Dynamics 365 Customer Engagement (CE) Plugins: A Comprehensive Guide

Table of Contents

- Online Communities and Forums

Chapter 1: Introduction

What is Dynamics 365 CE?

Dynamics 365 Customer Engagement (CE) is a suite of applications designed to help businesses manage their customer relationships and data. It includes various modules such as Sales, Customer Service, Marketing, and Field Service, all integrated to provide a comprehensive CRM solution. These applications enable organizations to streamline their customer interactions, automate business processes, and gain insights from customer data.

Dynamics 365 CE is part of the broader Dynamics 365 suite, which includes Enterprise Resource Planning (ERP) solutions. The CE applications are highly customizable and can be extended to meet specific business needs through custom code, configurations, and integrations.

Importance of Plugins in Dynamics 365 CE

Plugins are a vital component of Dynamics 365 CE because they allow developers to inject custom business logic into the platform. These custom logic executions occur in response to various events within the system, such as creating, updating, or deleting records. Plugins enable organizations to tailor the behavior of Dynamics 365 CE to their unique business requirements, ensuring that the system works seamlessly with their processes.

For example, plugins can be used to:

- Automatically calculate and update field values based on specific business rules.
- Integrate Dynamics 365 CE with external systems.
- Validate data before it is saved to ensure data integrity.
- Implement complex business workflows and processes.

By leveraging plugins, businesses can significantly enhance the functionality and efficiency of their Dynamics 365 CE applications.

Prerequisites for Learning Plugins

To effectively develop and work with plugins in Dynamics 365 CE, you should have a foundational understanding of the following:

- **Dynamics 365 CE Environment:** Familiarity with the Dynamics 365 CE interface, entities, fields, and basic configurations.
- **C# Programming Language:** Plugins are written in C#, so a good grasp of C# syntax, object-oriented programming, and .NET framework is essential.
- **Microsoft Visual Studio:** Knowledge of using Visual Studio for developing and debugging C# projects.
- **Basic Concepts of CRM and Business Processes:** Understanding how CRM systems work and the common business processes they support.

Chapter 2: Getting Started with Dynamics 365 CE Plugins

Introduction to Plugins

Plugins in Dynamics 365 CE are custom business logic components that execute in response to specific events triggered by the platform. These events can occur during various operations such as creating, updating, deleting, or retrieving records. Plugins are .NET assemblies that implement the IPlugin interface, which defines the Execute method where the custom logic is written.

There are several key concepts to understand about plugins:

- **Event Pipeline:** Plugins execute within a well-defined event pipeline that determines the order of operations.
- **Synchronous vs Asynchronous Execution:** Plugins can run synchronously (blocking the main operation until the plugin completes) or asynchronously (executing in the background without blocking the main operation).
- **Stages of Execution:** Plugins can be registered to run in different stages of the event pipeline: Pre-Validation, Pre-Operation, and Post-Operation.

By understanding these concepts, you can develop plugins that are efficient, reliable, and tailored to your business needs.

Plugin Development Environment Setup

Before you start developing plugins for Dynamics 365 CE, you need to set up your development environment. Here's a step-by-step guide to get you started:

1. **Install Visual Studio:**
    - Download and install the latest version of Visual Studio. The Community edition is sufficient for most plugin development tasks.
2. **Download and Install the Dynamics 365 SDK:**
    - The Dynamics 365 Software Development Kit (SDK) provides tools and resources for developing custom solutions. You can download it from the Microsoft Download Center.
3. **Set Up the Plugin Registration Tool:**
    - The Plugin Registration Tool (PRT) is part of the SDK and is used to register plugins with the Dynamics 365 CE platform. Extract the SDK zip file, navigate to the Tools folder, and run the PluginRegistration.exe file.
4. **Connect to Your Dynamics 365 Instance:**
    - Open the Plugin Registration Tool and click on the + Create New Connection button. Enter the necessary details to connect to your Dynamics 365 CE instance. Once connected, you'll be able to see the list of registered plugins, steps, and assemblies.

Creating Your First Plugin

Let's walk through the process of creating a simple plugin that triggers on the creation of a new account record and updates a custom field.

