Spring 08

VMware Infrastructure 3.5 Plugin and Extension Programming Guide

Schley Andrew Kutz <akutz@lostcreations.com>

Revision 1.3 – 2008/03/11

Table of Contents

Disclaimer

This document is not in any way, shape, form, or measure sponsored, endorsed, or its content supported by VMware. While the succeeding pages may give the impression that this paper was written in cooperation with VMware, this work is the result of hours of using Lutz's Reflector to peer into VMware's intermediate language (IL), Lutz's Resourcer to figure out where icons come from (it's not the icon stork), ProccessMon, FileMon, and RegMon to take a look at things happening in real time, and finally the Microsoft structured query language (SQL) manager to explore the new VI 3.5 database schema. In summary, although the knowledge from these explorations resulted in an idea of the VI plugin architecture and working plugin, do not consider it to be the final word on anything. We will simply have to wait for VMware to provide finality to this whole plugin hoopla.

In short, all the information contained in this document may be entirely and completely wrong. Read it at your own risk. If you find yourself stuck in an infinite time loop once you finish, remember two things: 1) ice sculptures impress the heck out of the ladies and 2) you are not god. You may be **a** god, but not **the** god. That honor is left to Mr. Morgan Freeman.

Introduction

If you are reading this, then you have shown an interest in creating client plugins and server extensions for VMware Infrastructure 3 (VI3). VI3 is already a very capable virtualization solution, and the ability to extend VI3 beyond its original intent is what makes the plugin architecture so powerful. From incorporating graphical storage VMotion (SVMotion) functionality in the client to implementing the first application-level high availability (HA) system, the VI3 plugin architecture allows third-party developers to customize VI3 into exactly the virtualization infrastructure that they need.

You may be wondering, if VMware had intended for developers to write plugins and extensions for VI3, then why have they not releases official documentation for that purpose? The answer is quite complicated, but the short version is that the plugin architecture, in its current state, is not something that VMware feels comfortable sharing with the world. That is not to say that it is not a great architecture. In fact the engineers at VMware should be commended creating a great set of brand new application programming interfaces (APIs) for third-party VMware developers to use. When VMware does make the plugin architecture publically available, there will be a lot of happy people.

If VMware is not ready to release the architecture, then why write this document? Well, the answer is simple really. The VI 2.5 client is a powerful administration tool, but it lacks functionality that takes advantage of some of the VirtualCenter 2.5 and ESX 3.5 features. It would be painful to see external tools being created to leverage this server functionality when the VI client already offers such a wealth of common APIs. So, while VMware created the plugin architecture to fill a need, this document proposes to fill one as well – provide a useful resource for third-party developers who wish to create VI3 client plugins and server extensions.

Getting Started

This section details the software and knowledge that one must possess to begin creating VI client plugins and server extensions.

Styles

The following styles are used in this document.

```
Command Line Interface (CLI)
```

```
Code
```

```
CodeComment
```

`CodeSignature`

`FilePath`

Note

Key Concepts

The following is handy information to have:

- VI client plugins must be developed with .NET 2.0 because they are loaded into the same application domain as the VI client, and application domains are restricted to a single .NET version.
- VI server daemons can be written in whatever language you wish.
- While it may be tempting to simply open your own port for a VI server daemon, remember that the server administrator may not let you.Additionally, the remoting protocol you use should be supported on any OS. So stick with SOAP. HTTP is universal. Except where it's not.

Software

This is a list of software that was found to be quite necessary when discovering how to create plugins and extensions, and then doing so.

- **Microsoft Visual Studio 2008** – We are Linux and OS X geeks by natures, but this hands-down the most superb, integrated development environment (IDE) ever created.
- **Lutz's Reflector** – This is the tool that makes reverse-engineering IL a snap.
- **Lutz's Resourcer** – Pulls those icons out of compiled resources files like nobody's business.
- **ProcessMonitor** – Whooooo are you, who who, who who. We really want to know and now we do, thanks to ProcessMonitor. No process' activity shall escape our watchful gaze!
- **FileMon** – See what files are being accessed in order to figure out how what files need to be opened, altered, and saved. Did that sound dirty to anyone else?
- **RegMon** – Just like FileMon, but for the Windows registry.

