Chapter 1

Plugins and Workflow Activities

Plugins and workflow activities are compiled as assemblies and run in the Global Assembly Cache (GAC). Some of the most complex CRM development is found in the .NET code associated with these two types of components. With CRM 2011, JScript and REST services provide a great deal of additional functionality that was not readily available in previous versions, such as creating and updating data. However, for many process and business requirements, plugin and workflow activity assemblies are still required.

This chapter outlines how to develop, register, and debug both types of components and introduces key concepts to working with data available through the CRM SDK.

Developing Plugins

To demonstrate the development of a plugin, this chapter looks at a specific business case—the deactivation of the parent account of an opportunity, triggered when the opportunity has been lost. This allows for discussion of the overall structure of a plugin, illustrated by retrieving a related entity from within a plugin, setting the state on an entity, and sending an e-mail.

Basic Code Framework for Plugins

All plugins begin with the same basic framework of code, which performs some context checking to ensure that the code fires only when expected. The basic code framework is shown in Listing 1-1, along with the additional context checks that ensure that it's triggering only on the loss of an opportunity.

Listing 1-1. Basic Plugin Framework, Executing on Loss of an Opportunity

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

```
using Microsoft.Xrm.Sdk.Query;
using Microsoft.Xrm.Sdk.Messages;
using System.Runtime.Serialization;
using Microsoft.Crm.Sdk.Messages;

namespace DemoPlugin
{
 public class Opportunity:IPlugin
 {
  public void Execute(IServiceProvider serviceProvider)
  {
   IPluginExecutionContext context =
   (IPluginExecutionContext)serviceProvider
.GetService(typeof(IPluginExecutionContext));
   if (context == null)
   {
    throw new ArgumentNullException("localContext");
   }
   IOrganizationServiceFactory serviceFactory =
(IOrganizationServiceFactory)serviceProvider
.GetService(typeof(IOrganizationServiceFactory));

   IOrganizationService service = serviceFactory
.CreateOrganizationService(context.InitiatingUserId);

   // specific code for a specific business case, you
will want to modify at this point for your own
   // business needs
   if (context.InputParameters.Contains("OpportunityClose")
    && context.InputParameters["OpportunityClose"] is Entity)
   {
    Entity EntityOpportunityClose = (Entity)context.Input
Parameters["OpportunityClose"];
    if (EntityOpportunityClose.LogicalName != "opportunityclose")
    {
     return;
    }
    if (context.MessageName == "Lose")
    {
     // core functionality goes here
    }
   }
  }
 }
}
```

Core Functionality for Plugins

Once you have the basic framework in place, you can move on to the core functionality. This plugin fires on the loss of an opportunity, which means only a small number of all of the opportunity properties are available by default. The first thing to do is query the opportunity record for additional properties that will be used in the logic of this plugin.

Querying Data

Querying an entity is very common functionality. You should break out querying into a separate function that can be called from a variety of locations. You may decide that you want to break all your common "utility" functions out into a separate class or assembly that can be referenced by any of your projects. Listing 1-2 shows the code for a generic method that can be used to query an entity by its GUID and return all its attributes.

Listing 1-2. GetEntity Method Returns All Attributes of a Specific Record

```
private Entity GetEntity(Guid
entityid,IOrganizationService service,String entity)
{
 Entity resultEntity = null;
 RetrieveMultipleRequest getRequest = new
RetrieveMultipleRequest();
 QueryExpression qex = new QueryExpression(entity);
 qex.ColumnSet = new ColumnSet() { AllColumns = true };
 qex.Criteria.FilterOperator = LogicalOperator.And;
 qex.Criteria.AddCondition(new ConditionExpression(entity
+ "id", ConditionOperator.Equal, entityid));
 getRequest.Query = qex;
 RetrieveMultipleResponse returnValues =
(RetrieveMultipleResponse)service.Execute(getRequest);
 resultEntity = returnValues.EntityCollection
.Entities[0];
 return resultEntity;
}
```

There are many ways to perform queries. In this case, the `retrieveMultiple` method is used, which will return an `EntityCollection` (array of zero to many entity records). Because the GUID is used to do the lookup, it is guaranteed that only a single record will ever be returned, so the first value in the array (`Entities[0]`) is forcibly returned.

Setting State

The next step in this plugin's flow of logic is to set the state of the associated parent account record to `InActive`. For most properties on an entity, you can set the value through a single line of code (for example, setting the value of a string property is a simple assignment line of code). But for setting state, things are substantially more involved. In this case, you want to write another method that will let you set the state on a variety of entities so that the code can be contained and reused.