1. **Create a New Class Library Project in Visual Studio:**
    - Open Visual Studio and create a new project. Select Class Library and name it MyFirstPlugin.
2. **Add Necessary References:**
    - Add references to the Microsoft.Xrm.Sdk and Microsoft.Crm.Sdk.Proxy assemblies. These can be found in the SDK's bin folder.
3. **Implement the IPlugin Interface:**
    - Create a new class that implements the IPlugin interface. This interface requires you to implement the Execute method.

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

namespace MyFirstPlugin
{
  public class AccountPlugin : IPlugin
  {
    public void Execute(IServiceProvider serviceProvider)
    {
      // 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 account.
        if (entity.LogicalName != "account")
          return;

        // Update a custom field.
        entity["new_customfield"] = "Plugin executed!";
      }
    }
  }
}
```

4. **Build the Project:**
    - Build the project to generate the assembly (DLL file).

Registering Plugins Using Plugin Registration Tool

Now that you have created your first plugin, the next step is to register it with Dynamics 365 CE using the Plugin Registration Tool.

1. **Open the Plugin Registration Tool:**
    - Connect to your Dynamics 365 CE instance.

2. **Register the Assembly:**
    - Click on Register > Register New Assembly.
    - Browse to the DLL file generated from your Visual Studio project and select it.
    - Enter a name and description for the assembly and click Register.
3. **Create a Step:**
    - After registering the assembly, you need to create a step that specifies when the plugin should execute.
    - Right-click on the newly registered assembly and select Register New Step.
    - Fill in the details, such as the message (e.g., Create), primary entity (e.g., account), and execution stage (e.g., Post-Operation).
    - Click Register New Step to complete the registration.

Your plugin is now registered and will execute whenever a new account record is created in Dynamics 365 CE.

Chapter 3: Plugin Architecture and Lifecycle

Understanding Plugin Architecture

Plugins in Dynamics 365 CE are .NET assemblies that execute custom business logic in response to specific events triggered by the platform. These events can be operations such as creating, updating, deleting, or retrieving records. The plugins are written in C# and implement the IPlugin interface, which defines a single method, Execute, where the custom logic resides.

Plugins can be registered to execute during different stages of the event pipeline, which consists of three primary stages:

- **Pre-Validation:** This stage occurs before the main system operation begins and before any security checks are performed. Plugins registered in this stage execute outside the database transaction.
- **Pre-Operation:** This stage occurs before the main system operation but after security checks. Plugins registered here execute within the database transaction.
- **Post-Operation:** This stage occurs after the main system operation has completed. Plugins registered in this stage execute within the database transaction.

Plugin Execution Pipeline

The plugin execution pipeline determines the order in which plugins and other operations are executed in response to an event. Understanding the pipeline is crucial for developing effective plugins.

1. **Pre-Validation Stage:**
    - Runs before any system validation.
    - Ideal for performing custom validation logic.
2. **Pre-Operation Stage:**
    - Runs after validation but before the main operation.
    - Used for modifying the input data before it is processed.
3. **Main Operation:**
    - The core operation, such as creating, updating, or deleting a record.
4. **Post-Operation Stage:**
    - Runs after the main operation has completed.
    - Useful for operations that depend on the result of the main operation, such as sending notifications or updating related records.

Synchronous vs Asynchronous Plugins

Plugins can be executed synchronously or asynchronously, depending on the business requirements and performance considerations.

- **Synchronous Plugins:**
    - Execute immediately and block the main operation until the plugin completes.
    - Suitable for scenarios where immediate processing is required.
    - May impact performance if the plugin logic is complex or time-consuming.

- **Asynchronous Plugins:**
    - Execute in the background without blocking the main operation.
    - Ideal for operations that do not require immediate execution, such as sending emails or performing batch updates.
    - Improve system performance by offloading processing to background tasks.

Pre-Validation, Pre-Operation, and Post-Operation Stages