Terminology

So far this document has used terms like *client plugin* and *server extension*. These are just two terms that this document uses when referring to VI plugin and extension development. The following is a complete list:

- **Client Plugin** – A client plugin is installed on computers running the VI 2.5 client. These plugins extend the functionality of the VI 2.5 client. These

plugins may come in the form of an installer or simply a compressed container, such as a zip file.

- **Server Extension** – A server extension is registered on the VC server using the VI software development kit (SDK) method,RegisterExtension. Registering a server extension can provide a local server for remote plugins to contact (such as a state server). A server extension can also register a well-known client plugin link such that connecting VI clients will see that a new client plugin is available under "Available Plugins" and have the opportunity to download it.

- **Client-Only Server Extension**– A server extension that exists for the sole purpose of providing an easy way for users to download a VI client plugin.

- **Extension key** – This is the unique key that identifies an extension. VMware has adopted the Java naming convention for uniqueness and thus the key should resemble **com.lostcreations.svmotion**.

Namespaces

Creating plugins and extensions requires knowing about more than just the SDK assembly. Not every interface, class, method, property, or field will be mentioned, and in fact probably 99% of them won't (we are easing into this white paper). The first revision of this document will hopefully include enough explanations of the classes and ilk that you need to know in order to, with confidence, build your own client plugin and server extension.

Additionally, the best way to truly understand the following libraries is to use the object browser in VisualStudio or Lutz's Reflector to take a look inside for yourselves. In this initial revision of the paper, we are just trying to point you in the right direction.

VMware.VIClient.Plugins

This is the assembly that contains the interface a class must implement to be considered a client plugin.The VI 2.5 client uses reflection to peer into assemblies looking for classes that implement the VMware.VIClients.Plugins.Plugininterface (shame on you VMware for not following the standard .NET convention of prefixing interfaces with the capital letter "I") and load those classes as client plugins.

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\Update Manager\VIPlugins.dll`

Interfaces

Plugin

This is the interface that a class must implement to be discovered as a plugin and loaded into the VI client.

Methods

- `void Load(VIApp viApp)` – This method is when the plugin is manually loaded or when the VI client is launched. This method is where a plugin's initialization should occur, such as creating context menu items, views, and menu strip icons.

- `void Unload()` – This method is invoked when the plugin is manually unloaded or when the VI client is closed. Any resources that were opened by the plugin, such as database connections, should be closed in this method.

VIApp

This class provides a means of retrieving a minimal amount of information about the VI client. One of the more important roles of this class is providing the means to retrieve the currently logged in user session of the VI client so that the plugin can interact with the VC server or ESX server(s).

Properties

- `string ServiceUrl`– This is the uniform resource locator (URL) associated with the currently logged on session.

- `System.Net.Cookie SessionCookie`– This is the authenticated cookie of the current session.

- `Menus`– An array that returns references to the VI client's main menus, context menus, and toolbar depending on which one of the TypeStrings' enumerations is used.

- `Views`– An array that returns references to the VI client's views (the tabs on the right-hand side of the client) depending on which one of the TypeStrings' enumerations is used.

Menu

This interface implements VMware.VIClient.Plugins.Extension and is used to represent a menu object.

Extension

This interface provides a very important property named Context.

Properties

- **Context** – This property allows you to access a list of managed object references that represent all of the items selected at the time the object was activated. For example, if you right-click on a VM while three others are selected then the Context property will contain a list of four managed object references.

Classes

TypeStrings

This class includes child classes that are effectively enumerations of type System.String. These enumerations are used as the indices for the following arrays:

- VMware.VIClient.Plugins.VIApp.Menus
- VMware.VIClient.Plugins.VIApp.Views

VimApi

This namespace should look familiar to all VI developers. The only difference is that VMware's implementation has some class name differences. For example, VimService is now InternalVimService. That is the biggest change; otherwise you will feel right at home.

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\2.5\VimSoapService.25.dll`
- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\2.5\VimSoapService.25.XmlSerializers.dll`

Vmomi

While it may be possible to build a client plugin entirely with VIPlugins.dll, one would be remiss not to include a reference to VirtualInfrastructure.25.dll. This assembly provides the important namespaces Vmomi and Vmomi.Support. Vmomi either stands for *VMware Object Management Interface* or *VMware Managed Object Management Interface*. Whatever it stands for, the Vmomi namespace provides a strongly typed class for every ManagedObject type. So instead of invoking the RelocateVM method from the VimService class, the method is directly attached to a VirtualMachine class. Nifty, huh?