Listing 1-3 shows a method that sets the state of an entity by passing in the entity name and the GUID. In this case, the entity will always be set to `InActive`, but additional parameters to this method could be added to make it dynamic, too.

Listing 1-3. Setting the State and Status of an Entity

```
private void SetEntityStatus(IOrganizationService
service, Guid recordGUID, string entityName)
{
 SetStateRequest setState = new SetStateRequest();
 setState.EntityMoniker = new EntityReference();
 setState.EntityMoniker.Id = recordGUID;
 setState.EntityMoniker.Name = entityName;
 setState.EntityMoniker.LogicalName = entityName;

 //Setting 'State' (0 - Active ; 1 - InActive)
 setState.State = new OptionSetValue();
 setState.State.Value =  1;

 //Setting 'Status' (1 - Active ; 2 - InActive)
 setState.Status = new OptionSetValue();
 setState.Status.Value =  2;
 SetStateResponse setStateResponse =
(SetStateResponse)service.Execute(setState);
}
```

Sending E-mail

CRM can send e-mail. This is done, as shown in Listing 1-4, with the following actions:

1. The opportunity record is passed in, along with the service information and the opportunity ID.
2. A message body is assigned (this can be dynamic and can include HTML for formatting). The message body will be the body of the e-mail.

3. The To and From e-mail addresses are assigned by setting the entity GUID for the system user and assigning them to an `ActivityParty` object.
4. The regarding object is set to the opportunity (this allows the e-mail to be linked to the opportunity record in CRM for tracking purposes).
5. Finally, the e-mail properties are set, and the e-mail is created and sent.

Listing 1-4. Creating an E-mail

```
public void CreateEmail(Entity opportunity,
IOrganizationService service,Guid oppId)
{
 string msg = "Dear Owner: <br/><br/> Please review the
opportunity.";
 string recipient = opportunity.Attributes["new_
manager"].ToString();

 ActivityParty fromParty = new ActivityParty
 {
  PartyId = new EntityReference("systemuser",new
Guid("620FA4D5-1656-4DDF-946F-20E6B1F19447"))
 };

 ActivityParty toParty = new ActivityParty
 {
  PartyId = new EntityReference("systemuser",  new
Guid(recipient))
 };

 EntityReference regarding = new
EntityReference("opportunity", oppId);

 Email email = new Email
 {
  To = new ActivityParty[] { toParty },
  From = new ActivityParty[] { fromParty },
  Subject = opportunity.Attributes["new_
titleanddescription"].ToString(),
  Description = msg,
  RegardingObjectId = regarding
 };

 service.Create(email);
}
```

Tying It Together

With all the methods defined, and the core plugin framework code in place, you simply have to tie it together with the code shown in Listing 1-5. This code replaces the `// core functionality goes here` comment in Listing 1-1.

Listing 1-5. Core Functionality of the Plugin: Tying It Together

```
if (context.MessageName == "Lose")
{
 Entity Opportunity = null;
 Opportunity = GetEntity(((Microsoft.Xrm.Sdk
.EntityReference)
  (EntityOpportunityClose.Attributes["opportunityid"]))
.Id, service,"opportunity");

 // if the opportunity record has a parent account
associated with it, continue the logic execution
 if(Opportunity.Contains("parentaccountid"))
 {
  parentAccount = GetEntity(((Microsoft.Xrm.Sdk
.EntityReference)
    (Opportunity.Attributes["parentaccountid"])).Id,
service, "account");

  SetEntityStatus(service, ((Microsoft.Xrm.Sdk
.EntityReference)
    (Opportunity.Attributes["parentaccountid"])).Id,
"account");

  CreateEmail(opportunity, service,
EntityOpportunityClose.Attributes["opportunityid"])).Id);
 }
}
```

Developing Workflow Activities

Workflow activity development is very similar to plugin development. Differences include the class (which inherits from **`CodeActivity`**) and the way the context, service, and service factory are created. The code in Listing 1-6 shows the framework for a workflow. The comment **`// core functionality goes here`** is where the custom code for each workflow activity you are creating begins.

