity[attributeName] as AliasedValue).Value.ToString();

}

else {

return targetEntity[attributeName].ToString();

}

}

return string.Empty;

}

Sample Plug-in: Compare Pre and Post images on Update

<http://sumedha8.blogspot.in/2012/10/sample-plug-in-compare-pre-and-post.html>

<http://www.toplinestrategies.com/blogs/net/dynamics-crm-plugins-how-determine-updated-fields>

only synchronous post-event and asynchronous registered plug-ins have [OutputParameters](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.outputparameters.aspx) populated as the response is the result of the core platform operation. The property is of type [ParameterCollection](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.parametercollection.aspx) where the keys to access the response data are the names of the actual public properties in the response.

**Pre and post entity images**

[PreEntityImages](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.preentityimages.aspx) and [PostEntityImages](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.postentityimages.aspx) contain snapshots of the primary entity's attributes before (pre) and after (post) the core platform operation. Microsoft Dynamics CRM populates the pre-entity and post-entity images based on the security privileges of the impersonated system user. Only entity attributes that are set to a value or are **null** are available in the pre or post entity images. You can specify to have the platform populate these **PreEntityImages** and **PostEntityImages** properties when you register your plug-in. The entity alias value you specify during plug-in registration is used as the key into the image collection in your plug-in code.

There are some events where images aren’t available. For example, only synchronous post-event and asynchronous registered plug-ins have [PostEntityImages](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.postentityimages.aspx) populated. The create operation doesn’t support a pre-image and a delete operation doesn’t support a post-image. In addition, only a small subset of messages support pre and post images as shown in the following table.

|  |  |  |
| --- | --- | --- |
| **Message Request** | **Property** | **Description** |
| [AssignRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.assignrequest.aspx) | Target | The assigned entity. |
| [CreateRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.messages.createrequest.aspx) | Target | The created entity. |
| [DeleteRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.messages.deleterequest.aspx) | Target | The deleted entity. |
| [DeliverIncomingEmailRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.deliverincomingemailrequest.aspx) | EmailId | The delivered email ID. |
| [DeliverPromoteEmailRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.deliverpromoteemailrequest.aspx) | EmailId | The delivered email ID. |
| [ExecuteWorkflowRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.executeworkflowrequest.aspx) | Target | The workflow entity. |
| [MergeRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.mergerequest.aspx) | Target | The parent entity, into which the data from the child entity is being merged. |
| [MergeRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.mergerequest.aspx) | SubordinateId | The child entity that is being merged into the parent entity. |
| [SendEmailRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.sendemailrequest.aspx) | EmailId | The sent entity ID. |
| [SetStateRequest](https://msdn.microsoft.com/en-gb/library/microsoft.crm.sdk.messages.setstaterequest.aspx) | EntityMoniker | The entity for which the state is set. |
| [UpdateRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.messages.updaterequest.aspx) | Target | The updated entity. |

IMP :

Registering for pre or post images to access entity attribute values results in improved plug-in performance as compared to obtaining entity attributes in plug-in code through [RetrieveRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.messages.retrieverequest.aspx) or [RetrieveMultipleRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.messages.retrievemultiplerequest.aspx) requests.

Suppose you registered the Plugin and added a Image with name “PreImage ”

Entity preMessageImage;

if (context.PreEntityImages.Contains(“PreImage”) && context.PreEntityImages["PreImage"] is Entity)

{

preMessageImage = (Entity)context.PreEntityImages["PreImage"];

accountnumber = (String)preMessageImage.Attributes["accountnumber"];

}

Suppose you registered the Plugin and added a Image with name “PostImage ”

Entity postMessageImage;

if (context.PostEntityImages.Contains(“PostImage”) && context.PostEntityImages["PostImage"] is Entity)

{

postMessageImage = (Entity)context.PostEntityImages["PostImage"];

accountnumber = (String)postMessageImage.Attributes["accountnumber"];

}

# Impersonation in plug-ins

Impersonation is used to execute business logic (custom code) on behalf of a Microsoft Dynamics CRM system user to provide a desired feature or service for that user. Any business logic executed within a plug-in, including Web service method calls and data access, is governed by the security privileges of the impersonated user.

