

# Plugin and Query in CRM 2016

## Plugin

### Understand the data context passed to a plug-in

When a plug-in is run in response to an execution pipeline event for which it is registered, the plug-in's [Execute](#) method is called. That method passes an [IServiceProvider](#) object as a parameter, which contains a number of useful objects. The following sections describe some of the information that is passed to a plug-in when executed.

### Access the plug-in execution context

[IPluginExecutionContext](#) contains information that describes the run-time environment that the plug-in executes, information related to the execution pipeline, and entity business information. The context is contained in the [System.IServiceProvider](#) parameter that is passed at run time to a plug-in through its [Execute](#) method.

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

When a system event is fired that a plug-in is registered for, the system creates and populates the context and passes it to a plug-in through the previously mentioned classes and methods. The execution context is passed to each registered plug-in in the pipeline when they are executed. Each plug-in in the execution pipeline is able to modify writable properties in the context. For example, given a plug-in registered for a pre-event and another plug-in registered for a post-event, the post-event plug-in can receive a context that has been modified by the pre-event plug-in. The same situation applies to plug-ins that are registered within the same stage.

All the properties in [IPluginExecutionContext](#) are read-only. However, your plug-in can modify the contents of those properties that are collections. For more information about infinite loop prevention, see [Depth](#).

### Access the Organization service

To access the Microsoft Dynamics 365 organization service, it is required that plug-in code create an instance of the service through the [ServiceProvider.GetService](#) method.

```
// Obtain the organization service reference.  
IOrganizationServiceFactory serviceFactory = (IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));  
IOrganizationService service = serviceFactory.CreateOrganizationService(context.UserId);
```

The platform provides the correct web service URLs and network credentials for you when you use this method. Instantiating your own Web service proxy is not supported as it will create deadlock and authentication issues.

### Access the Notification service

Synchronous registered plug-ins can post the execution context to the Microsoft Azure Service Bus. The service provider object that is passed to the plug-in contains a reference to [IServiceEndpointNotificationService](#). It is through that notification service that synchronous plug-ins can send brokered messages to the Microsoft Azure Service Bus. For more information about Microsoft

Azure, see [Azure integration with Microsoft Dynamics 365](#). For more information about writing a plug-in that can post to the Microsoft Azure Service Bus, see [Write a custom Azure-aware plug-in](#).

### Input and output parameters

The [InputParameters](#) property contains the data that is in the request message currently being processed by the event execution pipeline. Your plug-in code can access this data. The property is of type [ParameterCollection](#) where the keys to access the request data are the names of the actual public properties in the request. As an example, take a look at [CreateRequest](#). One property of [CreateRequest](#) is named [Target](#), which is of type [Entity](#). This is the entity currently being operated upon by the platform. To access the data of the entity you would use the name “Target” as the key in the input parameter collection. You also need to cast the returned instance.

```
// The InputParameters collection contains all the data passed in the message request.
if (context.InputParameters.Contains("Target") && context.InputParameters["Target"] is Entity) {
    // Obtain the target entity from the input parameters.
    Entity entity = (Entity)context.InputParameters["Target"];
    if (entity.LogicalName != "account") { return; }
}
```

Note that not all requests contain a **Target** property that is of type [Entity](#), so you have to look at each request or response. For example, [DeleteRequest](#) has a [Target](#) property, but its type is [EntityReference](#). The preceding code example would be changed as follows.

```
// The InputParameters collection contains all the data passed in the message request.
if (context.InputParameters.Contains("Target") && context.InputParameters["Target"] is EntityReference)
{
    // Obtain the target entity from the input parameters.
    EntityReference entity = (EntityReference)context.InputParameters["Target"];
}
```

Once you have access to the entity data, you can read and modify it. Any data changes to the context performed by plug-ins registered in stage 10 or 20 of the pipeline are passed in the context to the core operation in stage 30.

Not all fields in an entity record that are passed to a plug-in through the context can be modified. You should check the field’s **IsValidForUpdate** metadata property to verify that it isn’t set to **false**. Attempting to change the value of a field that can’t be updated results in an exception.

Similarly, the [OutputParameters](#) property contains the data that is in the response message, for example [CreateResponse](#), currently being passed through the event execution pipeline. However, only synchronous post-event and asynchronous registered plug-ins have [OutputParameters](#) populated as the response is the result of the core platform operation. The property is of type [ParameterCollection](#) 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](#) and [PostEntityImages](#) contain snapshots of the primary entity's attributes before (pre) and after (post) the core platform operation. Microsoft Dynamics 365 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](#) 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
<a href="#">AssignRequest</a>	Target	The assigned entity.
<a href="#">CreateRequest</a>	Target	The created entity.
<a href="#">DeleteRequest</a>	Target	The deleted entity.
<a href="#">DeliverIncomingEmailRequest</a>	EmailId	The delivered email ID.
<a href="#">DeliverPromoteEmailRequest</a>	EmailId	The delivered email ID.
<a href="#">ExecuteWorkflowRequest</a>	Target	The workflow entity.
<a href="#">MergeRequest</a>	Target	The parent entity, into which the data from the child entity is being merged.
<a href="#">MergeRequest</a>	SubordinateId	The child entity that is being merged into the parent entity.
<a href="#">SendEmailRequest</a>	EmailId	The sent entity ID.
<a href="#">SetStateRequest</a>	EntityMoniker	The entity for which the state is set.
<a href="#">UpdateRequest</a>	Target	The updated entity.

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](#) or [RetrieveMultipleRequest](#) requests.



#### Security Note

A pre-image passed in the execution context to a plug-in or custom workflow activity might contain data that the logged-on user doesn't have the privileges to access. Microsoft Dynamics 365 administrators and other users with high-level permissions can register plug-ins to run under the "system" user account or plug-in code can make calls as a "system" user on behalf of the logged-on user. If this happens, logged-on users can access data that their field level security does not allow access to. More information: [Impersonation in plug-ins](#)

Pass data between plug-ins

The message pipeline model for Microsoft Dynamics 365 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](#). 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.

```

using System;

// Microsoft Dynamics CRM namespace(s)
using Microsoft.Xrm.Sdk;

namespace Microsoft.Crm.Sdk.Samples
{
    /// <summary>
    /// A plug-in that sends data to another plug-in through the SharedVariables
    /// property of IPluginExecutionContext.
    /// </summary>
    /// <remarks>Register the PreEventPlugin for a pre-operation stage and the
    /// PostEventPlugin plug-in on a post-operation stage.
    /// </remarks>
    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.
            }
        }
    }
}

```

It is important that any type of data added to the shared variables collection be serializable otherwise the server will not know how to serialize the data and plug-in execution will fail.

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](#), you must access the [ParentContext.SharedVariables](#) collection. For all other cases, [IPluginExecutionContext.SharedVariables](#) contains the collection.

## Impersonation in plug-ins

Impersonation is used to execute business logic (custom code) on behalf of a Microsoft Dynamics 365 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](#) 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 365 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.

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.](#)

### 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](#) 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 <b>SdkMessageProcessingStep.ImpersonatingUserId</b> attribute is set to <b>null</b> or <b>Guid.Empty</b> at plug-in registration.
Impersonated user	The <b>ImpersonatingUserId</b> 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](#) 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.

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 365 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](#) exception with the custom message string as the exception **Message** property value. If you throw [InvalidPluginExecutionException](#) 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](#) back to the platform.

If a synchronous plug-in returns an exception other than [InvalidPluginExecutionException](#) back to the platform, the error dialog is displayed to the user and the exception message ([System.Exception.Message](#)) 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 365 platform trace. For more information about tracing, see the Logging and Tracing section of the [Debug a plug-in](#) topic.

## Write a plug-in

Plug-ins are custom classes that implement the [IPlugin](#) interface. You can write a plug-in in any .NET Framework 4.5.2 CLR-compliant language such as Microsoft Visual C# and Microsoft Visual Basic .NET. To be able to compile plug-in code, you must add Microsoft.Xrm.Sdk.dll and Microsoft.Crm.Sdk.Proxy.dll assembly references to your project. These assemblies can be found in the SDK\Bin folder of the SDK. [Download the Microsoft Dynamics CRM SDK package.](#)

### Plug-in design

Your plug-in design should take into account the web application *auto-save* feature introduced in Microsoft Dynamics 365 (online & on-premises). Auto-save is enabled by default but can be disabled at an organization level. When auto-save is enabled there is no **Save** button. The web application will save data in the form automatically 30 seconds after the last unsaved change. You can apply form scripts to disable the auto-save behaviors on a form level. Depending on how you registered your plug-in, auto-save may result in your plug-in being called more frequently for individual field changes instead of one plug-in invocation for all changes. You should assume that any user can save any record at any time, whether this is done using Ctrl+S, by pressing a save button, or automatically due to the auto-save feature.

It is a best practice to register your plug-in or workflow on entities and specific fields that matter most. Avoid registering a plug-in or workflow for changes to all entity fields. If you have an existing plug-in or workflow that was implemented before the availability of the auto save feature, you should re-test that code to verify its proper operation. For more information see [TechNet: Manage auto-save](#).

### Writing a Basic Plug-in

The following sample shows some of the common code found in a plug-in. For this sample, the code omits any custom business logic that would perform the intended task of the plug-in. However, the code does show a plug-in class that implements the [IPlugin](#) interface and the required [Execute](#) method.



```

using System;
using System.ServiceModel;
using Microsoft.Xrm.Sdk;

public class MyPlugin: IPlugin
{
    public void Execute(IServiceProvider serviceProvider)
    {
        // Extract the tracing service for use in debugging sandboxed plug-ins.
        // If you are not registering the plug-in in the sandbox, then you do
        // not have to add any tracing service related code.
        ITracingService tracingService =
            (ITracingService)serviceProvider.GetService(typeof(ITracingService));

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

        // The InputParameters collection contains all the data passed in the message request.
        if (context.InputParameters.Contains("Target") &&
            context.InputParameters["Target"] is Entity)
        {
            // Obtain the target entity from the input parameters.
            Entity entity = (Entity)context.InputParameters["Target"];

            // Verify that the target entity represents an entity type you are expecting.
            // For example, an account. If not, the plug-in was not registered correctly.
            if (entity.LogicalName != "account")
                return;

            // Obtain the organization service reference which you will need for
            // web service calls.
            IOrganizationServiceFactory serviceFactory =
                (IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));
            IOrganizationService service = serviceFactory.CreateOrganizationService(context.UserId);

            try
            {
                // Plug-in business logic goes here.
            }

            catch (FaultException<OrganizationServiceFault> ex)
            {
                throw new InvalidPluginExecutionException("An error occurred in MyPlug-in.", ex);
            }

            catch (Exception ex)
            {
                tracingService.Trace("MyPlugin: {0}", ex.ToString());
                throw;
            }
        }
    }
}

```

The [IServiceProvider](#) parameter of the [Execute](#) method is a container for several service useful objects that can be accessed within a plug-in. The service provider contains instance references to the execution context, [IOrganizationServiceFactory](#), [ITracingService](#), and more. The sample code demonstrates how to obtain references to the execution context, [IOrganizationService](#), and [ITracingService](#) from the service provider parameter. For more information about the tracing service, refer to [Debug a plug-in](#).

The execution context contains a wealth of information about the event that caused the plug-in to execute and the data contained in the message that is currently being processed by the pipeline. For more information about the data context, see [Understand the data context passed to a plug-in](#).