Listing 1-6. Workflow Activity

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Activities;
using Microsoft.Xrm.Sdk;
using Microsoft.Xrm.Sdk.Workflow;
using Microsoft.Xrm.Sdk.Query;
using Microsoft.Xrm.Sdk.Messages;
using Microsoft.Xrm.Sdk.Client;
using Microsoft.Xrm.Sdk.Discovery;
using System.Runtime.Serialization;
using Microsoft.Crm.Sdk.Messages;

namespace LeaseWorkflowActivity
{
 public class LeaseRecurringWorkflow: CodeActivity
 {
  protected override void Execute(CodeActivityContext
executionContext)
  {
   IWorkflowContext context = executionContext
.GetExtension<IWorkflowContext>();
   IOrganizationServiceFactory serviceFactory =
executionContext.GetExtension<IOrganizationServiceFactory>();

   IOrganizationService service = serviceFactory
.CreateOrganizationService(context.UserId);

   // core functionality goes here
  }
 }
}
```

Note CRM Online doesn't provide a way to schedule workflows. If you want to set up a recurring workflow (which is a common way to call workflow activities repeatedly), you need to configure your workflow as both an *on demand* and a *child* workflow (see Figure 1-1). You then need to set the wait state and duration, as shown in Figure 1-2. To initiate the workflow, you need to kick it off manually (on demand). Once started, it will continue to call itself as indicated in the schedule. Remember: CRM has a built-in process that ensures infinite loops are terminated. This means for recurring workflows, you can't schedule the workflow to occur more frequently than seven times in an hour—anything more will cause the workflow to terminate.

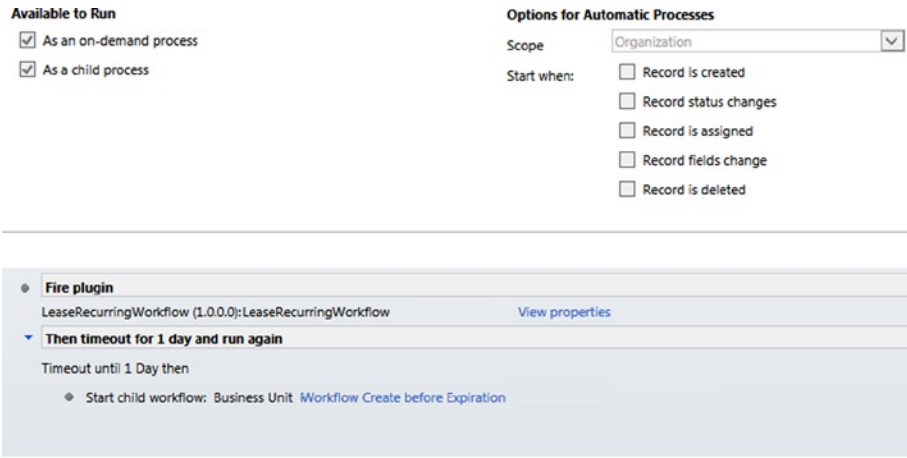

Figure 1-1. Allowing for a recurring workflow in CRM Online

Figure 1-2. The Workflow Scheduler

Plugin and Workflow Activity Registration

The Plugin Registration tool is critical component of plugins and workflow activity registration—yet it seems almost an afterthought. Over the years, the tool has gone through several updates, but it exists as a separate application from CRM and has a true "developer" feel about it. Though it looks and feels like something that isn't quite production ready, the tool's functionality and purpose are absolutely vital to the success of deploying and configuring your components.

Setting Up the Plugin Registration Tool