Plug-ins not executed by either the sandbox or asynchronous service execute under the security account that is specified on the **Identity** tab of the **CRMAppPool Properties** dialog box. The dialog box can be accessed by right-clicking the **CRMAppPool** application pool in Internet Information Services (IIS) Manager and then clicking **Properties** in the shortcut menu. By default, CRMAppPool uses the Network Service account identity but this can be changed by a system administrator during installation. If the CRMAppPool identity is changed to a system account other than Network Service, the new identity account must be added to the PrivUserGroup group in Active Directory. More information: [TechNet: Change a Microsoft Dynamics CRM service account or AppPool identity](https://technet.microsoft.com/library/hh699751.aspx) for more detailed instructions.

The two methods that can be employed to impersonate a user are discussed below.

## Impersonation during plug-in registration

One method to impersonate a system user within a plug-in is by specifying the impersonated user during plug-in registration. When registering a plug-in programmatically, if the **SdkMessageProcessingStep.ImpersonatingUserId** attribute is set to a specific Microsoft Dynamics CRM system user, Web service calls made by the plug-in execute on behalf of the impersonated user. If **ImpersonatingUserId** is set to a value of **null** or **Guid.Empty** during plug-in registration, the calling/logged on user or the standard "system" user is the impersonated user.

Whether the calling/logged on user or "system" user is used for impersonation is dependent on the request being processed by the pipeline and is beyond the scope of the SDK documentation. For more information about the "system" user, refer to the next topic.

|  |
| --- |
| **System_CAPS_noteNote** |
| When you register a plug-in using the sample plug-in registration tool that is provided in the SDK download, service methods invoked by the plug-in execute under the account of the calling or logged on user by default unless you select a different user in the **Run in User’s Context** dropdown menu. For more information about the tool sample code, refer to the tool code under the SDK\Tools\PluginRegistration folder of the SDK package. [Download the Microsoft Dynamics CRM SDK package.](http://go.microsoft.com/fwlink/?LinkID=627298) |

## Impersonation during plug-in execution

Impersonation that was defined during plug-in registration can be altered in a plug-in at run time. Even if impersonation was not defined at plug-in registration, plug-in code can still use impersonation. The following discussion identifies the key properties and methods that play a role in impersonation when making Web service method calls in a plug-in.

The platform passes the impersonated user ID to a plug-in at run time through the [UserId](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.userid.aspx) property. This property can have one of three different values as shown in the table below.

|  |  |
| --- | --- |
| **UserId Value** | **Condition** |
| Initiating user or "system" user | The **SdkMessageProcessingStep.ImpersonatingUserId** attribute is set to **null** or **Guid.Empty** at plug-in registration. |
| Impersonated user | The **ImpersonatingUserId** property is set to a valid system user ID at plug-in registration. |
| "system" user | The current pipeline was executed by the platform, not in direct response to a service method call. |

The [InitiatingUserId](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.initiatinguserid.aspx) property of the execution context contains the ID of the system user that called the service method that ultimately caused the plug-in to execute.

|  |
| --- |
| **System_CAPS_importantImportant** |
| For plug-ins executing offline, any entities created by the plug-in are owned by the logged on user. Impersonation in plug-ins is not supported while in offline mode. |

**Handle exceptions in plug-ins**

For synchronous plug-ins, whether registered in the sandbox or not, the Microsoft Dynamics CRM platform handles exceptions passed back from a plug-in by displaying an error message in a dialog of the web application user interface. The exception message for asynchronous registered plug-ins is written to a System Job (**AsyncOperation**) record which can be viewed in the System Jobs area of the web application.

For synchronous plug-ins, you can optionally display a custom error message in the error dialog of the web application by having your plug-in throw an [InvalidPluginExecutionException](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.invalidpluginexecutionexception.aspx) exception with the custom message string as the exception **Message** property value. If you throw [InvalidPluginExecutionException](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.invalidpluginexecutionexception.aspx) and do not provide a custom message, a generic default message is displayed in the error dialog. It is recommended that plug-ins only pass an [InvalidPluginExecutionException](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.invalidpluginexecutionexception.aspx) back to the platform.