The platform provides the correct Web service URLs and network credentials when you obtain the organization Web service reference from the service provider. Instantiating your own Web service proxy is not supported because it creates deadlock and authentication issues. After you have the organization service reference, you can use it to make method calls to the organization Web service. You can retrieve or change business data in a single Microsoft Dynamics 365 organization by issuing one or more message requests to the Web service. For more information about message requests, see [Use messages \(request and response classes\) with the Execute method](#).

A typical plug-in should access the information in the context, perform the required business operations, and handle exceptions. For more information about handling exceptions in a plug-in, refer to [Handle exceptions in plug-ins](#). A more complete plug-in sample is available in the topic [Sample: Create a basic plug-in](#).

For improved performance, Microsoft Dynamics 365 caches plug-in instances. The plug-in's [Execute](#) method should be written to be stateless because the constructor is not called for every invocation of the plug-in. Also, multiple system threads could execute the plug-in at the same time. All per invocation state information is stored in the context, so you should not use global variables or attempt to store any data in member variables for use during the next plug-in invocation unless that data was obtained from the configuration parameter provided to the constructor. Changes to a plug-in's registration will cause the plug-in to be re-initialized.

### Write a Plug-in Constructor

The Microsoft Dynamics 365 platform supports an optional plug-in constructor that accepts either one or two string parameters. If you write a constructor like this, you can pass any strings of information to the plug-in at run time.

The following sample shows the format of the constructor. In this example, the plug-in class is named SamplePlugin.

```
public SamplePlugin()  
public SamplePlugin(string unsecure)  
public SamplePlugin(string unsecure, string secure)
```

The first string parameter of the constructor contains public (unsecure) information. The second string parameter contains non-public (secure) information. In this discussion, secure refers to an encrypted value while unsecure is an unencrypted value. When using Microsoft Dynamics 365 for Microsoft Office Outlook with Offline Access, the secure string is not passed to a plug-in that executes while Dynamics 365 for Outlook is offline.

The information that is passed to the plug-in constructor in these strings is specified when the plug-in is registered with Microsoft Dynamics 365. When using the Plug-in Registration tool to register a plug-in, you can enter secure and unsecure information in the **Secure Configuration** and **Unsecure Configuration** fields provided in the **Register New Step** form. When registering a plug-in

programmatically using the Microsoft Dynamics 365 SDK, **SdkMessageProcessingStep.Configuration** contains the unsecure value and **SdkMessageProcessingStep.SecureConfigId** refers to a **SdkMessageProcessingStepSecureConfig** record that contains the secure value.

### Support Offline Execution

You can register plug-ins to execute in online mode, offline mode, or both. Offline mode is only supported on Microsoft Dynamics 365 for Microsoft Office Outlook with Offline Access. Your plug-in code can check whether it is executing in offline mode by checking the [IsExecutingOffline](#) property.

When you design a plug-in that will be registered for both online and offline execution, remember that the plug-in can execute twice. The first time is while Microsoft Dynamics 365 for Microsoft Office Outlook with Offline Access is offline. The plug-in executes again when Dynamics 365 for Outlook goes online and synchronization between Dynamics 365 for Outlook and the Microsoft Dynamics 365 server occurs. You can check the [IsOfflinePlayback](#) property to determine if the plug-in is executing because of this synchronization.

### Web Access for Isolated (sandboxed) Plug-ins

If you plan on registering your plug-in in the sandbox, you can still access Web addresses from your plug-in code. You can use any .NET Framework class in your plug-in code that provides Web access within the Web access restrictions outlined [Plug-in isolation, trusts, and statistics](#). For example, the following plug-in code downloads a Web page.

```
// Download the target URI using a Web client. Any .NET class that uses the
// HTTP or HTTPS protocols and a DNS lookup should work.
using (WebClient client = new WebClient())
{
    byte[] responseBytes = client.DownloadData(webAddress);
    string response = Encoding.UTF8.GetString(responseBytes);
}
```

For sandboxed plug-ins to be able to access external Web services, the server where the Sandbox Processing Service role is installed must be exposed to the Internet, and the account that the sandbox service runs under must have Internet access. Only outbound connections on ports 80 and 443 are required. Inbound connection access is not required. Use the Windows Firewall control panel to enable outbound connections for the Microsoft.Crm.Sandbox.WorkerProcess application located on the server in the %PROGRAMFILES%\Microsoft Dynamics 365\Server\bin folder.

### Use Early-Bound Types

To use early-bound Microsoft Dynamics 365 types in your plug-in code simply include the types file, generated using the CrmSvcUtil program, in your Microsoft Visual Studio plug-in project.

Conversion of a late-bound entity to an early-bound entity is handled as follows:

```
Account acct = entity.ToEntity<Account>();
```

In the previous line of code, the acct variable is an early-bound type. All [Entity](#) values that are assigned to [IPluginExecutionContext](#) must be late-bound types. If an early-bound type is assigned to the context, a [SerializationException](#) will occur. For more information, see [Understand the data context passed to a plug-in](#). Make sure that you do not mix your types and use an early bound type where a late-bound type is called for as shown in the following code.

```
context.InputParameters["Target"] = new Account() { Name = "MyAccount" }; // WRONG: Do not do this.
```

In the example above, you do not want to store an early-bound instance in the plug-in context where a late-bound instance should go. This is to avoid requiring the platform to convert between early-bound and late bound types before calling a plug-in and when returning from the plug-in to the platform.

## Plug-in Assemblies

There can be one or more plug-in types in an assembly. After the plug-in assembly is registered and deployed, plug-ins can perform their intended operation in response to a Microsoft Dynamics 365 run-time event.

In Microsoft Dynamics 365, plug-in assemblies must be readable by everyone to work correctly. Therefore, it is a security best practice to develop plug-in code that does not contain any system logon information, confidential information, or company trade secrets.

Each plug-in assembly must be signed, either by using the **Signing** tab of the project's properties sheet in Microsoft Visual Studio or the Strong Name tool, before being registered and deployed to Microsoft Dynamics 365. For more information about the Strong Name tool, run the sn.exe program, without any arguments, from a Microsoft Visual Studio Command Prompt window.

If your assembly contains a plug-in that can execute while the Dynamics 365 for Outlook is offline, there is additional security that the Microsoft Dynamics 365 platform imposes on assemblies. For more information, see [Walkthrough: Configure assembly security for an offline plug-in](#).

## Debug a plug-In

The following steps describe how to debug a plug-in executing on Microsoft Dynamics 365 on-premises. To debug a plug-in that executes in the sandbox on Microsoft Dynamics 365 (online), you must use tracing as described later in this topic.

### Debug a plug-in

1. Register and deploy the plug-in assembly.

If there is another copy of the assembly at the same location and you cannot overwrite that copy because it is locked by Microsoft Dynamics 365, you must restart the service process that was executing the plug-in. Refer to the table below for the correct service process. More information: [Register and Deploy Plug-Ins](#)

2. Configure the debugger.

Attach the debugger to the process on the Microsoft Dynamics 365 server that will run your plug-in. Refer to the following table to identify the process.

Plug-in Registration Configuration	Service Process
online	w3wp.exe
offline	Microsoft.Crm.Application.Host.exe
asynchronous registered plug-ins (or custom workflow assemblies)	CrmAsyncService.exe
sandbox (isolation mode)	Microsoft.Crm.Sandbox.WorkerProcess.exe

If there are multiple processes running the same executable file, for example multiple w3wp.exe processes, attach the debugger to all instances of the running executable process. Next, set one or more breakpoints in your plug-in code.

### 3. Test the plug-in.

Run the Microsoft Dynamics 365 application, or other custom application that uses the SDK, and perform whatever action is required to cause the plug-in to execute. For example, if a plug-in is registered for an account creation event, create a new account.

### 4. Debug your plug-in code.

Make any needed changes to your code so that it performs as you want. If the code is changed, compile the code into an assembly and repeat steps 1 through 4 in this procedure as necessary. However, if you change the plug-in assembly's major or minor version numbers, you must unregister the earlier version of the assembly and register the new version. More information: [Register and Deploy Plug-Ins](#)

### 5. Register the plug-in in the database.

After the edit/compile/deploy/test/debug cycle for your plug-in has been completed, unregister the (on-disk) plug-in assembly and then reregister the plug-in in the Microsoft Dynamics 365 database. More information: [Register and Deploy Plug-Ins](#)

It is possible to debug a database deployed plug-in. The compiled plug-in assembly's symbol file (**.pdb**) must be copied to the server's <crm-root>\Server\bin\assembly folder and Internet Information Services (IIS) must then be restarted. After debugging has been completed, you must remove the symbol file and reset IIS to prevent the process that was executing the plug-in from consuming additional memory.

For more information about debugging a plug-in using the Plug-in Profiler tool, see [Analyze plug-in performance](#).

### Debug a sandboxed plug-in

It is important to perform these steps before the first execution of a sandboxed plug-in. If the plug-in has already been executed, either change the code of the assembly, causing the hash of the assembly to

change on the server, or restart the Microsoft Dynamics 365 Sandbox Processing Service on the sandbox server.

### Configure the Server

The sandbox host process monitors the sandbox worker process which is executing the plug-in. The host process checks if the plug-in stops responding, if it is exceeding memory thresholds, and more. If the worker process doesn't respond for than 30 seconds, it will be shutdown. In order to debug a sandbox plug-in, you must disable this shutdown feature. To disable the shutdown feature, set the following registry key to 1 (DWORD):

**HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\MSCRM\SandboxDebugPlugins**

#### *Debug the Plug-in*

Follow these steps to debug a sandboxed plug-in.

1. Register the plug-in in the sandbox (isolation mode) and deploy it to the Microsoft Dynamics 365 server database.
2. Copy the symbol file (.pdb) of the compiled plug-in assembly to the server\bin\assembly folder on the server running the sandbox worker process named Microsoft.Crm.Sandbox.WorkerProcess.exe. This is the server hosting the Sandbox Processing Service role.
3. Follow the instructions in steps 2 through 4 presented at the beginning of this topic.

For more information about debugging a plug-in using the Plug-in Profiler tool, see [Analyze plug-in performance](#).

#### *Logging and tracing*

An alternative method to troubleshoot a plug-in or custom workflow activity (custom code), compared to debugging in Microsoft Visual Studio, is to use tracing. Tracing assists developers by recording run-time custom information as an aid in diagnosing the cause of code failures. Tracing is especially useful to troubleshoot Microsoft Dynamics 365 (online) registered custom code as it is the only supported troubleshooting method for that scenario. Tracing is supported for sandboxed (partial trust) and full trust registered custom code and during synchronous or asynchronous execution. Tracing isn't supported for custom code that executes in Microsoft Dynamics 365 for Outlook or other mobile client.

Recording of run-time tracing information for Microsoft Dynamics 365 is provided by a service named [ITracingService](#). Information provided to this service by custom code can be recorded in three different places as identified here.

#### *Trace log*

Trace log records of type **PluginTraceLog** can be found in the web application by navigating to **Settings** and choosing the **Plug-in Trace Log** tile. The tile is only visible if you have access to the trace log entity records in your assigned security role. Writing of these records is controlled by the trace settings mentioned in the next section. For information on required privileges for the **PluginTraceLog** entity, see [Privileges by entity](#).

Trace logging takes up organization storage space especially when many traces and exceptions are generated. You should only turn trace logging on for debugging and troubleshooting, and turn it off after your investigation is completed.

#### *Error dialog*

A synchronous registered plug-in or custom workflow activity that returns an exception back to the platform results in an error dialog box in the web application presented to the logged on user. The user may select the **Download Log File** button in the dialog to view the log containing exception and trace output.