The registration tool (pluginregistration.exe) is located in the SDK\bin folder of the CRM SDK, which you can download from `www.microsoft.com/en-us/download/details.aspx?id=24004`. Once you run this application, you're asked to set several properties. Take the following steps to set up your connection:

1. Click Create New Connection.
2. In the Label property, give the connection an appropriate name (such as the name of the environment you are connecting to).
3. In the Discovery URL property, set the value to what is specific to your environmental settings. To get this value, click Developer Resources in the Customization settings of your CRM instance and look at Service Endpoints. You will find the URL value under Discovery Service, as shown in Figure 1-3. Enter the value of what you find before the /XRMServices folder (for example, in Figure 1-3, the value would be `https://disco.crm.dynamics.com`).

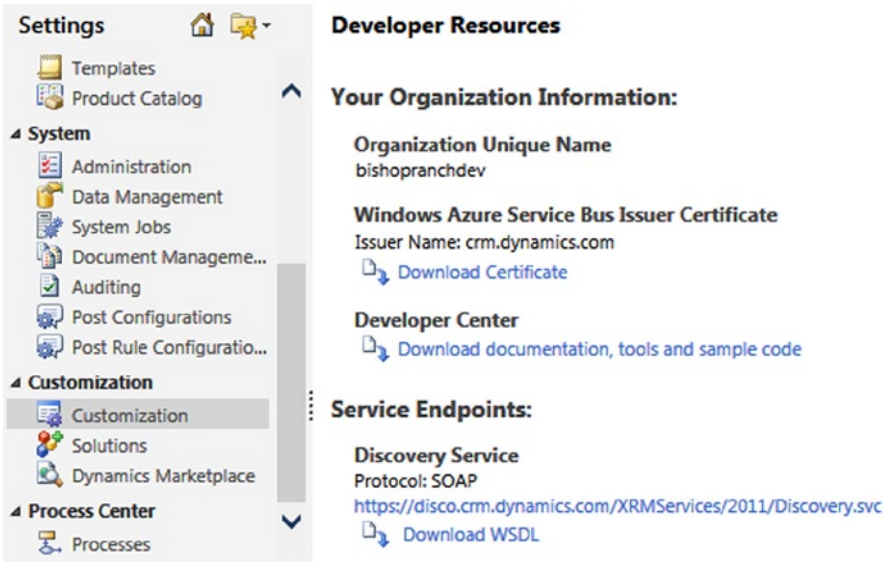

Figure 1-3. Discovery Service URL in Developer Resources

4. Enter your credentials in the User Name field (you will be prompted for a password after you click the Connect button).

Once you have successfully connected to your environment, a new connection will appear under Connections in your Plugin Registration tool. When you have connected to this new connection, you will see all the registered plugins and custom workflow activities listed in the window on the right, as shown in Figure 1-4.

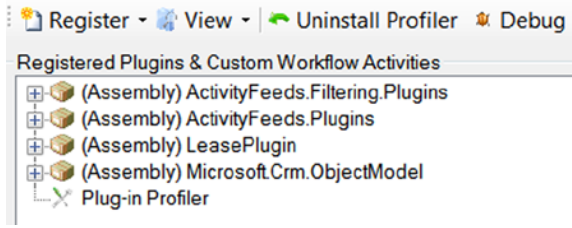

Figure 1-4. Viewing registered components

Registering an Assembly

To register a plugin or workflow activity, click the Register option on the toolbar and select Register New Assembly (shown in Figure 1-5). The Register New Plugin screen that pops up is used for plugin and for workflow activity registration. Browse

to your DLL (which may contain one or more plugins and activities) and click Load Assembly. Doing so lets you select which components to register and where to register them to.

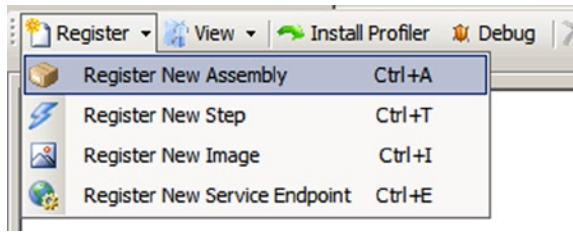

Figure 1-5. Register New Assembly

Next, set the Isolation Mode property to Sandbox. (In CRM Online, this value is always set to Sandbox, but in CRM on-premise you can chose between Sandbox and None, which allows for some level of control over trust level execution.) Set the Location where the Assembly should be stored to Database. Take note that when deploying to the database, you must also copy the DLL and PDB files to the \Server\bin\assembly folder of your CRM installation to be able to debug. (This is noted in the Database setting area, shown in Figure 1-6.)