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\Update Manager\VirtualInfrastructure.25.dll`

Classes

ServiceInstanceContent

For VI SDK developers the ServiceInstanceContent class is the Vmomi representation of the VimApi.ServiceContent class.

VmomiSupport

This namespace contains support classes for the Vmomi classes. One class in particular stands out.

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\Update Manager\VirtualInfrastructure.25.dll`

Classes

Service

For VI SDK developers, the Service class is an odd combination of the ServiceContent and VimService classes.

CAPICOM

Now we're kickin' it old school. Old like back when you didn't throw paper or scissors, because all you had was rock. CAPICOM is of course Microsoft's component object model (COM) library of cryptographic APIs, hence CAPICOM. Why must you still use this library? We have not yet figured out a way to turn off SSL checking from the VI SDK, and although the Vmomi.Certificate class has a property that tells it to ignore SSL errors, we have not figured how to log onto VI with just the Vmomi classes (if you know, please tell us!). Thus, we need CAPICOM to programmatically add and remove certificates to the system's local certificate store. You'll see this acted out later on when discussing extension registration.

- **Default location**: `C:\Program Files\Common Files\Microsoft Shared\CAPICOM\CapiCom.dll`

For more information on CAPICOM please see Microsoft's CAPICOM reference. For an example of how to use CAPICOM please see this document's section,Creating a Server Extension.

VMware.CustomControls

This namespace contains many custom user interface elements used by the VI 2.5 client. One such class is *ViewEx*.

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\Update Manager\VMware.CustomControls.25.dll`

VpxClientCommon

This namespace also contains many VMware user interface classes. This is where you will find *PropViewBase*.

- **Default location**: `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\Update Manager\vpxClientCommon.25.dll`

VI3.5 SDK 2.5 Extension Management

The VI SDK 2.5 includes new functionality that enables the enumeration, addition, and deletion of server extensions. Examples of these methods can be found in this document's section,Creating a Server Extension. For more information on VI3.5 SDK 2.5 extension management, please see the online VI3.5 SDK 2.5 reference.

Client Plugin Architecture

Although it may not appear so by the sheer number of individual parts that make up both, the client plugin architecture is actually far more complex than the server extension architecture. This is because a server extension's daemon can run as a process entirely independent of VC whereas a client plugin is loaded into the same application domain as the VI client itself. This of course also means that if your client plugin throws a fatal exception, without proper error handling your plugin could very well bring the entire VI client to a James Dean'ing halt.

File System Layout

Client plugins are installed in the VI client plugins' path. This is by default `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins`.

Discovery Process

How does the VI client discover what plugins are available? The discovery process searches both the VC server and the local computer for available plugins.The VI client's plugin manager invokes a method at `VpxClient.PluginManager.PluginManagerImpl.DiscoverClientPlugins()`. This method enumerates the contents of the plugins' folder, searching for assemblies that implement the VMware.VIClient.Plugins.Plugin interface.

The `VpxClient` namespace mentioned above belongs to the assembly VIClient.dll that has a default location of `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\2.5\VIClient.dll`.

This is why it is possible to simply take a compiled .NET class library assembly, stick it in a folder, and place it in the plugins directory. The VI client **will** find it. Simplicity at its finest. No special registration necessary. In fact, when you download an

available plugin from the VI client plugin manager the installer simply places the folder the new plugin was extracted to into the VI plugin directory, and the VI client will recognize the new plugin the next time the plugin manager's discovery method is invoked.

Tomcat Architecture

Before we can discuss the server extension architecture it would behoove us to first review how Tomcat is configured on the VC 2.5 server. While it may seem as if the SDK and various plugin daemons are all being served through a single pair of ports (80-HTTP and 443-HTTPS), the fact of the matter is that those two ports are simply dumb proxies. The real servers are running on internal ports, receiving proxied communication from the front-end instance of Tomcat.

