Event Execution Pipeline

This topic has not yet been rated -

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

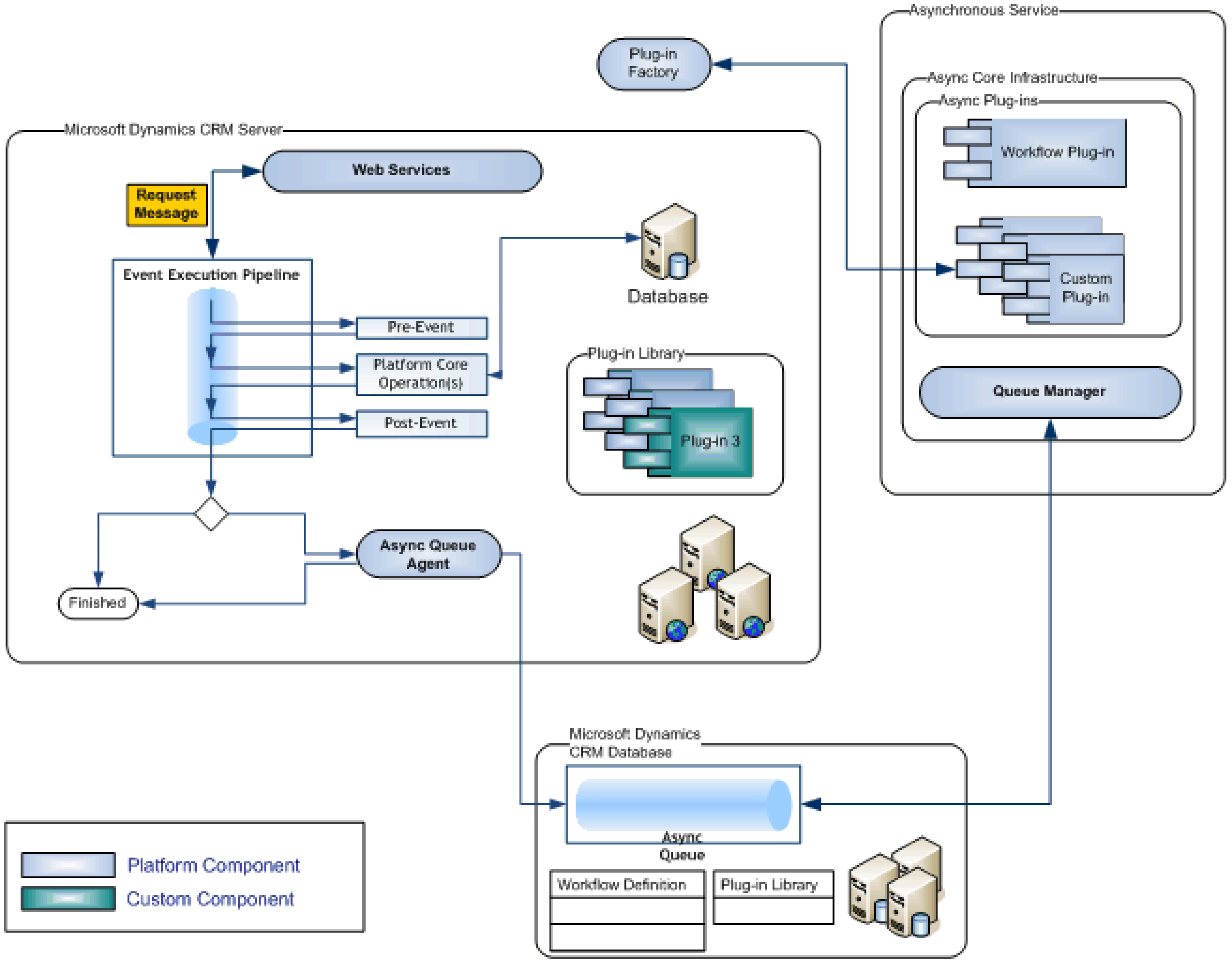
Note

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

In This Topic

Architecture and Related Components

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



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

Important

Regardless of whether a plug-in executes synchronously or asynchronously, there is a 2 minute time limit imposed on the execution of a (message) request. If the execution of your plug-in logic exceeds the time limit, a System.TimeoutException is thrown. If a plug-in needs more processing time than the 2 minute time limit, consider using a workflow or other background process to accomplish the intended task.

Pipeline Stages

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

Event	Stage name	Stage number	Description
Pre-Event	Pre-validation	10	Stage in the pipeline for plug-ins that are to execute before the main system operation. Plug-ins registered in this stage may execute outside the database transaction. The pre-validation stage occurs prior to security checks being performed to verify the calling or logged on user has the correct permissions to perform the intended operation.
Pre-Event	Pre-operation	20	Stage in the pipeline for plug-ins that are to execute before the main system operation. Plug-ins registered in this stage are executed within the database transaction.
Platform Core Operation	MainOperation	30	In-transaction main operation of the system, such as create, update, delete, and so on. No custom plug-ins can be registered in this stage. For internal use only.
Post-Event	Post-operation	40	Stage in the pipeline for plug-ins which are to execute after the main operation. Plug-ins registered in this stage are executed within the database transaction.
Post-Event	Post-operation	50	Stage in the pipeline for plug-ins which are to execute after the main operation. Plug-ins registered in this

	(Deprecated)		stage may execute outside the database transaction. This stage only supports Microsoft Dynamics CRM 4.0 based plug-ins.