If a synchronous plug-in returns an exception other than [InvalidPluginExecutionException](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.invalidpluginexecutionexception.aspx) back to the platform, the error dialog is displayed to the user and the exception message ([System.Exception.Message](https://msdn.microsoft.com/library/system.exception.message.aspx)) with stack trace is also written to one of two places. For plug-ins not registered in the sandbox, the information is written to the Application event log on the server that runs the plug-in. The event log can be viewed by using the Event Viewer administrative tool. For plug-ins registered in the sandbox, the exception message and stack trace is written to the Microsoft Dynamics CRM platform trace. For more information about tracing, see the Logging and Tracing section of the [Debug a plug-In](https://msdn.microsoft.com/en-gb/library/gg328574.aspx) topic.

# Event execution pipeline

**Dynamics CRM 2016**

[Other Versions](javascript:void(0))

https://i-msdn.sec.s-msft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636107049706926704

* [Dynamics CRM 2015](https://msdn.microsoft.com/en-gb/library/gg327941(v=crm.7).aspx)
* [Dynamics CRM 2013](https://msdn.microsoft.com/en-gb/library/gg327941(v=crm.6).aspx)
* [Dynamics CRM 2011](https://msdn.microsoft.com/en-gb/library/gg327941(v=crm.5).aspx)

Applies To: Dynamics CRM 2016, Dynamics CRM Online

The Microsoft Dynamics CRM event processing subsystem executes plug-ins based on a message pipeline execution model. A user action in the Microsoft Dynamics CRM Web application or an SDK method call by a plug-in or other application results in a message being sent to the organization Web service. The message contains business entity information and core operation information. The message is passed through the event execution pipeline where it can be read or modified by the platform core operation and any registered plug-ins.

|  |
| --- |
| **System_CAPS_noteNote** |
| While there are several Web services hosted by the Microsoft Dynamics CRM platform, only events triggered by the organization and OData endpoints can cause plug-ins to execute. |

## In This Topic

[Architecture and related components](https://msdn.microsoft.com/en-gb/library/gg327941.aspx#bkmk_Architecture)

[Pipeline stages](https://msdn.microsoft.com/en-gb/library/gg327941.aspx#bkmk_PipelineStages)

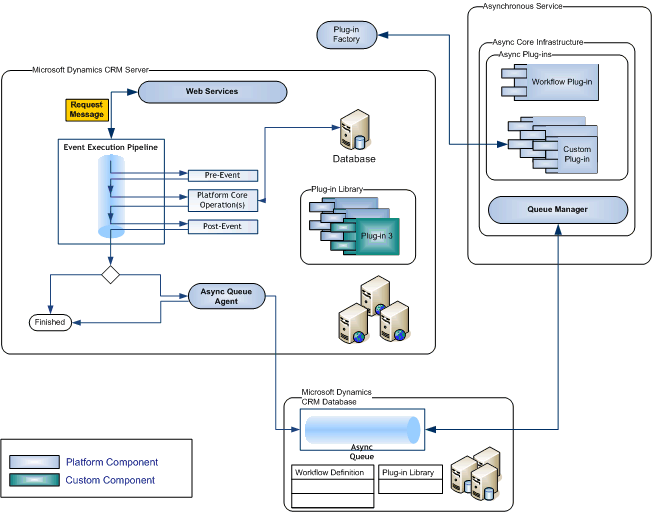
[Message processing](https://msdn.microsoft.com/en-gb/library/gg327941.aspx#bkmk_MessageProcessing)

[Plug-in registration](https://msdn.microsoft.com/en-gb/library/gg327941.aspx#bkmk_PluginRegistration)

[Inclusion in database transactions](https://msdn.microsoft.com/en-gb/library/gg327941.aspx#bkmk_DatabaseTransactions)

## Architecture and related components

The following figure illustrates the overall architecture of the Microsoft Dynamics CRM platform with respect to both synchronous and asynchronous event processing.



Synchronous and Asynchronous Event Processing Diagram

The event execution pipeline processes events either synchronously or asynchronously. The platform core operation and any plug-ins registered for synchronous execution are executed immediately. Synchronous plug-ins that are registered for the event are executed in a well-defined order. Plug-ins registered for asynchronous execution are queued by the Asynchronous Queue Agent and executed at a later time by the asynchronous service.