#### *System job*

For asynchronous registered plug-in or custom workflow activities that returns an exception, the tracing information is shown in the **Details** area of the **System Job** form in the web application.

#### *Enable trace logging*

To enable trace logging in an organization that supports this feature, in the web application navigate to **Settings > Administration > System Settings**. In the **Customization** tab, locate the drop-down menu labeled **Enable logging to plug-in trace log** and select one of the available options.

Option	Description
Off	Writing to the trace log is disabled. No <b>PluginTraceLog</b> records will be created. However, custom code can still call the <a href="#">Trace</a> method even though no log is written.
Exceptions	Trace information is written to the log if an exception is passed back to the platform from custom code.
All	Trace information is written to the log upon code completion or an exception is passed back to the platform from the custom code.

If the trace logging setting is set to **Exception** and your custom code returns an exception back to the platform, a trace log record is created and tracing information is also written to one other location. For custom code that executes synchronously, the information is presented to the user in an error dialog box, otherwise, for asynchronous code, the information is written to the related system job.

By default, the System Administrator and System Customizer roles have the required privileges to change the trace logging setting, which is stored in a [TraceSettings](#) entity record. Trace settings have an organization scope.

#### *Write to the tracing service*

Before writing to the tracing service, you must first extract the tracing service object from the passed execution context. Afterwards, simply add [Trace](#) calls to your custom code where appropriate passing any relevant diagnostic information in that method call.

```

//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)
{
    // Obtain the target entity from the Input Parameters.
    tracingService.Trace
        ("AdvancedPlugin: Getting the target entity from Input Parameters.");
    Entity entity = (Entity)context.InputParameters["Target"];

    // Obtain the image entity from the Pre Entity Images.
    tracingService.Trace
        ("AdvancedPlugin: Getting image entity from PreEntityImages.");
    Entity image = (Entity)context.PreEntityImages["Target"];
}

```

Next, build and deploy the plug-in or custom workflow activity. During execution of the custom code, the information provided in the **Trace** method calls is written to a trace log entity record by [ITracingService](#), if supported by your organization and enabled, and may also be made available to the user in a Web dialog or system job as described in the previous section. Tracing information written to the trace log is configured in the trace settings. For more information see [Enable trace logging](#).

If your custom code executes within a database transaction, and an exception occurs that causes a transaction rollback, all entity data changes by your code will be undone. However, the **PluginTraceLog** records will remain after the rollback completes.

#### *About the tracing service*

The [ITracingService](#) batches the information provided to it through the **Trace** method. The information is written to a new **PluginTraceLog** record after the custom code successfully runs to completion or an exception is thrown.

PluginTraceLog records have a finite lifetime. A bulk deletion background job runs once per day to delete records that are older than 24 hours from creation. This job can be disabled when needed.



## Walkthrough: Configure assembly security for an offline plug-in

The Microsoft Dynamics 365 platform applies an additional security restriction to registered offline plug-in assemblies. When Microsoft Dynamics 365 for Microsoft Office Outlook with Offline Access is installed, an AllowList key is added to the system registry on the client computer. For each assembly containing an offline plug-in that you register, you must add a registry sub-key under the AllowList key with the key name derived from the assembly's public key token. Failure to add this key results in the offline plug-in not being executed by the platform even though the plug-in is registered. This walkthrough describes how to add this sub-key for a plug-in assembly.

### Get the public key token

1. Load the assembly containing the offline plug-in into the Plug-in Registration tool.
2. Select the plug-in assembly in the tree view of the tool.
3. Copy (Ctrl+C) the value in the **Public Key Token** field into the paste buffer.

### Add an AllowList key

1. Run the registry editor by selecting **Start**, then select **Run** and type regedit.exe.
2. In the tree view pane, navigate to the **AllowList** key. The complete path of the key is HKEY\_CURRENT\_USER\Software\Microsoft\MSCRMClient\AllowList.
3. Select the **AllowList** key and right click to display the context menu.
4. Select **New** then click **Key** to create a new sub-key.
5. Paste the public key token value into the name of the new sub-key.
6. Close the registry editor.

## Register and Deploy Plug-Ins

Plug-ins and custom workflow activities are custom code that you develop to extend the existing functionality of Microsoft Dynamics 365. Before a plug-in or custom workflow activity can be used, it must be registered with the server. You can programmatically register plug-ins and custom workflow activities with Microsoft Dynamics 365 (online & on-premises) by writing registration code using certain SDK classes. However, to ease the learning curve and to speed up development and deployment of custom code, a plug-in and custom workflow activity registration tool is included in Bin folder of the SDK. [Download the Microsoft Dynamics CRM SDK package.](#)

While this topic focuses primarily on plug-ins, most of the information is also applicable to custom workflow activities. One difference between the two is that for custom workflow activity assemblies, you register just the assembly. For plug-ins, you register the plug-in assembly and one or more steps per plug-in. For more information about custom workflow activities, see [Custom workflow activities \(workflow assemblies\)](#).

## Plug-in Registration Tool

The Plug-in Registration tool provides a graphical user interface and supports registering plug-ins and custom workflow activities with Microsoft Dynamics 365. However, plug-ins and custom workflow activities can only be registered in the sandbox (isolation mode) of Microsoft Dynamics 365 (online).

For more information about how to register and deploy a plug-in by using the tool, see [Walkthrough: Register a plug-in using the plug-in registration tool](#). The tool can be added to the Visual Studio **Tools** menu as an external tool to speed up the development process.

## Plug-in Storage

For an on-premises deployment, plug-ins that are not registered in the sandbox can be stored in the Microsoft Dynamics 365 server's database or the *on-disk* file system. We strongly recommend that you store your production-ready plug-ins in the Microsoft Dynamics 365 database, instead of on-disk. Plug-ins stored in the database are automatically distributed across multiple Microsoft Dynamics 365 servers in a data center cluster. On-disk storage of plug-ins is useful for debugging plug-ins using Microsoft Visual Studio. However, you can debug a plug-in that is stored in the database. For more information, see [Debug a plug-in](#).

Plug-ins registered in the sandbox must be stored in the database regardless of the Microsoft Dynamics 365 deployment (on-premises, IFD, or Online).

## Deployment

For on-premises or Internet-facing (IFD) Microsoft Dynamics 365 installations, when you deploy plug-ins from another computer to the Microsoft Dynamics 365 server disk (on-disk deployment), the plug-in assembly must be manually copied to the server before registration. The assembly must be deployed to the `<install_dir>\Program Files\Microsoft CRM\server\bin\assembly` folder on each server where the plug-in is to execute.

Plug-in registration should be done after the assembly has been copied to the `...\bin\assembly` folder on the server to prevent the situation where a system user causes an event in Microsoft Dynamics 365 to be raised but the registered plug-in assembly does not yet exist on the server. For server database deployment, the plug-in assembly is automatically copied during plug-in registration so that the earlier situation is not an issue.

Depending on your plug-in's design, your plug-ins may require other referenced assemblies to run. Regardless of whether you deploy your plug-in to the database or disk, if your plug-in requires other assemblies to run, you must put copies of these assemblies in the global assembly cache on each server where the plug-in is to execute. This does not apply to a Microsoft Dynamics 365 (online) server because you do not have access to the global assembly cache on that server.

### *To move a plug-in from a development environment to a staging or production server*

1. On the development computer, build the plug-in code. Do not include debug information. Optimize the plug-in for performance.
2. Register the plug-in in the Microsoft Dynamics 365 server database.
3. Using the Microsoft Dynamics 365 web application, create a solution or use an existing one, and add the plug-in to that solution.

4. After you have added any other desired components to the solution, export the solution.
5. Import the solution on to the staging or production server.

### Assembly Versioning and Solutions

Plug-in assemblies can be versioned using a number format of *major.minor.build.revision* defined in the Assembly.info file of the Microsoft Visual Studio project. Depending on what part of the assembly version number is changed in a newer solution, the following behavior applies when an existing solution is updated through import.

- The build or revision assembly version number is changed.

This is considered an in-place upgrade. The older version of the assembly is removed when the solution containing the updated assembly is imported. Any pre-existing steps from the older solution are automatically changed to refer to the newer version of the assembly.

- The major or minor assembly version number, except for the build or revision numbers, is changed.

When an updated solution containing the revised assembly is imported, the assembly is considered a completely different assembly than the previous version of that assembly in the existing solution. Plug-in registration steps in the existing solution will continue to refer to the previous version of the assembly. If you want existing plug-in registration steps for the previous assembly to point to the revised assembly, you will need to use the Plug-in Registration tool to manually change the step configuration to refer to the revised assembly type. This should be done before exporting the updated assembly into a solution for later import.

For more information about solutions, refer to [Introduction to solutions](#).

### Security Restrictions

There is a security restriction that enables only privileged users to register plug-ins. For plug-ins that are not registered in isolation, the system user account under which the plug-in is being registered must exist in the **Deployment Administrators** group of Deployment Manager. Only the System Administrator user account or any user account included in the **Deployment Administrators** group can run Deployment Manager.

For non-isolated plug-ins, failure to include the registering user account in the **Deployment Administrators** group results in an exception being thrown during plug-in registration. The exception description states "Not have enough privilege to complete Create operation for an SDK entity."

The system user account under which the plug-in is being registered must have the following organization-wide security privileges:

- prvCreatePluginAssembly
- prvCreatePluginType
- prvCreateSdkMessageProcessingStep
- prvCreateSdkMessageProcessingStepImage

- `privCreateSdkMessageProcessingStepSecureConfig`

For more information, see [Security role and privilege reference](#) and [The security model of Microsoft Dynamics 365](#).

For plug-ins registered in the sandbox (isolation mode), the system user account under which the plug-in is being registered must have the System Administrator role. Membership in the **Deployment Administrators** group is not required.

### Register Plug-ins Programmatically

The key entity types used to register plug-ins and images

are: **PluginAssembly**, **PluginType**, **SdkMessageProcessingStep**, and **SdkMessageProcessingStepImage**.

The key entity types used to register custom workflow activities are **PluginAssembly** and **PluginType**.

Use these entities with the create, update, retrieve, and delete operations.

For more information on images, see [Understand the data context passed to a plug-in](#).

### Enable or Disable Custom Code Execution

You can use Windows PowerShell to enable or disable custom code, including plug-ins and custom workflow activities, on the server as described here. Alternatively, you can use the Deployment Web service. For more information, see [Deployment entities and deployment configuration settings](#) to set the **CustomCodeSettings.AllowExternalCode** property.

To enable custom code execution

1. Open a Windows PowerShell command window.
2. Add the Microsoft Dynamics 365 PowerShell snap-in:

```
Add-PSSnapin Microsoft.Crm.PowerShell
```

3. Retrieve the current setting:

```
$setting = get-crmsetting customcodesettings
```

4. Modify the current setting:

```
$setting.AllowExternalCode="True"
```

```
set-crmsetting $setting
```

5. Verify the setting:

```
get-crmsetting customcodesettings
```

To disable custom code execution

1. Open a Windows PowerShell command window.
2. Add the Microsoft Dynamics 365 PowerShell snap-in:

```
Add-PSSnapin Microsoft.Crm.PowerShell
```

3. Retrieve the current setting:

**\$setting = get-crmsetting customcodesettings**

4. Modify the current setting:

**\$setting.AllowExternalCode=0**

**set-crmsetting \$setting**

5. Verify the setting:

**get-crmsetting customcodesettings**

### Walkthrough: Register a plug-in using the plug-in registration tool

This walkthrough demonstrates how to register a plug-in by using the Plug-in Registration tool that is provided in the SDK. The plug-in to register is the FollowupPlugin from the [Sample: Create a basic plug-in](#) topic.