Plugins can be registered to execute during different stages of the event pipeline, each serving different purposes:

- **Pre-Validation Stage:**
    - Runs before any security checks and the main operation.
    - Useful for custom validation logic.
    - Executes outside the database transaction, meaning changes made here are not rolled back if the main operation fails.
- **Pre-Operation Stage:**
    - Runs after validation but before the main operation.
    - Ideal for modifying input data before it is processed.
    - Executes within the database transaction, so changes are rolled back if the main operation fails.
- **Post-Operation Stage:**
    - Runs after the main operation has completed.
    - Suitable for actions dependent on the results of the main operation, such as sending notifications or updating related records.
    - Executes within the database transaction.

Chapter 4: Plugin Development Best Practices

Writing Efficient Plugins

Efficiency is crucial for ensuring that plugins do not degrade system performance. Here are some best practices for writing efficient plugins:

1. **Minimize Resource Usage:**
    - Use the Service Locator pattern to instantiate required services only when needed.
    - Avoid retrieving more data than necessary.
2. **Optimize Queries:**
    - Use fetch XML or QueryExpression to retrieve only the required fields.
    - Implement paging for large data sets.
3. **Use Caching:**
    - Cache frequently accessed data to reduce repeated calls to the database.
4. **Avoid Long-Running Operations:**
    - Offload long-running tasks to asynchronous plugins or background services.

Error Handling and Logging

Proper error handling and logging are essential for diagnosing issues and ensuring reliable plugin execution.

1. **Implement Robust Error Handling:**
    - Use try-catch blocks to handle exceptions.
    - Provide meaningful error messages to help diagnose issues.
2. **Use Tracing and Logging:**
    - Utilize the ITracingService to log detailed information about plugin execution.
    - Implement custom logging to track plugin performance and errors.
3. **Graceful Degradation:**
    - Design plugins to handle failures gracefully, ensuring that the main operation can continue if possible.

Security Considerations

Security is a critical aspect of plugin development. Here are some key considerations:

1. **Avoid Hardcoding Sensitive Information:**
    - Use secure and unsecure configuration parameters to store sensitive data.
2. **Validate User Input:**
    - Ensure that all user inputs are validated to prevent security vulnerabilities such as SQL injection.
3. **Implement Role-Based Security:**
    - Check user roles and permissions before performing operations to ensure compliance with security policies.

Performance Optimization

Optimizing plugin performance is essential for maintaining system responsiveness and user satisfaction.

1. **Profile and Analyze Plugins:**
    - Use profiling tools to identify performance bottlenecks.
    - Analyze plugin execution times and optimize code accordingly.
2. **Optimize Database Access:**
    - Reduce the number of database calls by batching operations where possible.
    - Use appropriate indexes to speed up data retrieval.
3. **Leverage Asynchronous Execution:**
    - Offload non-critical operations to asynchronous plugins to improve overall system performance.

Chapter 5: Advanced Plugin Development

Working with Secure and Unsecure Configuration

Plugins can be configured with secure and unsecure configuration data, providing flexibility in managing sensitive information.

1. **Secure Configuration:**
    - Accessible only to users with the System Administrator role.
    - Use for storing sensitive data, such as connection strings or credentials.
2. **Unsecure Configuration:**
    - Accessible to all users.
    - Use for storing non-sensitive data, such as plugin settings or thresholds.

Using Shared Variables

Shared variables allow data to be passed between plugins and other components during a single transaction.

1. **Setting Shared Variables:**
    - Use the SharedVariables property of the plugin execution context to store data.

```
context.SharedVariables["key"] = value;
```

2. **Retrieving Shared Variables:**
    - Access shared variables in subsequent plugins or custom workflow activities.

```
var value = context.SharedVariables["key"];
```

Implementing Custom Workflow Activities

Custom workflow activities extend the functionality of Dynamics 365 CE workflows by allowing developers to create reusable components with custom business logic.

1. **Create a Custom Workflow Activity:**
    - Implement the CodeActivity class and override the Execute method.