|  |
| --- |
| **System_CAPS_importantImportant** |
| Regardless of whether a plug-in executes synchronously or asynchronously, there’s a two-minute time limit imposed on the execution of a (message) request. If the execution of your plug-in logic exceeds the time limit, a [System.TimeoutException](https://msdn.microsoft.com/library/system.timeoutexception.aspx) is thrown. If a plug-in needs more processing time than two minutes, consider using a workflow or other background process to accomplish the intended task. This two-minute time limit applies only to plug-ins registered to execute under partial trust, also known as the sandbox. More information:[Plug-in isolation, trusts, and statistics](https://msdn.microsoft.com/en-gb/library/gg334752.aspx) |

## Pipeline stages

The event pipeline is divided into multiple stages, of which 4 are available to register custom developed or 3rd party plug-ins. Multiple plug-ins that are registered in each stage can be further be ordered (ranked) within that stage during plug-in registration.

|  |  |  |  |
| --- | --- | --- | --- |
| **Event** | **Stage name** | **Stage number** | **Description** |
| Pre-Event | Pre-validation | 10 | Stage in the pipeline for plug-ins that are to execute before the main system operation. Plug-ins registered in this stage may execute outside the database transaction.   |  | | --- | | **System_CAPS_securitySecurity Note** | | The pre-validation stage occurs prior to security checks being performed to verify the calling or logged on user has the correct permissions to perform the intended operation. | |
| Pre-Event | Pre-operation | 20 | Stage in the pipeline for plug-ins that are to execute before the main system operation. Plug-ins registered in this stage are executed within the database transaction. |
| Platform Core Operation | MainOperation | 30 | 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. For internal use only. |
| Post-Event | Post-operation | 40 | Stage in the pipeline for plug-ins which are to execute after the main operation. Plug-ins registered in this stage are executed within the database transaction. |

## Message processing

Whenever application code or a workflow invokes a Microsoft Dynamics CRM Web service method, a state change in the system occurs that raises an event. The information passed as a parameter to the Web service method is internally packaged up into a [OrganizationRequest](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.organizationrequest.aspx) message and processed by the pipeline. The information in the **OrganizationRequest** message is passed to the first plug-in registered for that event where it can be read or modified before being passed to the next registered plug-in for that event and so on. Plug-ins receive the message information in the form of context that is passed to their [Execute](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iplugin.execute.aspx) method. The message is also passed to the platform core operation.

## Plug-in registration

Plug-ins can be registered to execute before or after the core platform operation. Pre-event registered plug-ins receive the **OrganizationRequest** message first and can modify the message information before the message is passed to the core operation. After the core platform operation has completed, the message is then known as the [OrganizationResponse](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.organizationresponse.aspx). The response is passed to the registered post-event plug-ins. Post-event plug-ins have the opportunity to modify the message before a copy of the response is passed to any registered asynchronous plug-ins. Finally, the response is returned to the application or workflow that invoked the original Web service method call.

Because a single Microsoft Dynamics CRM server can host more than one organization, the execution pipeline is organization specific. There is a virtual pipeline for every organization. Plug-ins registered with the pipeline can only process business data for a single organization. A plug-in that is designed to work with multiple organizations must be registered with each organization's execution pipeline.

## Inclusion in database transactions

Plug-ins may or may not execute within the database transaction of the Microsoft Dynamics CRM platform. Whether a plug-in is part of the transaction is dependent on how the message request is processed by the pipeline. You can check if the plug-in is executing in-transaction by reading the [IsInTransaction](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.iexecutioncontext.isintransaction.aspx) property inherited by [IPluginExecutionContext](https://msdn.microsoft.com/en-gb/library/microsoft.xrm.sdk.ipluginexecutioncontext.aspx) that is passed to the plug-in. If a plug-in is executing in the database transaction and allows an exception to be passed back to the platform, the entire transaction will be rolled back. Stages 20 and 40 are guaranteed to be part of the database transaction while stage 10 may be part of the transaction.

Any registered plug-in that executes during the database transaction and that passes an exception back to the platform cancels the core operation. This results in a rollback of the core operation. In addition, any pre-event or post event registered plug-ins that have not yet executed and any workflow that is triggered by the same event that the plug-in was registered for will not execute.