The plug-in is to be registered on the **account** entity, [CreateRequest](#) message, on a post-event, and in the sandbox. The plug-in can be registered on any Microsoft Dynamics 365 (online & on-premises) deployment where your user account has the System Customizer or System Administrator role.

The following prerequisites must be completed before starting this walkthrough:

- Get the PluginRegistration.exe tool, located in the Tools\PluginRegistration folder of the SDK. [Download the Microsoft Dynamics CRM SDK package.](#)
- Obtain a system user account on a Microsoft Dynamics 365 server.
- Your user account must have the System Customizer or System Administrator role. See [How role-based security can be used to control access to entities in Microsoft Dynamics 365](#).

### Connect to the Microsoft Dynamics 365 Server

1. Run the Plug-in Registration tool.
2. Click **CREATE NEW CONNECTION**.
3. In the **Login** dialog, select the deployment type radio button corresponding to the Microsoft Dynamics 365 server you intend to register plug-ins with. The **On-premises** radio button includes an IFD deployment, the **Online** button is for the Windows Live provider of Microsoft Dynamics 365 (online), and the **Office 365** button is for the Microsoft Online Services provider of Microsoft Dynamics 365 (online).

The image shows two side-by-side 'Login' dialog boxes. The left dialog is for 'Online' deployment, and the right dialog is for 'On-premises' deployment. Both dialogs have a 'Login' button and a 'Cancel' button.

**Left Dialog (Online):**

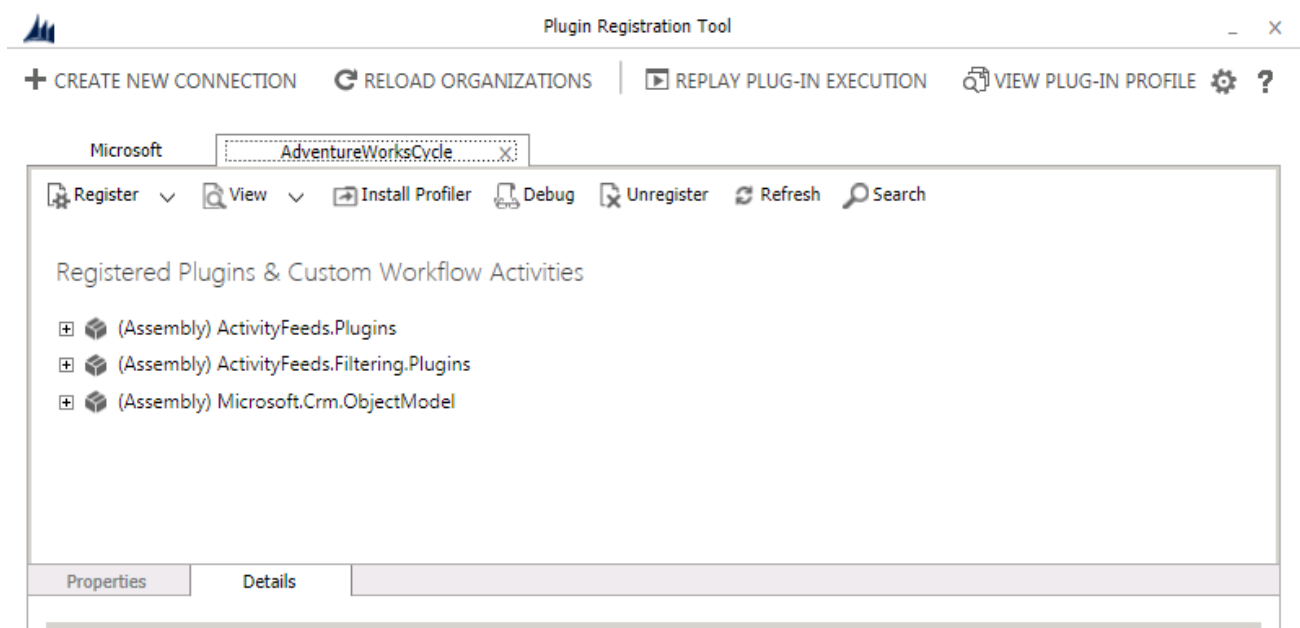
- Deployment Type: ☐ On-premises ☒ Online ☐ Office 365
- Online Region: North America (dropdown)
- User Name: dsmith@contoso.onmicrosoft.com
- Password: [masked]
- ☒ Always display list of available orgs

**Right Dialog (On-premises):**

- Deployment Type: ☒ On-premises ☐ Online ☐ Office 365
- Server: corpserv
- Port: 80 ☐ Use SSL
- Authentication Sour: Internet-facing deployment(IFD) (dropdown)
- ☐ Use Default Credentials
- User Name: dsmith
- Password: [masked]
- Domain: contoso.com
- ☒ Always display list of available orgs

4. If you check **Always display list of available orgs**, you are presented with a list of organizations that you belong to after you click **Login**. This enables you to choose the organization that you want to register the plug-in with. Otherwise, your default organization is used.
5. Enter the indicated information about the server and login account, and then click **Login**.

You should see a collapsed list of registered plug-in or custom workflow activity assemblies and service endpoints. The activity feeds and Microsoft.Crm.ObjectModel assemblies are required for Microsoft Dynamics 365 to function properly so the tool prevents you from altering them. Selecting an item in the list results in the **Properties** and **Details** tab panes displaying information about that list item.



The application's main window

Register a plug-in assembly

1. Select an organization tab to make it active.
2. In the toolbar of the tab, click **Register** and then **Register New Assembly**.
3. In the **Register New Assembly** dialog box, click the ellipses [...] button to the right of the **Step#1** field.
4. In the **Open** dialog box, navigate to the location of the compiled SamplePlugin.dll assembly. The default location is SDK\SampleCode\CS\Plugin-ins\bin\Debug. Select the assembly, and then click **Open**.
5. In the **Step#2** section, expand the **SamplePlugins** assembly to view all plug-ins in that assembly. Select (check) only the **Microsoft.Crm.Sdk.Samples.FollowupPlugin** plug-in.
6. In the **Step#3** section, select the **Sandbox** option.
7. In the **Step#4** section, select the **Database** option.

Register New Assembly

Step 1: Specify the location of the assembly to analyze

E:\crmsolutions\src\sdk\v6\Package\SampleCode\CS\Plug-ins\bin\Debug\SamplePlugins.dll

...

Load Assembly

Step 2: Select the plugin and workflow activities to register

☒ Select All / Deselect All

☐

(Assembly) SamplePlugins

☐

(Plugin) Microsoft.Crm.Sdk.Samples.AccountNumberPlugin - Isolatable

☐

(Plugin) Microsoft.Crm.Sdk.Samples.AdvancedPlugin - Isolatable

☒

(Plugin) Microsoft.Crm.Sdk.Samples.FollowupPlugin - Isolatable

☐

(Plugin) Microsoft.Crm.Sdk.Samples.PreEventPlugin - Isolatable

Step 3: Specify the isolation mode

☒ Sandbox ?

☐ None

Step 4: Specify the location where the assembly should be stored

☒ Database ?

☐ Disk ?

☐ GAC ?

Step 5: Log

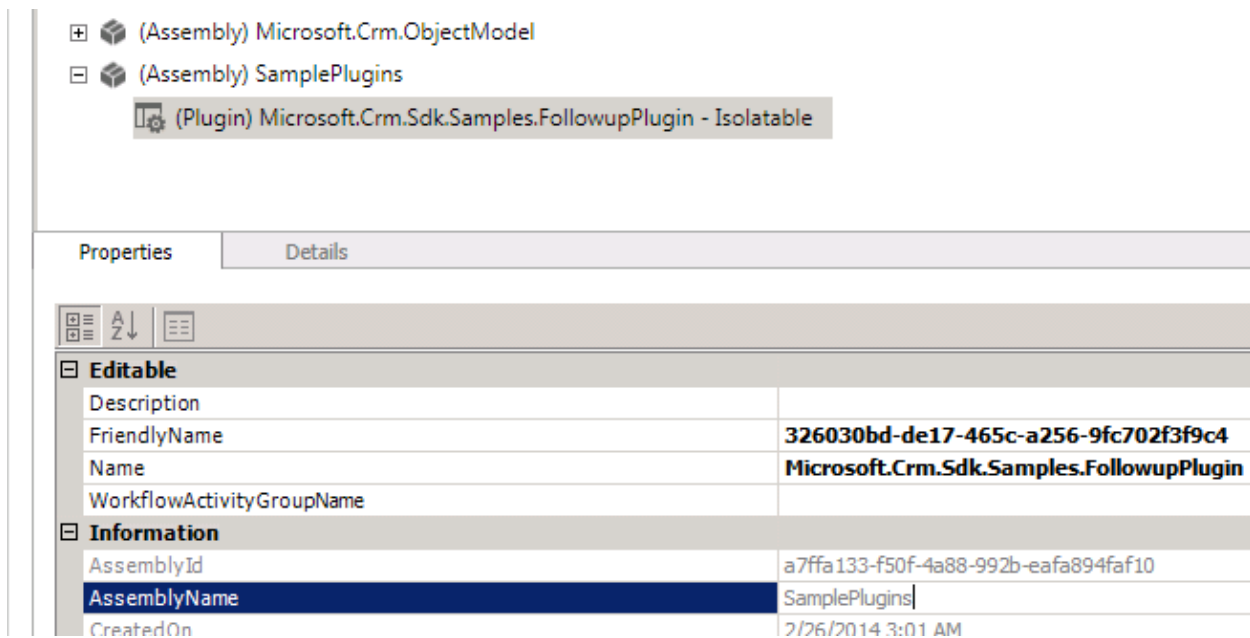
Register Selected Plugins

Close

Dialog to register an assembly

- Click **Register Selected Plugins**. You can close any open dialog boxes.





A registered plug-in shown in the tree view

Do you see an error in the **Log** area and the log contains the following message?