The main Tomcat configuration file can be found at (default location) `C:\Program Files\VMware\Infrastructure\VirtualCenter Server\tomcat\conf\server.xml`.

Check out the file (default location) `C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter\proxy.xml`. A quick look inside the file reveals that this is the configuration file that the primary Tomcat instance uses in order to configure its proxies. For example, the following node configures the SDK proxy:

```
<e id="3">
   <_type>vim.ProxyService.LocalServiceSpec</_type>
   <accessMode>httpsWithRedirect</accessMode>
   <port>8085</port>
   <serverNamespace>/sdk</serverNamespace>
</e>
```

It is evident from the extensible markup language (XML) above that a proxy connection is being configured from port 443 (HTTPS) to a local daemon listening on port 8085.

Why does any of this matter? Well, because it is the responsibility of your server extension to provide a daemon that will receive incoming communication from the client – if such a thing is appropriate. As mentioned, there are some server extensions that serve only to exist as a mechanism by which VI administrators can download a client plugin directly from their VI clients.For these types of server extensions it does not make sense to provide their own back-end proxy. These server extensions simply need a place to put their client setup files. For this purpose we recommend creating a directory called `plugins` under `C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter\docRoot\`. This directory is the document root for the primary

Tomcat server, that is to say that when a user visits http://YOUR_VC_SERVER/file.foo, file.foo is being served from this directory (for the purposes of this explanation we will not get into file generation techniques).

Additionally, there is nothing to say that you must use the Tomcat proxy. A remoting technology such as .NET Remoting is perfectly acceptable, but remember that whatever means of communication is chosen, it should support the widest range of clients as possible.

Database Schema

The VC 2.5 installation adds some new tables to the VC database.

- **VPX_EXT** – This table holds basic information about an extension, including its unique key (unique by Java naming conventions, e.g. com.lostcreations.svmotion).
- **VPX_EXT_CLIENT** – This table holds information about possible clients associated with a server extension.
- **VPX_EXT_PRIVS** – This table holds custom privileges associated with a server extension.
- **VPX_EXT_SERVER** – This table holds information about a server extension's daemon end-point and protocol information.
- **VPX_EXT_SERVER_EMAIL** – This table holds administrative e-mail addresses associated with a server extension.
- **VPX_EXT_TYPE_IDS**– This table holds custom type identifiers associated with a server extension.

To find out more about these database tables please use the Microsoft SQL manager to login to your SQL server and browse around. Alternatively you can also read about the ExtensionManager. This document includes an example of registering a server extension later on that will help clear up any confusion that may have been born of this section.

Server Extension Architecture

The server extension architecture is largely defined as four items: database information, resource files for the server extension, a possible client binary available for download, and an end-point on the VC server for said client to communicate with.

File System Layout

A server extension exists on the file system in a few places.

Resource Files

A server extension's resource files are by default located at `C:\Program Files\VMware\Infrastructure\VirtualCenter Server\extensions\KEY` where `KEY` may equal something like **com.lostcreations.svmotion**. The actual resource files exist two levels down in `locales\LOCALE` where `LOCALE` equals something like **en**. The root directory is actually a really great place to store any additional utilities that the server extension may need later on.

Client Setup File

The client setup file that is associated with this server extension (if there is one) should be placed somewhere in the path of the VC server's web server. We place our client setup files in the sub-folder called `plugins` in the document root path of the primary Tomcat server: `C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter\docRoot\`.

We used Microsoft Visual Studio 2008's Setup and Deployment project to create MSI installers for both our SVMotion client and server extension. The problem is that the Tomcat server that ships with VC does not support the MSI MIME type. We tried adding the MSI MIME type as an octet/stream (just as the EXE MIME type is) to the Tomcat server's web.xml file, but even after restarting Tomcat the web server refused to serve the MSI file. In the end we chose to register our server extension with it telling VI clients to go to our own server lostcreations.com to retrieve the client setup file. If anyone figures out how to get Tomcat to server MSI files we greatly appreciate you letting us know.

Discovery Process

The VI client searches for server extensions by invoking the method `ArrayList() VpxClient.PluginManager.PluginManagerImpl.DiscoveServerPlugins()`. This method uses the VimApi ExtensionManager to get a list of the available server extensions. If the server extension references a client plugin that does not exist in the client plugin directory then it will prompt the user to download the plugin.