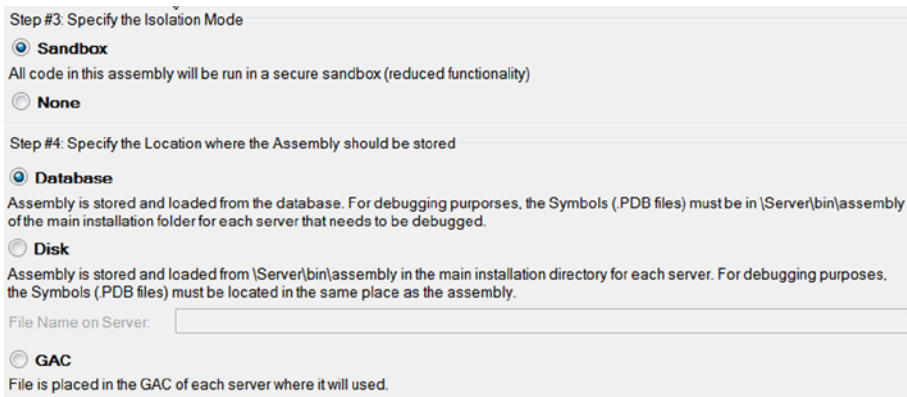

Figure 1-6. Sandbox and Database settings when registering plugin

> **Note** When a workflow activity has been registered, it's marked with (Workflow Activity). Nothing further needs to be done once the assembly has been loaded (no steps need to be associated with it). See Figure 1-7.

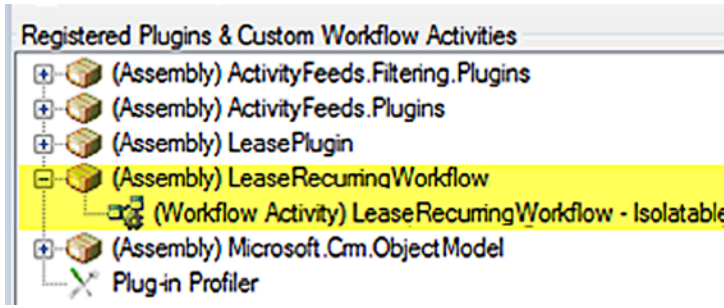

Figure 1-7. A registered workflow activity

Registering a Step

Now that the assembly has been registered, next you must configure the settings for when it will execute by setting up one or more steps. For this example, take the opportunity plugin outlined earlier in this chapter. The business case for this plugin is that when an opportunity is lost, the associated account record will be deactivated. For the registration configuration, that means you have to set up a new step that triggers on the loss of an opportunity. Figure 1-8 shows how to configure the step for this scenario.

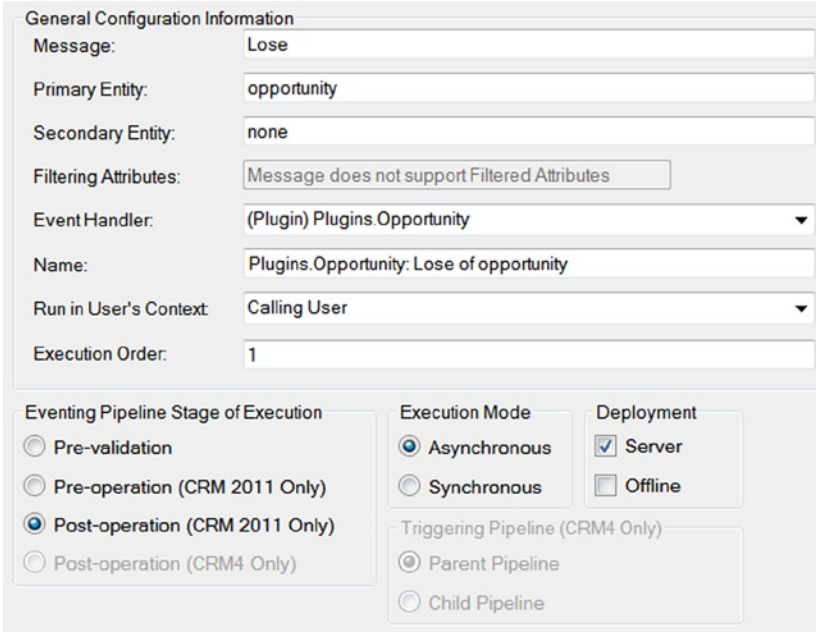

Figure 1-8. Configuring a step

Here are some of the key properties that are set here:

1. *Message:* This is the *event*occurring that will trigger the plugin to fire. There are dozens of potential events—or messages—that can be subscribed to. Press a single letter, and all the available messages starting with that letter are shown. In this case, when you press L, you see that Lose is one of three options.
2. *Primary Entity:* This is the name of the CRM entity that you will be registering the step to fire in conjunction with. All of the standard and custom entities within your environment will be listed here. In this case, type *opportunity*.
3. *Event Handler and Name:* These should default, but make sure they're set to appropriate values for your step.
4. *Run in User's Context:* This property is important. It determines the permissions that the plugin will fire under. If you want a plugin to execute exactly the same, regardless of which user is performing an action, set this to an account that has administrative privileges (such as CRM Admin). If you want this to be restricted to security settings that the user has defined, set it to Calling User.