Message Processing

Whenever application code or a workflow invokes a Microsoft Dynamics CRM Web service method, a state change in the system occurs that raises an event. The information passed as a parameter to the Web service method is internally packaged up into a OrganizationRequest message and processed by the pipeline. The information in the **OrganizationRequest** message is passed to the first plug-in registered for that event where it can be read or modified before being passed to the next registered plug-in for that event and so on. Plug-ins receive the message information in the form of context that is passed to their Execute method. The message is also passed to the platform core operation.

Plug-in Registration

Plug-ins can be registered to execute before or after the core platform operation. Pre-event registered plug-ins receive the **OrganizationRequest** message first and can modify the message information before the message is passed to the core operation. After the core platform operation has completed, the message is then known as the OrganizationResponse. The response is passed to the registered post-event plug-ins. Post-event plug-ins have the opportunity to modify the message before a copy of the response is passed to any registered asynchronous plug-ins. Finally, the response is returned to the application or workflow that invoked the original Web service method call.

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

Note

In Microsoft Dynamics CRM 4.0, there existed a parent and a child pipeline. These pipelines have been consolidated into one pipeline for this release. A pre-event plug-in registered in the parent pipeline of Microsoft Dynamics CRM 4.0 is equivalent to registering in stage 10. A pre-event plug-in registered in the child pipeline of Microsoft Dynamics CRM 4.0 is equivalent to registering in stage 20.

Inclusion in Database Transactions

Plug-ins may or may not execute within the database transaction of the Microsoft Dynamics CRM platform. Whether a plug-in is part of the transaction is dependent on how the message request is processed by the pipeline. You can check if the plug-in is executing in-transaction by reading the IsInTransaction property inherited by IPluginExecutionContext that is passed to the plug-in. If a plug-in is executing in the database transaction and allows an exception to be passed back to the platform, the entire transaction will be rolled back. Stages 20 and 40 are guaranteed to be part of the database transaction while stage 10 and 50 may be part of the transaction.

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

Write a Plug-In

This topic has not yet been rated - Rate this topic

Plug-ins are custom classes that implement the IPlugin interface. You can write a plug-in in any .NET Framework 4 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.

Tip

The Developer Toolkit for Microsoft Dynamics CRM 2011 and Microsoft Dynamics CRM Online provides a streamlined experience for plug-in development. For more information, see Create and Deploy Plug-ins Using the Developer Toolkit.

In This Topic

Writing a Basic Plug-in

Write a Plug-in Constructor

Support Offline Execution

Web Access for Isolated (sandboxed) Plug-ins

Use Early-Bound Types

Plug-in Assemblies

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.

C#

Copy

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

namespace Microsoft.Crm.Sdk.Samples
{
    public class SamplePlugin: IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            // Obtain the execution context from the service provider.
            IPluginExecutionContext context =
                (IPluginExecutionContext)serviceProvider.GetService(typeof(IP
luginExecutionContext));

            // Get a reference to the organization service.
            IOrganizationServiceFactory factory =
                (IOrganizationServiceFactory)serviceProvider.GetService(typeo
f(IOrganizationServiceFactory));
            IOrganizationService service = factory.CreateOrganizationService(
context.UserId);

            // Get a reference to the tracing service.
            ITracingService tracingService = (ITracingService)serviceProvider
.GetService(typeof(ITracingService));

            try
            {
                // Plug-in business logic goes below this line.
                // Invoke organization service methods.
            }
            catch (FaultException<OrganizationServiceFault> ex)
            {
                // Handle the exception.
            }
        }
    }
}
```

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 CRM 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: Basic Plug-In.

Important

For improved performance, Microsoft Dynamics CRM 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 in plug-ins or attempt to store any data in member variables for use during the next plug-in invocation.

Write a Plug-in Constructor

The Microsoft Dynamics CRM 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.

C#
Copy

```
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 CRM for Microsoft Office Outlook with Offline Access, the secure string is not passed to a plug-in that executes while Microsoft Dynamics CRM 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 CRM. 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 CRM 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 CRM 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 CRM for Microsoft Office Outlook with Offline Access is offline. The plug-in executes again when Microsoft Dynamics CRM for Outlook goes online and synchronization between Microsoft Dynamics CRM for Outlook and the Microsoft Dynamics CRM 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.

C#

Copy

```
// 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);
```

Security Note

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 CRM\Server\bin folder.

Use Early-Bound Types

To use early-bound Microsoft Dynamics CRM 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:

C#

Copy

```
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.

C#

Copy

```
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 CRM run-time event.

Security Note

In Microsoft Dynamics CRM, 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 2010 or the Strong Name tool, before being registered and deployed to Microsoft Dynamics CRM. For more information about the Strong Name tool, run the sn.exe program, without any arguments, from a Visual Studio 2010 Command Prompt window.