Creating a Client Plugin

Creating a client plugin is a process with six distinct steps.

1. Define assembly properties.
2. Create a class that implements VMware.VIClient.Plugins.Plugin.
3. Write code to retrieve the VI client's session.
4. Create the plugin's menu items, views, and other graphical objects that users interact with into the VI client.

5. Install the plugin.
6. Activate the plugin.

All the below code examples are from the SVMotion client plugin. If you are having trouble reading the following code, please visit the SVMotion website to see the code presented in wide-screen format. Because of the width restrictions of a portable document format (PDF) file the code comments have been removed or reducedfor the below examples and not all fields and properties are present either. Again, please visit the SVMotion website to see an up-to-date and fully commented version of the source code for examples.

Additionally, the code seen here is a member of a Microsoft Visual Studio 2008 .NET 2.0 Windows Class Libraryproject with references to the usual system assemblies as well as VIPlugins.dll, VirtualInfrastructure.25.dll, and VimSoapServices.25.dll, and all the dependencies that these three assemblies require.

Define assembly properties

You need to define some assembly properties on your assembly.

```
usingSystem.Reflection;
usingSystem.Runtime.InteropServices;
usingSystem.Runtime.CompilerServices;

[assembly: AssemblyTitle( "SVMotion" )]
[assembly: AssemblyDescription( "Adds a graphical SVMotion
   option to the VI 2.5 client." )]
[assembly: AssemblyCompany( "l o s t c r e a t i o n s" )]
[assembly: AssemblyProduct( "SVMotion" )]
[assembly: AssemblyVersion( "0.0.3.0" )]
[assembly: AssemblyFileVersion( "0.0.3.0" )]
```

Full Source Code Link – AssemblyInfo.cs

Implementing the VMware.VIClient.Plugins.Plugin

The first step to creating a client plugin is creating a new class file (we will be using C#). Create the class file and then implement the VMware.VIClient.Plugins.Plugin interface. An example is seen below:

```
using System;
usingSystem.Text;
usingSystem.Collections.Generic;

namespace SVMotion
{
   public class SVMotionPlugin
      :VMware.VIClient.Plugins.Plugin
   {
      public void Load(
```

```
         VMware.VIClient.Plugins.VIAppviApp )
      {
         // Do nothing
      }

      public void Unload()
      {
         // Do nothing
      }
   }
}
```

Full Source Code Link – SVMotionPlugin.cs

The above class will successfully compile (given the right assembly references) and be recognized as a valid VI client plugin. However, as you may have noticed, not much will actually happen beyond a plugin showing up in the list of installed plugins. We need to add some functionality to the class.

Retrieving the Current Session

It is the first task of the Login method to retrieve the currently logged on session in use by the VI client. Retrieving the currently logged in session results in obtaining a reference to the session's ServiceInstanceContent object.

```
public void Load( VMware.VIClient.Plugins.VIAppviApp )
{
   // Get a SOAP service end point for the VimService
   // class.
   VirtualInfrastructure.Soap.SoapServiceInfossi = new
   VirtualInfrastructure.Soap.SoapServiceInfo( typeof(
      VimApi.InternalVimService ), null );

   // Get a coppy of the currently logged in VI client's
   // session by passing the URL of the currently logged
   // in session and the cookie with authenticated
   // credentials.
   Service = VmomiSupport.ServiceManager.GetService( ssi,
      new Uri( m_viapp.ServiceUrl.Trim() ),
      VIApplication.SessionCookie );

   // Get the server/session unique ID (the suid) of this
   // session'sServiceInstance managed object.
   stringsuid = VirtualInfrastructure.ManagedObject.ToSuid(
      VIApplication.ServiceInstance.WsdlTypeName,
      VIApplication.ServiceInstance.Id );

   // Get a reference to this session's
   // ServiceInstanceContent object by using the
```

```
   // Service object's ManagedObjectIdentifiedBy method and
   // the service content's unique server/session ID.
   ServiceInstanceContent =
      VirtualInfrastructure.ManagedObject.ToSuid(
      Service.ManagedObjectIdentifiedBy[ suid ] as
      Vmomi.ServiceInstance ).RetrieveContent();

   // This next line is very important. All plugins must
   // make sure that they create a property collector
   // or else they will fail if they are the first
   // plugin to load.
   Service.UpdatesManager.CreatePropertyCollector(
      ServiceInstanceContent.PropertyCollector,
      System.Threading.SynchronizationContext.Current );
}
```