<Message>Action failed for assembly 'SamplePlugins, Version=0.0.0.0, Culture=neutral, PublicKeyToken=829f574d80e89132': Deployment/Scalegroup does not allow running external code.</Message>

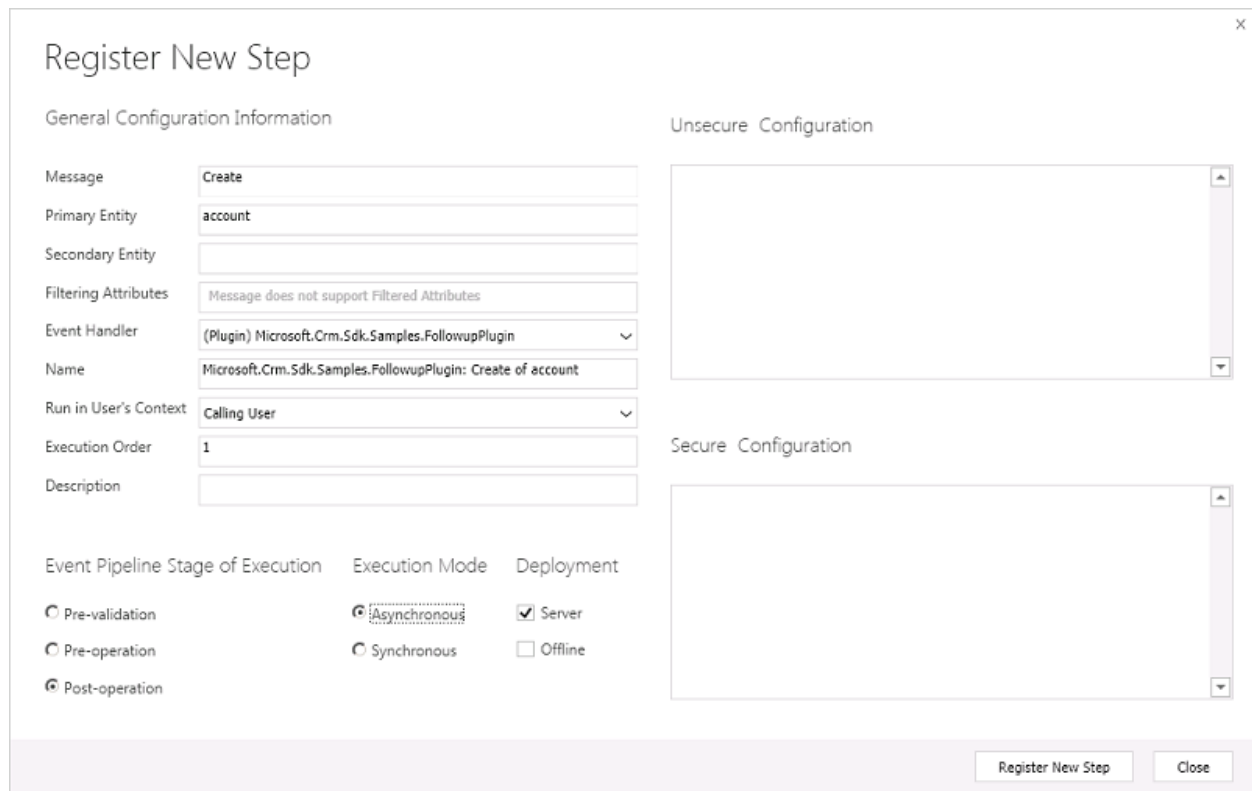
If so, you must enable custom code on the server and try again. For more information see [Enable or Disable Custom Code Execution](#).

The SamplePlugins.dll assembly and FollowupPlugin plug-in are now registered and deployed to the server. If you used the tool to register a custom workflow activity assembly, the next section on registering a step does not apply.

Register a plug-in step for an event

Plug-ins are registered to execute when an event is processed in the event execution pipeline. Each event has a stage name and number to indicate its location in the pipeline either before or after the core platform operation. A *step* refers to the SDK message processing step entity that is used to configure when and how the plug-in is to be executed.

1. In the **Registered Plug-ins & Custom Workflow Activities** tree view, expand the **(Assembly) SamplePlugins** node and select a registered plug-in.
2. Navigate to the **Register** menu in the toolbar, and then click **Register New Step**.
3. Complete the **Register New Step** dialog box as shown in the following figure.



The dialog box is titled "Register New Step" and contains two main sections: "General Configuration Information" and "Unsecure Configuration".

**General Configuration Information:**

- Message: Create
- Primary Entity: account
- Secondary Entity:
- Filtering Attributes: Message does not support Filtered Attributes
- Event Handler: (Plugin) Microsoft.Crm.Sdk.Samples.FollowupPlugin
- Name: Microsoft.Crm.Sdk.Samples.FollowupPlugin: Create of account
- Run in User's Context: Calling User
- Execution Order: 1
- Description:

**Execution Mode and Deployment:**

Event Pipeline Stage of Execution	Execution Mode	Deployment
<input type="radio"/> Pre-validation	<input checked="" type="radio"/> Asynchronous	<input checked="" type="checkbox"/> Server
<input type="radio"/> Pre-operation	<input type="radio"/> Synchronous	<input type="checkbox"/> Offline
<input checked="" type="radio"/> Post-operation		

**Unsecure Configuration:**

A large text area for unsecure configuration, currently empty.

**Secure Configuration:**

A large text area for secure configuration, currently empty.

At the bottom right, there are two buttons: "Register New Step" and "Close".

Dialog to register a new step

4. Click **Register New Step**.
5. Expand the **(Assembly) SamplePlugins** node and sub-nodes to see the plug-in and step nodes you created. You can now close the tool, but you may want to keep it open until after you test the plug-in and unregister the assembly.

To unregister the step, plug-in, or assembly, select its node in the tree, and then click **Unregister** in the tool bar. To modify an assembly or step registration, double-click the assembly or step node in the tree view. Alternately, you can select the node and click **Update** in the tool bar.

The plug-in is now registered to execute in the sandbox, for an account create event, and after the core operation executes. You registered the plug-in to run asynchronously since the creation of the follow-up task activity is not time critical. After an account is created, the plug-in will execute the next time the asynchronous service processes its queue.

### Test the plug-in

After you register the plug-in you can optionally test its execution by using the following procedure.

1. Open the Microsoft Dynamics 365 web application for the same organization that you registered the plug-in assembly under.
2. Move to the workplace, select **Accounts**, and then click **New**.
3. In the **Account Name** box, type an account name, for example, **Adventure Works Cycle**, and then click **Save & Close**.

4. Double-click the form name in the **Accounts** grid to open the form.
5. Click **Activities** to display a list of related activities for the account. You should see the activity named "Send email to the new customer" that the plug-in created.
6. If you registered the plug-in to run asynchronously, and did not select the **Delete AsyncOperation if StatusCode = Successful** option on the **Register New Step** form, there will be a new system job named "FollowupPlugin: Create of account". To view the related system job, click **Settings**, and then click **System Jobs**. Double-click the system job previously mentioned.

You can now unregister the step, plug-in, and assembly if you want. You may also want to delete the system job and account that you created.

### Analyze plug-in performance

The *Plug-in Profiler* is a tool that profiles the execution of plug-ins and custom workflow activities for an enhanced debugging experience in Microsoft Visual Studio. This tool, which can be run from the Command Prompt window or from within the Plug-in Registration tool, makes developing custom code against Microsoft Dynamics 365 (online & on-premises) quicker and easier. In addition, users can profile the execution of failing code and send the results to the developer of the code or independent software vendor (ISV) for analysis. The developer can replay the plug-in or custom workflow activity execution and debug the code remotely even when disconnected from the Microsoft Dynamics 365 server.

The tool can be used in either the debug or replay mode. Each of these modes is described in the following sections.

### Debug a plug-in using the plug-in profiler

1. Run the Plug-in Registration tool. You can find the tool's executable file in the Tools\PluginRegistration folder of the SDK. [Download the Microsoft Dynamics CRM SDK package.](#)
2. Click **CREATE NEW CONNECTION** to connect to a Microsoft Dynamics 365 server and organization. For more information on connecting to a server and organization, refer to the SDK topic: [Walkthrough: Register a plug-in using the plug-in registration tool.](#)
3. Register a plug-in and step on the Microsoft Dynamics 365 server. Keep a copy of the debug compiled plug-in assembly on the computer where you are running the tool.
4. In the toolbar for the target organization, select **Install Profiler**. You will now see a **Plug-in Profiler** node in the list.
5. Select a plug-in step and click **Start Profiling** in the toolbar to begin profiling. You can choose how the profiler executes in the displayed **Profiler Settings** dialog.
6. Perform the operation in Microsoft Dynamics 365 that causes the plug-in to run. For example, if the step is configured for an update to an account, then update an account.

7. If you have selected the **Exception** option in the **Profiler Settings** dialog, after the plug-in throws an exception and the **Business Process Error** dialog is displayed, click **Download Log File** and save this file. Alternately, if the plug-in does not throw an exception, click **Stop Profiling**.
8. In the Plug-in Registration tool, click **Debug**.
9. In the **Debug an Existing Plug-in** dialog box, provide the requested information in the **Setup** tab. Enter the location of the previously saved log file in the **Profile** field. Enter or select the location of the plug-in assembly and the class name of the plug-in that was executed.
10. Launch Microsoft Visual Studio and attach the debugger to the PluginRegistration.exe process.
11. Set a breakpoint in the plug-in code.
12. Click **Start Execution** in the **Debug an Existing Plug-in** dialog box.
13. After a slight delay, the plug-in will execute using the same execution context that was passed to it by the Microsoft Dynamics 365 server and the debugger will stop execution on the breakpoint that you previously set.
14. Continue debugging the plug-in as you would normally do. Any traces that the plug-in outputs are shown in the **Debug an Existing Plug-in** dialog box.

At this point you can alter the plug-in code, build it, re-attach the debugger to the PluginRegistration.exe process, and click **Start Execution** to continue with your debugging session. While performing these operations, you do not need to close the **Debug an Existing Plug-in** form.

You do not need to re-deploy the plug-in to the Microsoft Dynamics 365 server until after you have fixed the code problem. This debugging scenario works even if you have an optimized version of the plug-in on the server and a debug version of the plug-in on the computer where you are debugging.

#### Replay plug-in execution

Replaying plug-in execution does not require a connection to a Microsoft Dynamics 365 server and organization. The advantage of this method is that you can obtain the plug-in execution profile from a customer and debug the plug-in remotely. A restriction of the replay feature is that you cannot change the sequence of calls your plug-in code makes in the debugger while you are debugging.

The replay feature provides the plug-in with a snapshot of the call data and event execution context from the Microsoft Dynamics 365 server. You are getting the same events, GUIDs, and so on from calls to the Organization service but no data is being modified on the server as you debug the plug-in. During the debugging procedure in the previous section, the plug-in actually connects to the server and makes calls in real time.

#### Replay plug-in execution

1. Launch Microsoft Visual Studio and attach the debugger to the PluginRegistration.exe process.
2. Click **REPLAY PLUG-IN EXECUTION**.
3. Enter the log and plug-in assembly locations in the dialog box.
4. Click **Start Execution**.

5. Debug the plug-in as you would normally.

#### *Run the plug-in profiler standalone*

The profiler can be executed from a Command Prompt window independent of the Plug-in Registration tool. This is useful to obtain the plug-in profile log from a customer's Microsoft Dynamics 365 server to debug a failed plug-in. A developer can then use that log to replay the plug-in's execution in the Plug-in Registration tool and debug the plug-in using Microsoft Visual Studio.

#### *Run the plug-in profiler from a command prompt*

1. Open a Command Prompt window and set the working directory to the Tools\PluginRegistration folder in the SDK download.
2. Type the command: **PluginProfiler.Debugger.exe /?**.
3. Review the supported parameter list and re-run the PluginProfiler.Debugger.exe program with the appropriate parameters.

# Query

## Build queries with FetchXML

FetchXML is a proprietary query language that is used in Microsoft Dynamics 365 (online & on-premises). It's based on a schema that describes the capabilities of the language. The FetchXML language supports similar query capabilities as query expressions. In addition, it's used as a serialized form of query, used to save a query as a user-owned saved view in the **userquery** entity and as an organization-owned saved view in the **savedquery** entity.

A FetchXML query can be executed by using the [IOrganizationService.RetrieveMultiple](#) method. You can convert a FetchXML query to a query expression with the [FetchXmlToQueryExpressionRequest](#) message.

For information about how to use LINQPad to work with FetchXML, see this blog post: [Getting FetchXML from LINQPad](#).

For information about a utility that you can use to convert SQL scripts to FetchXML, see [SQL2FetchXML Help](#).

Use FetchXML to construct a query

To execute a FetchXML query in Microsoft Dynamics 365 and Microsoft Dynamics 365 (online), you must first build the XML query string. After you create the query string, use the [IOrganizationService.RetrieveMultiple](#) method to execute the query string. The privileges of the logged on user affects the set of records returned. Only records for which the logged on user has read access will be returned.