```
public class CustomWorkflowActivity : CodeActivity
{
  protected override void Execute(CodeActivityContext context)
  {
    // Custom business logic
  }
}
```

2. **Register the Custom Workflow Activity:**
    - Use the Plugin Registration Tool to register the custom workflow activity assembly.
    - Add the custom workflow activity to a workflow using the workflow designer.

Debugging Plugins

Effective debugging is essential for developing reliable plugins. Here are some techniques for debugging plugins:

1. **Attach Visual Studio to the Dynamics 365 Process:**
   - Use Visual Studio to attach to the w3wp.exe process (for on-premises) or the CrmAsyncService.exe process (for online).
2. **Use the Plugin Profiler:**
   - The Plugin Profiler is a tool provided by the SDK that allows you to profile and debug plugin execution.

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

namespace MyFirstPlugin
{
  public class AccountPlugin : IPlugin
  {
    public void Execute(IServiceProvider serviceProvider)
    {
      ITracingService tracingService =
(ITracingService)serviceProvider.GetService(typeof(ITracingService));
      tracingService.Trace("Plugin execution started");

      try
      {
        IPluginExecutionContext context =
(IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));

        if (context.InputParameters.Contains("Target") && context.InputParameters["Target"]
is Entity)
        {
          Entity entity = (Entity)context.InputParameters["Target"];

          if (entity.LogicalName != "account")
            return;

          entity["new_customfield"] = "Plugin executed!";
        }
      }
      catch (Exception ex)
      {
        tracingService.Trace("Exception: {0}", ex.ToString());
        throw;
      }
      finally
      {
        tracingService.Trace("Plugin execution finished");
      }
    } }}
```

Chapter 6: Real-World Examples and Use Cases

Common Scenarios for Plugin Usage

Plugins are versatile and can be used in various scenarios to enhance the functionality of Dynamics 365 CE. Here are some common use cases:

1. **Automated Calculations:**
    - Calculate and update field values based on specific business rules. For example, compute a discount percentage based on the total sales amount.
2. **Data Validation:**
    - Validate data before it is saved to ensure consistency and accuracy. For instance, check if a custom field meets certain criteria before allowing a record to be created.
3. **Integration with External Systems:**
    - Integrate Dynamics 365 CE with external systems, such as sending data to an external API or synchronizing records with another CRM system.
4. **Custom Business Logic:**
    - Implement custom workflows that are not available out-of-the-box. For example, trigger specific actions based on changes in related records.

Example: Auto Number Generation

Scenario: You need to generate an auto-incremented number for a custom field every time a new record is created.

Solution: Implement a plugin that runs in the Post-Operation stage of the Create event. This plugin will retrieve the last used number, increment it, and update the new record with the new number.

Code Example:

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

namespace MyFirstPlugin
{
  public class AutoNumberPlugin : IPlugin
  {
    public void Execute(IServiceProvider serviceProvider)
    {
      ITracingService tracingService =
(ITracingService)serviceProvider.GetService(typeof(ITracingService));
      IPluginExecutionContext context =
(IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));
      IOrganizationServiceFactory serviceFactory =
(IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));
      IOrganizationService service = serviceFactory.CreateOrganizationService(context.UserId);
```

```
        if (context.InputParameters.Contains("Target") && context.InputParameters["Target"] is Entity)
        {
            Entity entity = (Entity)context.InputParameters["Target"];

            if (entity.LogicalName == "your_custom_entity")
            {
                // Generate auto number
                string autoNumber = GenerateAutoNumber(service);

                // Update the custom field with the auto number
                Entity updateEntity = new Entity(entity.LogicalName, entity.Id);
                updateEntity["your_custom_field"] = autoNumber;
                service.Update(updateEntity);
            }
        }
    }

    private string GenerateAutoNumber(IOrganizationService service)
    {
        // Retrieve the last used number and increment
        // Implement your logic here
        return "AUTO-0001"; // Example return value
    }
  }
}
```

Example: Data Validation and Enrichment

Scenario: Validate and enrich data before saving a record. For instance, ensure that a contact's email address is valid and format the phone number correctly.

Solution: Implement a plugin that runs in the Pre-Operation stage of the Create and Update events. This plugin will validate and format the data before it is saved.

Code Example:

```
using System;
using System.Text.RegularExpressions;
using Microsoft.Xrm.Sdk;

namespace MyFirstPlugin
{
  public class DataValidationPlugin : IPlugin
  {
    public void Execute(IServiceProvider serviceProvider)
    {
      ITracingService tracingService =
(ITracingService)serviceProvider.GetService(typeof(ITracingService));
      IPluginExecutionContext context =
(IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));
```

```
        if (context.InputParameters.Contains("Target") && context.InputParameters["Target"] is Entity)
        {
            Entity entity = (Entity)context.InputParameters["Target"];

            if (entity.LogicalName == "contact")
            {
                if (entity.Attributes.Contains("emailaddress1"))
                {
                    string email = entity["emailaddress1"].ToString();
                    if (!IsValidEmail(email))
                    {
                        throw new InvalidPluginExecutionException("Invalid email address.");
                    }
                }

                if (entity.Attributes.Contains("telephone1"))
                {
                    string phone = entity["telephone1"].ToString();
                    string formattedPhone = FormatPhoneNumber(phone);
                    entity["telephone1"] = formattedPhone;
                }
            }
        }

        private bool IsValidEmail(string email)
        {
            try
            {
                var addr = new System.Net.Mail.MailAddress(email);
                return addr.Address == email;
            }
            catch
            {
                return false;
            }
        }

        private string FormatPhoneNumber(string phone)
        {
            // Format phone number (e.g., (123) 456-7890)
            return Regex.Replace(phone, @"(\d{3})(\d{3})(\d{4})", "($1) $2-$3");
        }
    }
}
```

Example: Custom Business Logic Implementation

Scenario: Implement custom business logic that triggers when an opportunity's status is set to "Won". For example, update a related entity with the opportunity details.

Solution: Create a plugin that runs in the Post-Operation stage of the Update event for the Opportunity entity. This plugin will update the related entity with the opportunity details.

Code Example:

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

namespace MyFirstPlugin
{
  public class OpportunityWonPlugin : IPlugin
  {
    public void Execute(IServiceProvider serviceProvider)
    {
      ITracingService tracingService =
(ITracingService)serviceProvider.GetService(typeof(ITracingService));
      IPluginExecutionContext context =
(IPluginExecutionContext)serviceProvider.GetService(typeof(IPluginExecutionContext));
      IOrganizationServiceFactory serviceFactory =
(IOrganizationServiceFactory)serviceProvider.GetService(typeof(IOrganizationServiceFactory));
      IOrganizationService service = serviceFactory.CreateOrganizationService(context.UserId);

      if (context.InputParameters.Contains("Target") && context.InputParameters["Target"] is Entity)
      {
        Entity entity = (Entity)context.InputParameters["Target"];

        if (entity.LogicalName == "opportunity" && entity.Contains("statuscode"))
        {
          OptionSetValue status = (OptionSetValue)entity["statuscode"];
          if (status.Value == 3) // Assuming 3 is the value for "Won"
          {
            // Retrieve related records and update
            EntityReference customerReference = (EntityReference)entity["customerid"];
            Entity relatedEntity = new Entity("related_entity", customerReference.Id);
            relatedEntity["opportunity_details"] = entity["name"];
            service.Update(relatedEntity);
          }
        }
      }
    }
  }
}
```

Chapter 7: Testing and Deploying Plugins

Unit Testing Plugins

Unit testing plugins is crucial for ensuring that they function correctly before deployment. Here's how you can set up unit tests for your plugins:

1. **Use a Mocking Framework:**

    - Utilize frameworks like Moq or FakeItEasy to create mock services and simulate plugin execution.
2. **Create Unit Test Projects:**
    - Add a new Unit Test project to your Visual Studio solution and reference your plugin assembly.
3. **Write Test Cases:**
    - Develop test cases to validate different aspects of the plugin logic, including edge cases and error handling.