5. *Eventing Pipeline Stage of Execution:* This setting will depend on what message you're setting the step to trigger on and when you want your plugin to fire. The loss of an opportunity should fire after the entity has been updated, so set this to Post-operation. Alternatively, you may want to set up a step that triggers on the creation of a new opportunity, and you need the plugin to fire before the opportunity is created. In that case, you would select Pre-validation.
6. *Execution Mode:* You can set this to Synchronous or Asynchronous. If you need to ensure that the plugin executes in full before the user can do anything else in CRM, set it to Synchronous.

■ **Note** Registering a workflow activity doesn't require the additional step of registering a step. A workflow activity will execute from a step within a defined process in the CRM user interface, as shown in Figure 1-9. To access this, create a new workflow (process) from the Process menu option in Settings in the CRM user interface.

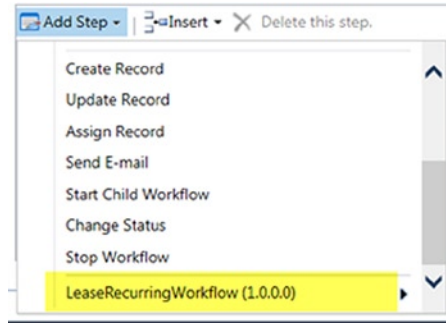

Figure 1-9. Workflow activity will be available in the Add Step window

Debugging Plugins

After a plugin has been registered properly in a development setting, generally you want to work through debugging. Debugging a CRM on-premise solution is similar to debugging any ASP.NET application. And it requires fewer steps than debugging CRM Online. Debugging Online can be more time-consuming and therefore requires more thought and care in development. The following sections outline how to debug in both environments.

Debugging CRM On-premise Solutions

When debugging on-premise, you can use Visual Studio to attach to a process and step through the code. The following steps are all that's required:

1. Make sure that the DLL and PDB that are built with your Visual Studio solution are current, and that the current builds of both are in the server/bin folder of your on-premise CRM installation folder. You need to register the plugins using the Plugin Registration tool prior to taking any further steps.
2. Click the Debug menu option in Visual Studio and select Attach to Process (see Figure 1-10). You want to attach to all instances of w3wp.exe listed in the Available Processes panel.