The FetchXML query string must conform to the schema definition for the FetchXML language. For more information, see [FetchXML schema](#).

You can save a query by creating a **SavedQuery** record, as demonstrated in [Sample: Validate and execute a saved query](#). Set **visible** on the **link-entity** node to **false** to hide the linked entity in the **Advanced Find** user interface. It will still participate in the execution of the query and will return the appropriate results.

Don't retrieve all attributes in a query because of the negative effect on performance. This is particularly true if the query is used as a parameter to an update request. In an update, if all attributes are included this sets all field values, even if they are unchanged, and often triggers cascaded updates to child records.

### Create the Query String

In the following example, the **FetchXML** statement retrieves all accounts:

```
<fetch mapping='logical'>
  <entity name='account'>
    <attribute name='accountid' />
    <attribute name='name' />
  </entity>
</fetch>
```

In the following example, the **FetchXML** statement retrieves all accounts where the last name of the owning user is not equal to Cannon:

```
<fetch mapping='logical'>
  <entity name='account'>
    <attribute name='accountid' />
    <attribute name='name' />
    <link-entity name='systemuser' to='owninguser'>
      <filter type='and'>
        <condition attribute='lastname' operator='ne' value='Cannon' />
      </filter>
    </link-entity>
  </entity>
</fetch>
```

In the following example, the **FetchXML** statement uses count to set the maximum number of records returned from the query. In this case first 3 accounts are returned from the query,

```
<fetch mapping='logical' count='3'>
  <entity name='account'>
    <attribute name='name' alias='name' />
  </entity></fetch>
```

This example shows an inner join between EntityMap and AttributeMap where the EntityMapID matches.

```
<fetch version='1.0' mapping='logical' distinct='false'>
  <entity name='entitymap'>
    <attribute name='sourceentityname' />
    <attribute name='targetentityname' />
    <link-entity name='attributemap' alias='attributemap' to='entitymapid' from='entitymapid' link-type='inner'>
      <attribute name='sourceattributename' />
      <attribute name='targetattributename' />
    </link-entity>
  </entity>
</fetch>
```

### *Execute the Query*

The following code shows how to execute a **FetchXML** query:

```

// Retrieve all accounts owned by the user with read access rights to the accounts and
// where the last name of the user is not Cannon.
string fetchXml = @"
<fetch mapping='logical'>
  <entity name='account'>
    <attribute name='accountid' />
    <attribute name='name' />
    <link-entity name='systemuser' to='owninguser'>
      <filter type='and'>
        <condition attribute='lastname' operator='ne' value='Cannon' />
      </filter>
    </link-entity>
  </entity>
</fetch> ";

EntityCollection result = service.RetrieveMultiple(new FetchExpression(fetchXml));
foreach (var c in result.Entities) {
    System.Console.WriteLine(c.Attributes["name"]);
}

```

### Query Results

When you execute a FetchXML query by using the [RetrieveMultiple](#) method, the return value is an [EntityCollection](#) that contains the results of the query. You can then iterate through the entity collection. The previous example uses the **foreach** loop to iterate through the result collection of the FetchXML query.

### Use FetchXML aggregation

In Microsoft Dynamics CRM 2015 and Microsoft Dynamics 365 (online), **FetchXML** includes grouping and aggregation features that let you calculate sum, average min, max and count.

The following aggregate functions are supported:

- sum
- avg
- min
- max
- count(\*)
- count(*attribute name*)

### About aggregation

To create an aggregate attribute, set the keyword **aggregate** to **true**, then specify a valid *entity name*, *attribute name*, and *alias* (variable name). You must also specify the type of aggregation you want to perform.

The following example shows a simple aggregate attribute in **FetchXML**.



```
<fetch distinct='false' mapping='logical' aggregate='true'>
  <entity name='entity name'>
    <attribute name='attribute name' aggregate='count' alias='alias name' />
  </entity>
</fetch>"
```

The result of a query with an aggregate attribute is different from the results of a standard query. The alias value is used as the tag identifier for the aggregate result.

The following example shows the format of the result of an aggregate query.

```
<resultset morerecords="0">
  <result>
    <alias>aggregate value</alias>
  </result>
</resultset>"
```

The following example shows the results of a query when the alias variable is set to *account\_count*.

```
<resultset morerecords="0">
  <result>
    <account_count>20</account_count>
  </result>
</resultset>"
```

## Page large result sets with FetchXML

You can page the results of a FetchXML query by using the paging cookie. The paging cookie is a performance feature that makes paging in the application faster for very large datasets. When you query for a set of records, the result will contain a value for the paging cookie. For better performance, you can pass that value when you retrieve the next set of records.

FetchXML and [QueryExpression](#) use different formats for their paging cookies. If you convert from one query format to the other by using the [FetchXmlToQueryExpressionRequest](#) message or the [QueryExpressionToFetchXmlRequest](#) message, the paging cookie value is ignored. In addition, if you request nonconsecutive pages, the paging cookie value is ignored.

## Use a left outer join in FetchXML to query for records "not in"

You can use a left outer join in FetchXML to perform a query that filters on the join table, such as to find all contacts who did not have any campaign activities in the past two months. Another common use for this type of a query is to find records “not in” a set, such as in these cases:

- Find all leads that have no tasks
- Find all accounts that have no contacts
- Find all leads that have one or more tasks

A left outer join returns each row that satisfies the join of the first input with the second input. It also returns any rows from the first input that had no matching rows in the second input. The nonmatching rows in the second input are returned as null values.

You can perform a left outer join in FetchXML by using the **entityname** attribute as a condition operator. The **entityname** attribute is valid in conditions, filters, and nested filters.

You can create a query using a left outer join programmatically and execute the query using the [RetrieveMultipleRequest](#), and you can save the query by creating a **SavedQuery** record. You can open a saved query that contains a left outer join in the Advanced Find or Saved Query editors in the web application and execute and view results, but some editor functionality is disabled. Those editors will allow modifications to the query, such as to change the columns returned, but the editor does not support changing the left outer join.

```
<fetch mapping='logical'>
  <entity name='account'>
    <attribute name='name' />
    <link-entity name='lead'
      from='leadid'
      to='originatingleadid'
      link-type='outer' />
    <filter type='and'>
      <condition entityname='lead'
        attribute='leadid'
        operator='null' />
    </filter>
  </entity>
</fetch>
```

Example: Find all leads that have no tasks, using an alias

The following shows how to construct the query in FetchXML:

```
<fetch version="1.0" output-format="xml-platform" mapping="logical" distinct="true">
  <entity name="lead">
    <attribute name="fullname" />
    <link-entity name="task" from="regardingobjectid" to="leadid" alias="ab" link-type="outer">
      <attribute name="regardingobjectid" />
    </link-entity>
    <filter type="and">
      <condition entityname="ab" attribute="regardingobjectid" operator="null" />
    </filter>
  </entity>
</fetch>
```

## Build queries with QueryExpression

In Microsoft Dynamics 365 (online & on-premises), you can use the [QueryExpression](#) class to programmatically build a query containing data filters and search conditions that define the scope of a database search. A query expression is used for single-object searches. For example, you can create a search to return all accounts that match certain search criteria. The [QueryBase](#) class is the base class for query expressions. There are two derived classes: [QueryExpression](#) and [QueryByAttribute](#). The **QueryExpression** class supports complex queries. The **QueryByAttribute** class is a simple means to search for entities where attributes match specified values.

Query expressions are used in methods that retrieve more than one record, such as the [IOrganizationService.RetrieveMultiple](#) method, in messages that perform an operation on a result set specified by a query expression, such as [BulkDeleteRequest](#) and when the ID for a specific record is not known.

In addition, there is a new attribute on the organization entity, **Organization.QuickFindRecordLimitEnabled**. When this **Boolean** attribute is **true**, a limit is

imposed on quick find queries. If a user provides search criteria in quick find that is not selective enough, the system detects this and stops the search. This supports a faster form of quick find and can make a big performance difference.

Don't retrieve all attributes in a query because of the negative effect on performance. This is particularly true if the query is used as a parameter to an update request. In an update, if all attributes are included this sets all field values, even if they are unchanged, and often triggers cascaded updates to child records.

There are two additional ways to create queries to retrieve records from Microsoft Dynamics 365. FetchXML, the proprietary Microsoft Dynamics 365 query language, can be used to perform some queries by using XML-based queries. For more information, see [Build queries with FetchXML](#). You can also use .NET Language-Integrated Query (LINQ) to write queries. More information: [Build queries with LINQ \(.NET language-integrated query\)](#).

To save a query, you can convert it to FetchXML by using the [QueryExpressionToFetchXmlRequest](#) and save it as a saved view by using the **userqueryentity**.

Use the [QueryByAttribute](#) class

In Microsoft Dynamics 365 (online & on-premises), you can use the [QueryByAttribute](#) class to build queries that test a set of attributes against a set of values. Use this class with the [RetrieveMultiple](#) method or the [IOrganizationService.RetrieveMultipleRequest](#) method.

The following table lists the properties that you can set to create a query expression using the [QueryByAttribute](#) class.

Property	Description
<a href="#">EntityName</a>	Specifies which type of entity is retrieved. A query expression can only retrieve a collection of one entity type. You can also pass this value by using the <a href="#">QueryExpression</a> constructor.
<a href="#">ColumnSet</a>	Specifies the set of attributes (columns) to retrieve.
<a href="#">Attributes</a>	Specifies the set of attributes selected in the query.
<a href="#">Values</a>	Specifies the attribute values to look for when the query is executed.
<a href="#">Orders</a>	Specifies the order in which the records are returned from the query.
<a href="#">PageInfo</a>	Specifies the number of pages and the number of records per page returned from the query.

The following code example shows how to use the **QueryByAttribute** class.

```
// Create query using querybyattribute
QueryByAttribute querybyexpression = new QueryByAttribute("account");
querybyexpression.ColumnSet = new ColumnSet("name", "address1_city", "emailaddress1");

// Attribute to query
querybyexpression.Attributes.AddRange("address1_city");

// Value of queried attribute to return
querybyexpression.Values.AddRange("Detroit");

// Query passed to the service proxy
EntityCollection retrieved = _serviceProxy.RetrieveMultiple(querybyexpression);

// Iterate through returned collection
foreach (var c in retrieved.Entities)
{
    System.Console.WriteLine("Name: " + c.Attributes["name"]);
    System.Console.WriteLine("Address: " + c.Attributes["address1_city"]);
    System.Console.WriteLine("E-mail: " + c.Attributes["emailaddress1"]);
}
```

Use the [QueryExpression](#) class

In Microsoft Dynamics 365 and Microsoft Dynamics 365 (online), you can use the [QueryExpression](#) class to build complex queries for use with the [IOrganizationService.RetrieveMultiple](#) method or the [RetrieveMultipleRequest](#) message. You can set query parameters to the [QueryExpression](#) by using the [ConditionExpression](#), [ColumnSet](#), and [FilterExpression](#) classes.

The [QueryExpression](#) class lets you create complex queries. The [QueryByAttribute](#) class is designed to be a simple way to search for entities where attributes match specified values.

The following table lists the properties that you set to create a query expression.