If your assembly contains a plug-in that can execute while the Microsoft Dynamics CRM for Outlook is offline, there is additional security that the Microsoft Dynamics CRM platform imposes on assemblies. For more information, see Walkthrough: Configure Assembly Security for an Offline Plug-In.

Understand the Data Context Passed to a Plug-In

This topic has not yet been rated - Rate this topic

IPluginExecutionContext contains information that describes the run-time environment that the plug-in is executing in, 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.

C#

Copy

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

When a system event is fired for which a plug-in is registered, the system creates and populates the context and passes it to a plug-in through the above 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 information on infinite loop prevention, refer to Depth.

In This Topic

Access to the Organization Service

To access the Microsoft Dynamics CRM organization service, it is required that plug-in code create an instance of the service through the ServiceProvider.GetService method.

C#

Copy

```
// Obtain the organization service reference.
IOrganizationServiceFactory serviceFactory = (IOrganizationServiceFactory)ser
viceProvider.GetService(typeof(IOrganizationServiceFactory));
IOrganizationService service = serviceFactory.CreateOrganizationService(conte
xt.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.

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.

C#

Copy

```
// 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 parmameters.
    Entity entity = (Entity)context.InputParameters["Target"];
```

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 and after the core platform operation. Microsoft Dynamics CRM populates the pre-entity and post-entity images based on the security privileges of the impersonated system user. You can specify to have the platform populate these 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.

There are some events where images are not available. For example, only synchronous post-event and asynchronous registered plug-ins have PostEntityImages populated. In addition, the create operation does not support a pre-image and a delete operation does not support a post-image.

Tip

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.

Note

Only entity attributes that are set to a value or **null** are available in the pre or post entity images.

Handle Exceptions in Plug-Ins

This topic has not yet been rated - Rate this topic

For synchronous plug-ins, the Microsoft Dynamics CRM platform handles exceptions passed back to the platform 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**) entity.

For plug-ins not registered in the sandbox, the exception message (System.Exception.Message) is also 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. Since the Application event log is not available to sandboxed plug-ins, sandboxed plug-ins should use tracing. For more information, seeDebug a Plug-In.

You can optionally display a custom error message in a dialog of the Web application by having your plug-in throw an InvalidPluginExecutionException exception with the custom message as the **Message** property value. It is recommended that plug-ins only pass an InvalidPluginExecutionException back to the platform.

Pass Data Between Plug-Ins

This topic has not yet been rated - Rate this topic

The message pipeline model 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.

C#
VB
Copy

```
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 SharedVariab
les
    /// property of IPluginExecutionContext.
    /// </summary>
    /// <remarks>Register the PreEventPlugin for a pre-event and the
    /// PostEventPlugin plug-in on a post-event.
    /// </remarks>
    public class PreEventPlugin : IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            // Obtain the execution context from the service provider.
            Microsoft.Xrm.Sdk.IPluginExecutionContext context = (Microsoft.Xr
m.Sdk.IPluginExecutionContext)
                serviceProvider.GetService(typeof(Microsoft.Xrm.Sdk.IPluginEx
ecutionContext));

            // Create or retrieve some data that will be needed by the post e
vent
            // 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 contex
t shared
            // variable named PrimaryContact.
            context.SharedVariables.Add("PrimaryContact", (Object)contact.ToS
tring());
        }
    }

    public class PostEventPlugin : IPlugin
    {
        public void Execute(IServiceProvider serviceProvider)
        {
            // Obtain the execution context from the service provider.
            Microsoft.Xrm.Sdk.IPluginExecutionContext context = (Microsoft.Xr
m.Sdk.IPluginExecutionContext)
                serviceProvider.GetService(typeof(Microsoft.Xrm.Sdk.IPluginEx
ecutionContext));

            // 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

0 out of 1 rated this helpful - Rate this topic

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

Plug-ins not executed by either the sandbox or asynchronous service execute under the security account that is specified on the **Identity** tab of the **CRMAppPool Properties** dialog box. The dialog box can be accessed by right-clicking the **CRMAppPool** application pool in Internet Information Services (IIS) Manager and then clicking **Properties** in the shortcut menu. By default, CRMAppPool uses the Network Service account identity but this can be changed by a system administrator during installation. If the **CRMAppPool** identity is changed to a system account other than Network Service, the new identity account must be added to the **PrivUserGroup** group in Active Directory. Refer to the "Change a Microsoft Dynamics CRM service account" topic in the Microsoft Dynamics CRM 2011 Implementation Guidefor complete and detailed instructions.

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

Impersonation during plug-in registration

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

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

Note

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 download.

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

The InitiatingUserId 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.

Important

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.

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 CRM, you must restart the service process that was executing the plug-in. Refer to the table below for the correct service process. For more information, see Register and Deploy Plug-Ins.

2. Configure the debugger.

Attach the debugger to the process on the Microsoft Dynamics CRM 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.Hoster.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, 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 CRM 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. For more information, see 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 CRM database. For more information, see Register and Deploy Plug-Ins.

Tip

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 information on 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 CRM 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**):

Copy

```
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 CRM 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.