Retail Pro® 9
CustomPluginClasses

Retail Pro International, LLC
400 Plaza Dr., Suite 200
Folsom, CA 95630 USA

USA 1-800-738-2457
International +1-916-605-7200
www.retailpro.com

About this Guide

This document explains the CustomPluginClasses used in the development of custom plugin programming for Retail Pro® 9.

If you believe the information presented here is incomplete or inaccurate, we encourage you to contact us at emanuals@retailpro.com.

The software described herein is furnished under a license agreement.

Retail Pro International, LLC
400 Plaza Dr., Suite 200
Folsom, CA 95630 USA

USA 1-800-738-2457
International +1-916-605-7200
www.retailpro.com

Document Revision History

Date	Description
07/24/2008	Original document released.
8/10/2009	Misc. minor changes

CustomPluginClasses

The CustomPluginClasses are a set of classes that implement the Plugin.tlb interfaces so that plugin developers do not have to do it themselves.

Although it is not a requirement that a Retail Pro 9 plugin implement these classes, by inheriting from one of the custom classes, your plugin receives these benefits:

- Simplifies the development of RetailPro Plugins – resulting in faster development time and fewer coding errors.
- You only need to implement (override) the particular methods relevant to your plugin and its problem domain. (ex: “HandleEvent” and “Initialize” for TCustomSidebuttonPlugin)
- Facilitates logging. Logging has been added to the TCustomAbstractPlugin class. All CustomPluginClasses descend from TCustomAbstractPlugin. To enable logging, set the “logLevel” property to something greater than logNone and in each method in which logging is desired, add a call to the overloaded method “Log”. The log is written to the local workstation\Log path. Also, debug logging has been implemented in each parent class method. By setting the logLevel to logDEBUG and running your new plugin (assuming, it compiles, and is in the “Plugins” directory along with it’s Manifest file, etc.) a minimally implemented plugin will log all calls to that plugin so you can see what is being sent to the plugin and in what order.giving you an idea of what data and what order each method is sending to the plugin.
- CustomPluginClasses is supported by RetailPro. Any changes to the API will be reflected in CustomPluginClasses. All you would need to do is save the CustomPluginClasses unit into your shared directory (over writing the existing version) and then next time you compile your plugin the new API feature(s) will be included.
- CustomPluginClasses is supported in Delphi and .NET. The .NET implementation is written in C#. The beauty of .NET is that other .NET languages can inherit from assemblies developed in other .NET languages. So VisualBasic.net can utilize the CustomPluginClasses.dll assembly as well and there is no need to create a version of CustomPluginClasses in VisualBasic.

Delphi implementation:

uPluginDiscover.pas

Place “uPluginDiscover” in a shared library path. There is no need to make any changes to this unit. (see uPluginGlobals)

uPluginGlobals.pas

Use “c_PluginModuleVersion” to set your plugin version, you can change the default messages of “c_ClassFactoryExceptionMsg” and “c_ClassFactoryExceptionCls” to format Class Factory error message to your liking.

The array “c_ClassFactoryArray” - used by the Discover unit, but declared here so you never have to modify the discover unit first add the name of the constant created by the TLB wizard for the CoClass you added to your project TLB, with a @ before the constant name then add the name of the class in which you implemented that CoClass add them in pairs for each CoClass you declared in your project TLB. NOTE: Don't forget to increment the upper array size. The lower array boundary MUST be left at ZERO!

If when creating the tlb file for your project you use the naming conventions provided here for the Discover class (Plugin_Discover), you will never have to change the discover line of the array (first line) as the TLB constant will be called CLASS_Plugin_Discover.

.NET Plugins – C# Example

CSInvoiceSideButton

Some key points in getting your plugin up and running:

In References:

1) Add:

   a) CustomPluginClasses
   b) MSXML
   c) RetailPro.Plugins

In Using clause:

2) Add

   a) RetailPro.CustomPluginClasses;
   b) System.Runtime.InteropServices;

In Properties:

3) Application Tab:

   OutPut type = Class Library
   Click "Assembly Information" and check "Make assembly COM-Visible"

4) Build Tab:

   In the Output section, check "Register for COM interop".

5) Signing Tab:

   Check "Sign the assembly"

   In the combobox titled "Choose a strong name key file:"

   Select <New...> (Add a name and password. !!!

   NOTE: Write down the name/password combination and keep it in a safe place!!!)

Plugin Manifest file:

6) In C# the "ServerName" that you will place in the manifest file should be the same as the Namespace where the Server resides. In this case it should look like this "CSInvoiceSideButton.Discover". This can be confusing when you have a "Project name", "Assembly name", "Default name". If you put the wrong server name in the manifest file, Retail Pro will complain. To be sure what RetailPro is looking for, you can search the Registry for "YourServerName". When you find the "YourServerName.DiscoveryClassName", that is the name to go into the Manifest file. Also, Retail Pro does not support "AutoReg" flag with .Net com servers, you have to register them manually or create a script. This is note pertains to step #4 above, having the IDE register your plugin for you. Please

see the Microsoft help files for guidance if you are going to register it yourself with the Gacutil and RegAsm utilities.