Property	Description
<a href="#">EntityName</a>	Specifies which type of entity will be retrieved. A query expression can only retrieve a collection of one entity type.
<a href="#">ColumnSet</a>	Specifies the set of attributes (columns) to retrieve.
<a href="#">Criteria</a>	Specifies complex conditional and logical filter expressions that filter the results of the query.
<a href="#">Distinct</a>	Specifies whether the results of the query contain duplicate records.
<a href="#">LinkEntities</a>	Specifies the links between multiple entity types.
<a href="#">Orders</a>	Specifies the order in which the records are returned from the query.
<a href="#">PageInfo</a>	Specifies the number of pages and the number of records per page returned from the query.

### Record count

To find out how many records the query returned, set the [ReturnTotalRecordCount](#) property to true before executing the query. When you do this, the [TotalRecordCount](#) will be set. Otherwise, this value will be -1.

### Example

The following sample shows how to use the [QueryExpression](#) class.

```
// Query using ConditionExpression and FilterExpression
ConditionExpression condition1 = new ConditionExpression();
condition1.AttributeName = "lastname";
condition1.Operator = ConditionOperator.Equal;
condition1.Values.Add("Brown");

FilterExpression filter1 = new FilterExpression();
filter1.Conditions.Add(condition1);

QueryExpression query = new QueryExpression("contact");
query.ColumnSet.AddColumns("firstname", "lastname");
query.Criteria.AddFilter(filter1);

EntityCollection result1 = service.RetrieveMultiple(query);
foreach (var a in result1.Entities) {
    Console.WriteLine("Name: " + a.Attributes["firstname"] + " " + a.Attributes["lastname"]);
}
```

### Use the ColumnSet class

In Microsoft Dynamics 365 (online & on-premises), you can use the [ColumnSet](#) class to specify what attributes to return from a query expression. The query returns only non-null values.

You can also use the [ColumnSet](#) class to reduce the size of a query result by defining only those attributes to be returned. To improve server performance, it is recommended that you don't execute a query that returns all columns.

The following code example shows how to use the **ColumnSet** class to specify what attributes to return from a query expression.

```
QueryExpression contactquery = new QueryExpression
{
    EntityName="contact",
    ColumnSet = new ColumnSet("firstname", "lastname", "contactid")
};
```

### Use the ConditionExpression class

In Microsoft Dynamics 365 (online & on-premises), you can use the [ConditionExpression](#) class to compare an attribute to a value or set of values by using an operator, such as "equal to" or "greater than".

The **ConditionExpression** class lets you pass condition expressions as parameters to other classes, such as [QueryExpression](#) and [FilterExpression](#).

The following table lists the properties you can set to create a condition using the **ConditionExpression** class.

Property	Description
<a href="#">AttributeName</a>	Specifies the logical name of the attribute in the condition expression.
<a href="#">Operator</a>	Specifies the condition operator. This is set by using the <a href="#">ConditionOperator</a> enumeration.
<a href="#">Values</a>	Specifies the values of the attribute.

When using the [AddCondition](#) method (or the constructor for [ConditionExpression](#)), it's important to understand whether the array is being added as multiple values or as an array.

The following code example shows two different outcomes depending on how the array is used.

```
string[] values = new string[] { "Value1", "Value2" };
ConditionExpression c = new ConditionExpression("name", ConditionOperator.In, values);
Console.WriteLine(c.Values.Count); //This will output 2
string[] values = new string[] { "Value1", "Value2" }; object value = values;
ConditionExpression c = new ConditionExpression("name", ConditionOperator.In, value);
Console.WriteLine(c.Values.Count); //This will output 1
```

In some cases, it is necessary to cast to either **object[]** or **object**, depending on the desired behavior. When you create a condition that compares an attribute value to an enumeration, such as a state code, you must use the **ToString** method to convert the value to a string.

### Example

The following code example shows how to use the **ConditionExpression** class.

```
// Query using ConditionExpression
ConditionExpression condition1 = new ConditionExpression();
condition1.AttributeName = "lastname";
condition1.Operator = ConditionOperator.Equal;
condition1.Values.Add("Brown");
FilterExpression filter1 = new FilterExpression();
filter1.Conditions.Add(condition1);
QueryExpression query = new QueryExpression("contact");
query.ColumnSet.AddColumns("firstname", "lastname");
query.Criteria.AddFilter(filter1);
EntityCollection result1 = _serviceProxy.RetrieveMultiple(query);
Console.WriteLine();
Console.WriteLine("Query using Query Expression with ConditionExpression and FilterExpression");
Console.WriteLine("-----");
foreach (var a in result1.Entities)
{
    Console.WriteLine("Name: " + a.Attributes["firstname"] + " " + a.Attributes["lastname"]);
}
Console.WriteLine("-----");
```

#### Example

The following code example shows how to use the **ConditionExpression** class to test for the inactive state.

```
ConditionExpression condition3 = new ConditionExpression();
condition3.AttributeName = "statecode";
condition3.Operator = ConditionOperator.Equal;
condition3.Values.Add(AccountState.Active);
```

#### Use the FilterExpression class

In Microsoft Dynamics 365 and Microsoft Dynamics 365 (online), you can use the [FilterExpression](#) class to build a query that expresses multiple conditions. For example, you can create a query expression that is the equivalent of a SQL statement such as ([FirstName] = 'Joe' OR [FirstName] = 'John') AND [City] = 'Redmond'.

The following table lists the properties for the [FilterExpression](#) class.

Property	Description
<a href="#">Conditions</a>	Gets or sets condition expressions that include attributes, condition operators, and attribute values.
<a href="#">FilterOperator</a>	Gets or sets logical <b>AND/OR</b> filter operators. This is set by using the <a href="#">LogicalOperator</a> enumeration.
<a href="#">Filters</a>	Gets or sets a hierarchy of condition and logical filter expressions that filter the results of the query.
<a href="#">IsQuickFindFilter</a>	Gets or sets a value that indicates whether the expression is part of a quick find query.

The [FilterExpression](#) class also includes several helper methods that make it easier to create queries.

The [AddCondition](#) method adds a [ConditionExpression](#) to the [Conditions](#) property for the [FilterExpression](#), reducing the amount of code needed to construct the condition expression.

The [AddFilter](#) method adds a new filter to the [Filters](#) property of the [FilterExpression](#) class.

### Filter expression example

The following code example shows how to use the [FilterExpression](#) class.

```

QueryExpression query = new QueryExpression("contact");
query.ColumnSet.AddColumns("firstname", "lastname", "address1_city");

query.Criteria = new FilterExpression();
query.Criteria.AddCondition("address1_city", ConditionOperator.Equal, "Redmond");

FilterExpression childFilter = query.Criteria.AddFilter(LogicalOperator.Or);
childFilter.AddCondition("lastname", ConditionOperator.Equal, "Tharpe");
childFilter.AddCondition("lastname", ConditionOperator.Equal, "Brown");

// Pass query to service proxy
EntityCollection results = service.RetrieveMultiple(query);
Console.WriteLine();
Console.WriteLine("Query using QE with multiple conditions and filters");
Console.WriteLine("-----");

// Print results
foreach (var a in results.Entities) {
    Console.WriteLine("Name: {0} {1}", a.GetAttributeValue<string>("firstname"), a.GetAttributeValue<string>("lastname"));
    Console.WriteLine("City: {0}", a.GetAttributeValue<string>("address1_city"));
}
Console.WriteLine("-----");

```

### About the [IsQuickFindFilter](#) property

You can use the [FilterExpression.IsQuickFindFilter](#) property, that is analogous to the **isquickfindfields** attribute that exists on the **filter** node in Fetch XML. When a Fetch query is saved, this is stored in the **SavedQuery** and **UserQuery** entities **IsQuickFind** properties.

The [IsQuickFindFilter](#) property was added to provide consistency between Query Expression and Fetch XML queries.

The following rules apply to the [IsQuickFindFilter](#) property:

- This field can only be set to **true** for filter expressions with a logical operator of type [LogicalOperator.Or](#). If it is set for expressions with a logical operator of type [LogicalOperator.And](#), the [IsQuickFindFilter](#) property is ignored.



- Only one filter expression in a filter expression hierarchy can be set with [IsQuickFindFilter](#) = **true**. If more than one is found, an exception is thrown.
- If a filter expression has [IsQuickFindFilter](#) set to **true**, it cannot have any child filter expression properties, it can only have [ConditionExpression](#) properties. If you add a child filter expression, an exception is thrown.
- All condition expressions related to a filter expression with [IsQuickFindFilter](#) set to **true** must be single non-null value conditions. In other words, given that a condition is made up of attribute, operator, and value, only conditions where the value property is a single value that is not **null** are supported. In addition, the only condition operators supported on these condition expressions are ones that work with a single value that is not null. If a **null** value or multiple values are detected, an exception is thrown.

### Build queries with LINQ

In Microsoft Dynamics 365 (online & on-premises), you can use the [QueryExpression](#) class to programmatically build a query containing data filters and search conditions that define the scope of a database search. A query expression is used for single-object searches. For example, you can create a search to return all accounts that match certain search criteria. The [QueryBase](#) class is the base class for query expressions. There are two derived classes: [QueryExpression](#) and [QueryByAttribute](#).

The **QueryExpression** class supports complex queries. The **QueryByAttribute** class is a simple means to search for entities where attributes match specified values.

Query expressions are used in methods that retrieve more than one record, such as the [IOrganizationService.RetrieveMultiple](#) method, in messages that perform an operation on a result set specified by a query expression, such as [BulkDeleteRequest](#) and when the ID for a specific record is not known.

In addition, there is a new attribute on the organization entity, **Organization.QuickFindRecordLimitEnabled**. When this **Boolean** attribute is **true**, a limit is imposed on quick find queries. If a user provides search criteria in quick find that is not selective enough, the system detects this and stops the search. This supports a faster form of quick find and can make a big performance difference.

Don't retrieve all attributes in a query because of the negative effect on performance. This is particularly true if the query is used as a parameter to an update request. In an update, if all attributes are included this sets all field values, even if they are unchanged, and often triggers cascaded updates to child records.

There are two additional ways to create queries to retrieve records from Microsoft Dynamics 365. FetchXML, the proprietary Microsoft Dynamics 365 query language, can be used to perform some queries by using XML-based queries. For more information, see [Build queries with FetchXML](#). You can also use .NET Language-Integrated Query (LINQ) to write queries. More information: [Build queries with LINQ \(.NET language-integrated query\)](#).

To save a query, you can convert it to FetchXML by using the [QueryExpressionToFetchXmlRequest](#) and save it as a saved view by using the **userqueryentity**.

### LINQ limitations

The LINQ query provider supports a subset of the LINQ operators. Not all conditions that can be expressed in LINQ are supported. The following table shows some of the limitations of the basic LINQ operators.

LINQ Operator	Limitations
<b>join</b>	Represents an inner or outer join. Only left outer joins are supported.
<b>from</b>	Supports one <b>from</b> clause per query.
<b>where</b>	The left side of the clause must be an attribute name and the right side of the clause must be a value. You cannot set the left side to a constant. Both the sides of the clause cannot be constants.  Supports the <b>String</b> functions <b>Contains</b> , <b>StartsWith</b> , <b>EndsWith</b> , and <b>Equals</b> .
<b>groupBy</b>	Not supported. FetchXML supports grouping options that are not available with the LINQ query provider. More information: <a href="#">Use FetchXML aggregation</a>
<b>orderBy</b>	Supports ordering by entity attributes, such as <b>Contact.FullName</b> .
<b>select</b>	Supports anonymous types, constructors, and initializers.
<b>last</b>	The <b>last</b> operator is not supported.
<b>skip</b> and <b>take</b>	Supports <b>skip</b> and <b>take</b> using server-side paging. The <b>skip</b> value must be greater than or equal to the <b>take</b> value.
<b>aggregate</b>	Not supported. FetchXML supports aggregation options that are not available with the LINQ query provider. More information: <a href="#">Use FetchXML aggregation</a>

### Filter multiple entities

You can create complex .NET Language-Integrated Query (LINQ) queries in Microsoft Dynamics 365 and Microsoft Dynamics 365 (online). You use multiple **Join** clauses with filter clauses to create a result that is filtered on attributes from several entities.