Full Source Code Link – SVMotionPlugin.cs

Interestingly enough there is a far easier way to get a reference to the current session's ServiceInstanceContent, although it is surely not supported by VMware (more so than this even!). The assembly "VIClient.dll" that is by default installed at `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\2.5\VIClient.dll` has a class named "VpxClient.Common.Globals.Vmomi". This class has static property accessors for the current session's Service, ServiceInstance, and ServiceInstanceContent. Although it is technically possible to simply have your plugin reference VIClient.dll and grab direct references to the current VI client's session information, it is clearly out of line with VMware's proposed (from what I can tell) plugin architecture. Stick to using the code block below and your plugins are more likely to be forward compatible with future versions of the VI client.

Creating Menu Items, Etc.

Once a reference to the session is obtained, it is time to add some menu items, views, and perhaps icons to the main toolbar. All these things are accomplished using the Menus and Views collection in conjunction with specific TypeStrings. The following is an example of adding context-menu items for specific inventory objects in the VI client, such as virtual machines (VMs), host servers, cluster compute resources, and such.

```
using VMware.VIClient.Plugins;

// Create an array of the inventory items you want to
// attach the context menu item to.
String[] iitems = new String[]
{
   TypeStrings.Inventory.VirtualMachine,
```

```
   TypeStrings.Inventory.VirtualMachineFolder,
   TypeStrings.Inventory.ResourcePool,
   TypeStrings.Inventory.HostSystem,
   TypeStrings.Inventory.ComputeResourceFolder,
   TypeStrings.Inventory.Datacenter,
   TypeStrings.Inventory.Cluster
};

// Create a new context menu item for each inventory
// object, attaching event handlers for its Created
// and Activated events.
Array.ForEach<String>( iitems, delegate( String item )
{
   VIApplication.Menus[ item ].Add( "-" );
   VMware.VIClient.Plugins.Menu menu =
      VIApplication.Menus[ item ].Add(
         "Migrate storage..." );

   menu.MenuItemCreated += new
      VMware.VIClient.Plugins.PluginEvent(
         svmotion_Menu_OnCreate );

   menu.Activated += new
      VMware.VIClient.Plugins.PluginEvent(
         svmotion_Menu_OnActivate );
}
```

Full Source Code Link – SVMotionPlugin.cs

We learned an interesting little tidbit in our playing around. It turns out that it is only possible to create menu items **in**Load method. Why is this a bummer you ask? Well, consider the following scenario. Imagine you want to create sub-menu items for your menu item depending on current state of the inventory item or based on some external data (such as a database full of third-party applications). You cannot do it! How annoying. The solution is to create your own ContextMenuStrip object and display it when a menu item you created at load-time is clicked. For more information on this problem and a solution please see the Invoke plugin.

Now it is time to define the callback methods that we attached to the MenuItemCreated and Activated events. We will be referencing the menu's Context property, so in case you forgot, go read about how it is used to get a list of managed object references.

```
private void svmotion_Menu_OnCreate( object sender )
{
   VMware.VIClient.Plugins.Menu m = sender as
      VMware.VIClient.Plugins.Menu;
   m.MenuItem.Click += new EventHandler(
```

```
      svmotion_MenuItem_OnClick );
}

private void svmotion_Menu_OnActivate( object sender )
{
   VMware.VIClient.Plugins.Menu m = sender as
      VMware.VIClient.Plugins.Menu;

   // Attach the associated list of managed object
   // references to the menu item's Tag property. The Tag
   // property exists for the purpose of attaching
   // associated data so that it may be accessed later on.
   m.MenuItem.Tag = m.Context.Object;
}

private void svmotion_MenuItem_OnClick(
   object sender, EventArgs e )
{
   System.Windows.Forms.MenuItem m = sender as
      System.Windows.Forms.MenuItem;

   SVMotionFormsvmf = new SVMotionForm(
      this, ( m.Tag as
         List<VMware.VIClient.Plugins.ManagedObjectReference>
   ) );

   svmf.ShowDialog();
}
```

Full Source Code Link – SVMotionPlugin.cs

If you add submenu items, it appears that there is a bug such that their Activated event is not actually fired. We performed ample testing and could not get it to work.