CRM 2015: Enhance Your Plugins by Using Shared Variables :

While it is commonly suggested that IExecutionContext.Depth should be used to prevent infinite loops in your plugins, IExecutionContext.SharedVariables can give you finer control over what your plugins are doing, in certain cases. Even better, Shared Variables provide a way to pass values from one plugin to another, as long as they’re in the same sequence of steps for a particular message. For instance, the example below (from the MSDN article, [Pass data between plug-ins (link is external)](https://msdn.microsoft.com/en-us/library/gg328579.aspx)) shows how to set a Shared Variable in the pre-event step and access it in the post-event.

    public class PreEventPlugin : IPlugin

    {

        public void Execute(IServiceProvider serviceProvider)

        {

            // Obtain the execution context from the service provider.

            Microsoft.Xrm.Sdk.IPluginExecutionContext context = (Microsoft.Xrm.Sdk.IPluginExecutionContext)

                serviceProvider.GetService(typeof(Microsoft.Xrm.Sdk.IPluginExecutionContext));

            // Create or retrieve some data that will be needed by the post event

            // plug-in. You could run a query, create an entity, or perform a calculation.

            //In this sample, the data to be passed to the post plug-in is

            // represented by a GUID.

            Guid contact = new Guid("{74882D5C-381A-4863-A5B9-B8604615C2D0}");

            // Pass the data to the post event plug-in in an execution context shared

            // variable named PrimaryContact.

            context.SharedVariables.Add("PrimaryContact", (Object)contact.ToString());

        }

    }

    public class PostEventPlugin : IPlugin

    {

        public void Execute(IServiceProvider serviceProvider)

        {

            // Obtain the execution context from the service provider.

            Microsoft.Xrm.Sdk.IPluginExecutionContext context = (Microsoft.Xrm.Sdk.IPluginExecutionContext)

                serviceProvider.GetService(typeof(Microsoft.Xrm.Sdk.IPluginExecutionContext));

            // Obtain the contact from the execution context shared variables.

            if (context.SharedVariables.Contains("PrimaryContact"))

            {

                Guid contact =

                    new Guid((string)context.SharedVariables["PrimaryContact"]);

                // Do something with the contact.

            }

        }

    }

One catch, however, is that the way you access the stored Shared Variable can change, depending on which stages are used, as described in the same article:

“For a plug-in registered in stage 20 or 40, to access the shared variables from a stage 10 registered plug-in that executes on create, update, delete or by a [RetrieveExchangeRateRequest (link is external)](https://msdn.microsoft.com/en-us/library/microsoft.crm.sdk.messages.retrieveexchangeraterequest.aspx), you must access the [ParentContext. (link is external)](https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.ipluginexecutioncontext.parentcontext.aspx)**SharedVariables** collection. For all other cases, [IPluginExecutionContext. (link is external)](https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.ipluginexecutioncontext.aspx)**SharedVariables** contains the collection.”

What this means is that if the Shared Variable is being set during Pre-Validation, you need to look at the Parent Context to find it when retrieving it in a Pre-Operation or Post-Operation step. It may not be in the immediate Parent Context either; you may need to check each parent’s parent to find where the Shared Variable has been stored.

Finally, you cannot currently share variables between steps using different messages and/or entities, even if one triggers the other. For instance, let’s say that we have a plugin that executes when an Account is updated and that this plugin updates associated Contacts. Let’s also say that we have another plugin that gets triggered when the Contact is updated, so we have a case where the Account update plugin triggers this Contact update plugin. You’d see that the IExecutionContext.Depth for the Contact plugin is set to 2, but the Shared Variables you set in the Account plugin will not be available to the Contact plugin.

Pass data between plug-ins

The message pipeline model for Microsoft Dynamics CRM defines a parameter collection of custom data values in the execution context that is passed through the pipeline and shared among registered plug-ins, even from different 3rd party developers. This collection of data can be used by different plug-ins to communicate information between plug-ins and enable chain processing where data processed by one plug-in can be processed by the next plug-in in the sequence and so on. This feature is especially useful in pricing engine scenarios where multiple pricing plug-ins pass data between one another to calculate the total price for a sales order or invoice. Another potential use for this feature is to communicate information between a plug-in registered for a pre-event and a plug-in registered for a post-event.

The name of the parameter that is used for passing information between plug-ins is [SharedVariables](https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.iexecutioncontext.sharedvariables.aspx). This is a collection of key\value pairs. At run time, plug-ins can add, read, or modify properties in the SharedVariables collection. This provides a method of information communication among plug-ins.

This sample shows how to use SharedVariables to pass data from a pre-event registered plug-in to a post-event registered plug-in.