Code Example:

```
using Microsoft.Xrm.Sdk;
using Moq;
using Xunit;

namespace MyFirstPlugin.Tests
{
  public class AutoNumberPluginTests
  {
    [Fact]
    public void Execute_ShouldGenerateAutoNumber()
    {
      // Arrange
      var context = new Mock<IPluginExecutionContext>();
      var tracingService = new Mock<ITracingService>();
      var serviceFactory = new Mock<IOrganizationServiceFactory>();
      var organizationService = new Mock<IOrganizationService>();
      var plugin = new AutoNumberPlugin();

      context.Setup(c => c.InputParameters).Returns(new ParameterCollection());
      context.Setup(c => c.InputParameters.Contains(It.IsAny<string>())).Returns(true);
      context.Setup(c => c.InputParameters[It.IsAny<string>()]).Returns(new
Entity("your_custom_entity"));

      // Act
      plugin.Execute(new ServiceProviderMock(tracingService.Object, context.Object,
serviceFactory.Object, organizationService.Object));

      // Assert
      // Verify that the plugin logic executed correctly
    }
  }
}
```

Deploying Plugins to Different Environments

Deploying plugins to different environments requires careful planning to ensure that plugins function as expected in each environment.

1. **Export and Import Solution:**

    - Package the plugin assembly in a Dynamics 365 solution and export it. Import the solution into the target environment.
2. **Use Deployment Tools:**
    - Utilize tools like Azure DevOps or PowerShell scripts to automate the deployment process.
3. **Test in a Sandbox Environment:**
    - Deploy the plugin to a sandbox environment for testing before moving it to production.

Continuous Integration and Continuous Deployment (CI/CD) for Plugins

Implementing CI/CD pipelines for plugins helps automate the build, test, and deployment processes.

1. **Set Up CI/CD Pipelines:**
    - Use Azure DevOps or GitHub Actions to create pipelines that build and test your plugin code.
2. **Automate Deployments:**
    - Automate the deployment of plugins using scripts and deployment tools.
3. **Monitor and Maintain:**
    - Monitor the CI/CD pipelines and address any issues that arise during the build and deployment processes.

Chapter 8: Troubleshooting and Debugging Plugins

Common Issues and Solutions

Here are some common issues encountered with plugins and their solutions:

1. **Plugin Not Executing:**
    - Verify that the plugin is registered correctly and associated with the correct event.
    - Check for any configuration issues in the Plugin Registration Tool.
2. **Performance Issues:**
    - Optimize plugin code to reduce execution time.
    - Use asynchronous plugins for long-running operations.
3. **Unhandled Exceptions:**
    - Implement proper error handling and logging to diagnose and resolve exceptions.

Using the Plugin Registration Tool

The Plugin Registration Tool is essential for managing plugin registrations and debugging. It provides functionalities to:

1. **Register New Plugins:**
    - Register plugin assemblies and configure their steps, including event and stage settings.
2. **Update and Unregister Plugins:**
    - Modify or remove existing plugin registrations as needed.
3. **Trace and Debug:**

- Enable tracing for plugins to capture detailed execution logs.

Debugging Strategies

Effective debugging involves:

1. **Using Trace Logs:**
   - Implement ITracingService to log execution details and trace issues.
2. **Attaching Debugger:**
   - Attach Visual Studio to the Dynamics 365 process to debug plugins in real-time.
3. **Profiling and Performance Analysis:**
   - Use profiling tools to analyze plugin performance and identify bottlenecks.

Chapter 9: Best Practices and Recommendations

General Best Practices

1. **Design for Scalability:**
   - Ensure plugins are designed to handle large volumes of data and high transaction rates.
2. **Keep Plugins Lightweight:**
   - Avoid placing heavy business logic in plugins; use asynchronous plugins for long-running tasks.
3. **Document and Maintain:**
   - Document plugin functionality, configurations, and dependencies. Regularly review and update plugins as needed.

Recommendations for Plugin Development

1. **Follow Coding Standards:**
   - Adhere to coding standards and best practices to ensure maintainability and readability.
2. **Implement Security Measures:**
   - Use secure coding practices to protect against vulnerabilities and ensure compliance with security policies.
3. **Test Thoroughly:**
   - Perform extensive testing in different environments to validate plugin behavior and performance.
4. **Monitor and Optimize:**
   - Continuously monitor plugin performance and optimize code based on feedback and performance metrics.