> **Note** When debugging workflow activities, you need to attach to the `CrmAsyncService` process in the Available Processes panel (w3wp.exe is the process to attach to for plugin debugging).

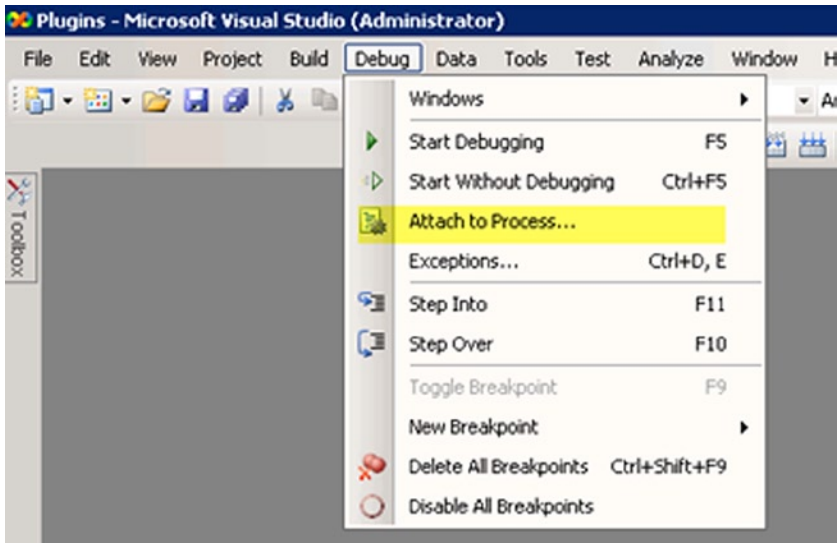

Figure 1-10. Attach to Process

> **Note** When debugging on a remote server, you need to use the Visual Studio Remote Debugger and configure Visual Studio to connect to the target machine.

Debugging CRM Online Plugins

CRM Online can't be debugged in the same way as CRM on-premise because there is no way to attach the Visual Studio environment directly to CRM Online. Instead, you need to take several steps to allow for debugging. This section describes those steps.

Installing the Profiler

The first step to debugging in the CRM Online environment is to install the profiler. You do that by opening the Plugin Registration tool and connecting to the instance that has the plugin you want to debug. Click the Install Profiler button on the menu bar (see Figure 1-11).

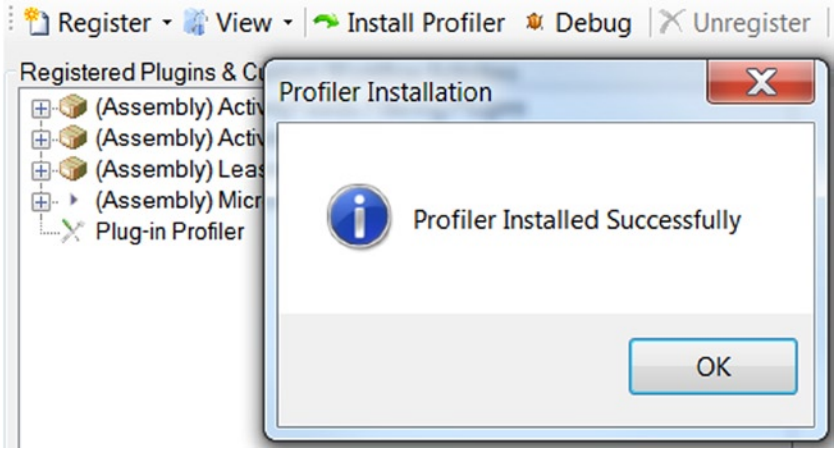

Figure 1-11. Installing the profiler for debugging

> **Note** To uninstall the profiler, make sure you click the Uninstall Profiler button in the Plugin Registration tool. If you simply unregister it (as you would anything else registered), you'll leave a lot of pieces installed and will have some additional cleanup work to do.

Profiling a Step

With the profiler installed, the next step is to right-click the step within a plugin that you want to debug and select the Start Profiling option. In the Profiler Settings (shown in Figure 1-12), you can set various properties. You can generally use the default settings here and simply click OK. Once a step has been configured with the profiler, it will have a note next to it in the Plugin Registration tool that says (Profiled) (see Figure 1-13). You can add profiling to as many steps as you need to successfully debug your code.

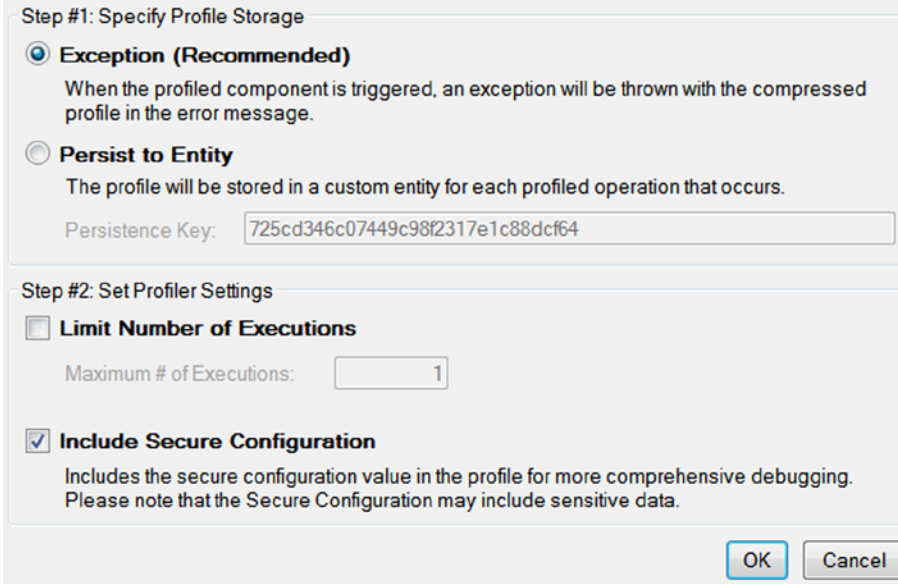

Figure 1-12. Profiler Settings on Step

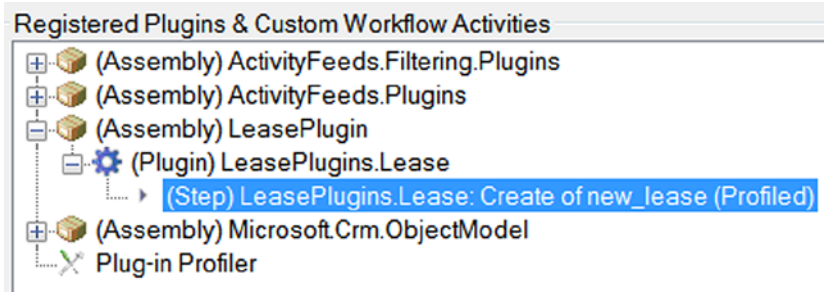

Figure 1-13. A step with profiler attached (Profiled)

Triggering the Profiler and Saving the Log File

Now that the step is successfully profiled, you need to trigger the code behind the step to execute. Go into the CRM Online front end and force the action to occur that will trigger the step (for example, if the step is on the loss of an opportunity, go into an opportunity and click Lost Opportunity). When the code is hit, an error similar to that shown in Figure 1-14 will pop up. Click the Download Log File button and save it.