It is very important to decorate your plugin classes with the following Attributes:

[ClassInterface(ClassInterfaceType.None)]

[Guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx")]

The GUID MUST be generated for YOUR plugin! Do not copy these Guids as they will conflict with your next Plugin you create. Each plugin must have is own unique GUID. To generate a guid for your plugin, choose the "Tools" > "Create GUID" menu, GUID format #4, copy and paste in two places: In your static constant class as above and in the GUID attribute as below. Make sure to do it for each and every plugin class you implement.

Debugging a C# Plugin:

To debug a .Net plugin while Retail Pro is running, follow these steps:

On the "My Project"->"Debug" tab, click the "Start Action" radio button labeled "Start external program" and enter the path of your Retail Pro exe ("C:\RetailPro9\RPRO9.exe").

In "Start Options" you can put in any Retail Pro command line parameters you want and you can leave the working directory empty.

In "Enable Debuggers" check "Enable unmanaged code debugging" Add any break points you want in your code, press (F5) to start debugging.

You can also follow these steps to Attach to a process: Start Retail Pro. Once it is running, in the VS IDE, navigate to "Debug">"Attach to Process...""> menu and select these options:

- Transport: "Default" Attach to: "
- Managed code" Available Processes: "
- RPRO9.exe" Check -> "
- Show Process in all sessions"

Now add the Sidebutton to the Invoice Menu. If you set a breakpoint in the HandleEvent method, then click the "C# SideButton" you just added in Retail Pro, your code will stop at the breakpoint previously set.

Initialize method:

Make sure to call the base.Initialize(), there are some important initializations that happen there.

```
public override void Initialize()
{
    base.Initialize();
    // Added additional initialization code here...
 }
```

Visual Basic plugin example:

VBInvoiceSideButton

Note on VB Namespaces:

Do not create a namespace here, set your Namespace in "MyProject" -> "Application" tab "Root Namespace". While it is perfectly legal to create "Local" Namespace here, your COM server will be Registered in Windows as “RootNamespace.LocalNamespace.Discover" RetailPro will complain because it is looking for the format of "ServerName.DiscoverClassName"

This note pertains to step #4 below, having the IDE register your plugin for you. Please see the Microsoft help files for guidance if you are going to register it yourself with the Gacutil and RegAsm utilities.

Some key points in getting your plugin up and running:

In References:

1) Add:

   a) CustomPluginClasses
   b) MSXML
   c) RetailPro.Plugins

In Imports clause:

2) Add

   a) RetailPro.CustomPluginClasses;
   b) System.Runtime.InteropServices;

In My Project:

3) Application Tab:

   OutPut type = Class Library
   Click "Assembly Information" and check "Make assembly COM-Visible"

4) Build Tab:

   In the Output section, check "Register for COM interop".

5) Signing Tab:

Check "Sign the assembly", in the combobox titled "Choose a strong name key file:" select <New...> (Add a name and password. **!!!NOTE: Make sure to right it down!!!**)

In Plugin Manifest file:

6) ServerName:

In VB, the "ServerName" that you will place in the manifest file should be the same as the "Root Namespace" Applications Tab. In this case it should look like this "VBInvoiceSideButton.Discover". This can be confusing when you have a "Project name", "Assembly name", "Local Namespace". If you put the wrong server name in the manifest file, Retail Pro will complain. To be sure what RetailPro is looking for, you can search the Registry for "YourServerName". When you find the "YourServerName.DiscoveryClassName", that is the name to go into the Manifest file. Also, Retail Pro does not support "AutoReg" flag with .Net com servers, you have to register them manually or create a script.

It is very important to decorate your plugin classes with the following Attributes:

```
<GuidAttribute(Discover.CLASS_DiscoverPlugin)>
```

The GUID MUST be generated for YOUR plugin! Do not copy these GUIDs as they will conflict with your next Plugin you create. Each plugin must have is own unique GUID. To generate a GUID for your plugin, choose the "Tools" > "Create GUID" menu, GUID format #4, copy and paste in two places: In your Public constant and in the GUID attribute as below. Make sure to do it for each and every plugin class you implement.

Debugging a VB Plugin:

To debug a .net plugin while Retail Pro is running, follow these steps:

On the "My Project"->"Debug" tab, click the "Start Action" radio button labeled "Start external program" and enter the path of your Retail Pro exe ("C:\RetailPro9\RPRO9.exe").

In "Start Options" you can put in any Retail Pro common line parameters you want and you can leave the working directory empty.

In "Enable Debuggers" check "Enable unmanaged code debugging". Add any break points you want in your code, press (F5) to start debugging.

Initialize method:

Make sure to call the base.Initialize(), there are some important initializations that occur.

```
Public overrides Sub Initialize()
        MyBase.Initialize();
        // Added additional initialization code here…
End Sub
```