The following sample shows how to create a LINQ query that works with two entities and filters the result based on values from each of the entities.

### Use late-bound entity class with a LINQ query

In Microsoft Dynamics 365 and Microsoft Dynamics 365 (online), you can use late binding with .NET Language-Integrated Query (LINQ) queries. Late binding uses the attribute logical name, and is resolved at runtime.

```

using (ServiceContext svcContext = new ServiceContext(_serviceProxy))
{
    var query_where3 = from c in svcContext.ContactSet
                        join a in svcContext.AccountSet
                        on c.ContactId equals a.PrimaryContactId.Id
                        where a.Name.Contains("Contoso")
                        where c.LastName.Contains("Smith")
                        select new
                        {
                            account_name = a.Name,
                            contact_name = c.LastName
                        };

    foreach (var c in query_where3)
    {
        System.Console.WriteLine("acct: " +
            c.account_name +
            "\t\t\t" +
            "contact: " +
            c.contact_name);
    }
}

```

#### *Using late binding in a join clause*

The following examples show how to use late binding in the **join** clause of a LINQ query.

Retrieve the full name of the contact that represents the primary contact for an account and the account name.

```

using (OrganizationServiceContext orgSvcContext = new OrganizationServiceContext(_serviceProxy))
{
    var query_join2 = from c in orgSvcContext.CreateQuery("contact")
                      join a in orgSvcContext.CreateQuery("account")
                      on c["contactid"] equals a["primarycontactid"]
                      select new
                      {
                          contact_name = c["fullname"],
                          account_name = a["name"]
                      };

    foreach (var c in query_join2)
    {
        System.Console.WriteLine(c.contact_name + " " + c.account_name);
    }
}

```

Retrieve Contact, Account and Lead data where the Lead was the originating Lead and the Contact's last name is not "Parker"

```
using (OrganizationServiceContext orgSvcContext = new OrganizationServiceContext(_serviceProxy))
{
    var query_dejoin = from c in orgSvcContext.CreateQuery("contact")
                        join a in orgSvcContext.CreateQuery("account")
                        on c["contactid"] equals a["primarycontactid"]
                        join l in orgSvcContext.CreateQuery("lead")
                        on a["originatingleadid"] equals l["leadid"]
                        where (string)c["lastname"] != "Parker"
                        select new { Contact = c, Account = a, Lead = l };
    foreach (var c in query_dejoin)
    {
        System.Console.WriteLine(c.Account.Attributes["name"] + " " +
            c.Contact.Attributes["fullname"] + " " + c.Lead.Attributes["leadid"]);
    }
}
```

#### *Using late binding in a left join*

The following example shows how to retrieve a list of Contact and Account information using a left join. A left join is designed to return parents with and without children from two sources. There is a correlation between parent and child, but no child may actually exist.

```
using (OrganizationServiceContext orgSvcContext = new OrganizationServiceContext(_serviceProxy))
{
    var query_join9 = from a in orgSvcContext.CreateQuery("account")
                      join c in orgSvcContext.CreateQuery("contact")
                      on a["primarycontactid"] equals c["contactid"] into gr
                      from c_joined in gr.DefaultIfEmpty()
                      select new
                      {
                          account_name = a.Attributes["name"]
                      };
    foreach (var c in query_join9)
    {
        System.Console.WriteLine(c.account_name);
    }
}
```

#### *Order results using entity attributes with LINQ*

In Microsoft Dynamics 365 and Microsoft Dynamics 365 (online), you can use lookup or OptionSet (Picklist) attributes to order results within a LINQ query. This topic shows several examples of this type of query.

#### *Using a Lookup Value to Order By*

The following sample shows use the lookup attribute **PrimaryContactId** in an **Order By** clause.

```

using (ServiceContext svcContext = new ServiceContext(_serviceProxy))
{
    var query_orderbylookup = from a in svcContext.AccountSet
                              where a.Address1_Name == "Contoso Pharmaceuticals"
                              orderby a.PrimaryContactId
                              select new
                              {
                                  a.Name,
                                  a.Address1_City
                              };

    foreach (var a in query_orderbylookup)
    {
        System.Console.WriteLine(a.Name + " " + a.Address1_City);
    }
}

```

Page large result sets with LINQ

In Microsoft Dynamics 365 (online & on-premises) you can page the results of a large .NET Language-Integrated Query (LINQ) query by using the **Take** and **Skip** operators. The **Take** operator retrieves a specified number of results and the **Skip** operator skips over a specified number of results.

#### *LINQ paging example*

The following example shows how to page the results of a LINQ query by using the **Take** and **Skip** operators:

```

int pageSize = 5;

var accountsByPage = (from a in svcContext.AccountSet
                      select new Account
                      {
                          Name = a.Name,
                      });

System.Console.WriteLine("Skip 10 accounts, then Take 5 accounts");
System.Console.WriteLine("=====");
foreach (var a in accountsByPage.Skip(2 * pageSize).Take(pageSize))
{
    System.Console.WriteLine(a.Name);
}

```

#### Query Data using the Web API

If you want to retrieve data for an entity set, use a GET request. When retrieving data, you can apply query options to set criteria for the data you want and the entity properties that should be returned.

## Basic query example

This example queries the accounts entity set and uses the **\$select** and **\$top** system query options to return the name property for the first three accounts:

### Request

```
GET [Organization URI]/api/data/v8.2/accounts?$select=name&$top=3 HTTP/1.1
Accept: application/json
OData-MaxVersion: 4.0
OData-Version: 4.0
```

### Response

```
HTTP/1.1 200 OK
Content-Type: application/json; odata.metadata=minimal
OData-Version: 4.0

{
  "@odata.context": "[Organization URI]/api/data/v8.2/$metadata#accounts(name)",
  "value": [
    {
      "@odata.etag": "W/\"501097\"",
      "name": "Fourth Coffee (sample)",
      "accountid": "89390c24-9c72-e511-80d4-00155d2a68d1"
    },
    {
      "@odata.etag": "W/\"501098\"",
      "name": "Litware, Inc. (sample)",
      "accountid": "8b390c24-9c72-e511-80d4-00155d2a68d1"
    },
    {
      "@odata.etag": "W/\"501099\"",
      "name": "Adventure Works (sample)",
      "accountid": "8d390c24-9c72-e511-80d4-00155d2a68d1"
    }
  ]
}
```

## Limits on number of entities returned

Unless you specify a smaller page size, a maximum of 5000 entities will be returned for each request. If there are more entities that match the query filter criteria, a `@odata.nextLink` property will be returned with the results. Use the value of the `@odata.nextLink` property with a new GET request to return the next page of data.

Queries on model entities aren't limited or paged. More information: [Query metadata using the Web API](#)

## Specify the number of entities to return in a page

Use the `odata.maxpagesize` preference value to request the number of entities returned in the response.

The following example queries the accounts entity set and returns the **name** property for the first three accounts.

## Request

```
GET [Organization URI]/api/data/v8.2/accounts?$select=name HTTP/1.1
Accept: application/json
OData-MaxVersion: 4.0
OData-Version: 4.0
Prefer: odata.maxpagesize=3
```

## Response

```
HTTP/1.1 200 OK
Content-Type: application/json; odata.metadata=minimal
OData-Version: 4.0
Content-Length: 402
Preference-Applied: odata.maxpagesize=3

{
  "@odata.context": "[Organization URI]/api/data/v8.2/$metadata#accounts(name)",
  "value": [
    {
      "@odata.etag": "W/\"437194\"",
      "name": "Fourth Coffee (sample)",
      "accountid": "7d51925c-cde2-e411-80db-00155d2a68cb"
    },
    {
      "@odata.etag": "W/\"437195\"",
      "name": "Litware, Inc. (sample)",
      "accountid": "7f51925c-cde2-e411-80db-00155d2a68cb"
    },
    {
      "@odata.etag": "W/\"468026\"",
      "name": "Adventure Works (sample)",
      "accountid": "8151925c-cde2-e411-80db-00155d2a68cb"
    }
  ],
  "@odata.nextLink": "[Organization URI]/api/data/v8.2/accounts?$select=name&$skiptoken=%3Ccookie%20pagenum"
}
```

Use the value of the **@odata.nextLink** property to request the next set of records. Don't change or append any additional system query options to the value. For every subsequent request for additional pages, you should use the same `odata.maxpagesize` preference value used in the original request. Also, cache the results returned or the value of the `@odata.nextLink` property so that previously retrieved pages can be returned to.

The value of the `@odata.nextLink` property is URI encoded. If you URI encode the value before you send it, the XML cookie information in the URL will cause an error.

### Apply system query options

Each of the system query options you append to the URL for the entity set is added using the syntax for query strings. The first is appended after `[?]` and subsequent query options are separated using `[&]`. All query options are case-sensitive as shown in the following example.

```
GET [Organization URI]/api/data/v8.2/accounts?$select=name,revenue&$top=3&$filter=revenue gt 100000
```

### Request specific properties

Use the **\$select** system query option to limit the properties returned as shown in the following example.

GET [Organization URI]/api/data/v8.2/accounts?\$select=name,revenue

This is a performance best practice. If properties aren't specified using \$select, all properties will be returned.

When you request certain types of properties you can expect additional read-only properties to be returned automatically.

If you request a money value, the **\_transactioncurrencyid\_value** lookup property will be returned. This property contains only the GUID value of the transaction currency so you could use this value to retrieve information about the currency using the [transactioncurrency EntityType](#). Alternatively, by requesting annotations you can also get additional data in the same request. More information: [Retrieve data about lookup properties](#)

If you request a property that is part of a composite attribute for an address, you will get the composite property as well. For example, if your query requests the **address1\_line1** property for a contact, the **address1\_composite** property will be returned as well. More information: [Composite attributes](#).

## Filter results

Use the \$filter system query option to set criteria for which entities will be returned.

## Standard filter operators

The Web API supports the standard OData filter operators listed in the following table.

Operator	Description	Example
Comparison Operators		
<b>eq</b>	Equal	<code>\$filter=revenue eq 100000</code>
<b>ne</b>	Not Equal	<code>\$filter=revenue ne 100000</code>
<b>gt</b>	Greater than	<code>\$filter=revenue gt 100000</code>
<b>ge</b>	Greater than or equal	<code>\$filter=revenue ge 100000</code>
<b>lt</b>	Less than	<code>\$filter=revenue lt 100000</code>
<b>le</b>	Less than or equal	<code>\$filter=revenue le 100000</code>
Logical Operators		
<b>and</b>	Logical and	<code>\$filter=revenue lt 100000 and revenue gt 2000</code>
<b>or</b>	Logical or	<code>\$filter=contains(name,'(sample)') or contains(name,'test')</code>
<b>not</b>	Logical negation	<code>\$filter=not contains(name,'sample')</code>
Grouping Operators		
<b>()</b>	Precedence grouping	<code>(contains(name,'sample') or contains(name,'test')) and revenue gt 5000</code>



Function	Example
contains	<code>\$filter=contains(name,'(sample)')</code>
endswith	<code>\$filter=endswith(name,'Inc.')</code>
startswith	<code>\$filter=startswith(name,'a')</code>