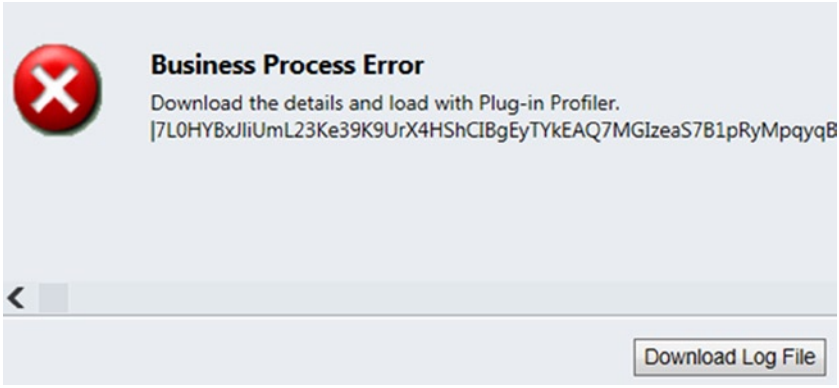

Figure 1-14. Error indicating that the profiler was successfully triggered

Debugging, Attaching, and Breakpoints

Back in the Plugin Registration tool, click the Debug button (on the menu, next to the Uninstall Profiler button). The Debug Existing Plug-in dialogue box shown in Figure 1-15 opens. Set the Profile Location property to the local path where you downloaded the log file (Figure 1-14). The Assembly Location property should be the local path where the assembly DLL and PDB files are located. Make sure that this is the same build that the Visual Studio code you'll be debugging is associated with.

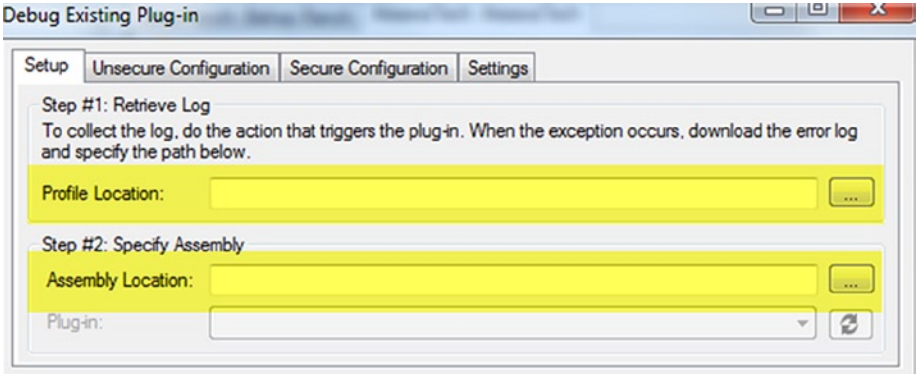

Figure 1-15. Debug Existing Plug-in dialog box

Open your plugin code in Visual Studio and set breakpoints where appropriate. You should start by putting a breakpoint on the first piece of code that will execute when triggered, so that you can ensure all your code is firing as expected.

Once your breakpoints are set, click the Debug option on the menu bar in Visual Studio and select Attach to Process. In the Available Processes panel, highlight the pluginregistration.exe process and click the Attach button.

Finally, once everything is attached in Visual Studio, click the Start Execution button in the Debug Existing Plug-in tool (Figure 1-16). You will now be able to step through your code.

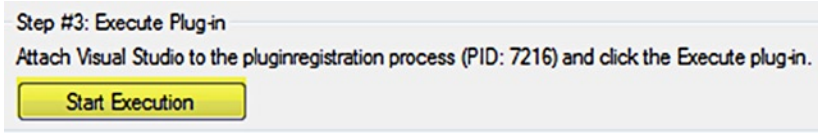

Figure 1-16. Start Execution

Conclusion

You've looked at how to develop and register plugins and workflow activities and have worked through some of the critical aspects to querying and modifying data. You've also stepped through debugging plugins in on-premise and CRM Online environments. Plugins and workflow activities will be some of the most labor-intensive activities that you do in CRM. But they can be optimized and made maintainable through intelligent planning and coding. Developing these components introduces you to interacting with CRM through the SDK, which is essential to building external applications that can pull data from and write records to CRM. Those topics are covered in the next chapter.