As you can see, when a user clicks on the new context menu item a form is created and shown to the user as a dialog box. The decision to show the form as a dialog box is simply to ensure that the user must interact with the form. Describing the form is out of scope for this document, but you can browse the form's source online.

Installation

Installing a client plugin occurs in three ways:

1. Install from installation program.
2. Copy a zip file to the correct directory.
3. Install from the VI client.

However a plugin is installed, its files will be located in (by default) `C:\Program Files\VMware\Infrastructure\Virtual Infrastructure Client\Plugins\`.

Installation Program

Simply follow the instructions that the installation program provides. The end result will be the installation program placing the plugin's file in the VI plugin directory.

Install from a Zip File

Unzip the contents of the file. If the contents are loose, that is they unzip to the current directory, create a directory for them and place them inside of it. Then copy the new directory to the VI plugin directory.

Install from the VI Client

If a server extension has been registered with an available client plugin download then the plugin will appear in the "Available Plugins" screen of the plugin manager in the VI client. Simply click the "Download and Install" button to have the plugin installed for you.

Activation

Activating a plugin simply requires that you click on the "Installed Plugins" tab of the VI client plugin manager and check the box next to the plugin. That's all there is to it.

Creating a Server Extension

Creating a server extension is important for a few reasons:

- It makes it easy for VI administrators to install plugins directly from the VI client.
- It allows you to centrally keep track of what plugins you are providing VI administrators.
- It allows you to define custom types for both you server extension and client plugin.

Registering the Extension

Registering a server extension is a little tricky. It may seem straightforward, but in order to accomplish it in a manner that works for all systems the process can get a little <jamesBrown>fonky</jamesBrown>. A few things to consider:

- You have to consider that SSL is a requirement for their VC installation.
- You have to consider that you may want to install files on the VC server.

The fact is that the latter helps the former. Because a requirement we place on registering extensions is that you should register them (although this is not

required, but we recommend it) from the VC server itself, we can actually manage the first consideration with some amount of ease.

Registering an extension consists of three distinct steps:

1. Logging on to the VC server.

2. Building the extension object.

3. Registering the extension.

Logging on to the Server

Remember, to log onto a VC server with the VimApi we have to trust the SSL certificate that is attached to the server. The problem with this is, what if we don't? We **could** prepare the environment manually so that the registration worked, but where is the fun in that? One of the benefits to running the registration code on the VC server is that we have access to the VC public certificate. It is located at (default location) `C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter\SSL\rui.crt`. However, we also run into another interesting issue; not only must the process registering the extension trust the certificate that the VC server presents, it must also access port 443 with the host name that the certificate was issued to, which is not guaranteed to be the name of the server. Luckily, there is a way to manage all of this.

The following code is a little long, but hopefully apparent in its purpose. Here is what happens in summary:

1. The VC certificate is loaded into an object via CAPICOM.

2. The common name of the certificate is read and then inserted at the end of the server's host file that is located at (by default) `C:\windows\system32\drivers\etc\hosts`. For those of you who are not familiar with a hosts file, the hosts file is checked for domain name service (DNS) resolution prior to hitting a DNS server. This way we can define DNS registrations locally.

3. Log onto the server.

Although it won't be shown in the example, the certificate is removed and the hosts file is restored once the registration is complete. Remember, the full source of the following examples is online.

```
// This method adds the VC public certificate to the local
// computer's certificate store making it possible to
// communicate with VC over SSL.
private void AddVMwareCertificate()
{
```

```
   // Get the path to VC public certificate.
   stringrui = string.Format( @"{0}\Application
      Data\VMware\VMware VirtualCenter\SSL\rui.crt",
         Environment.GetEnvironmentVariable(
            "ALLUSERSPROFILE" ) );

   // Use CAPICOM to open the certificate.
   CAPICOM.CertificateClass cert = new
      CAPICOM.CertificateClass();
   cert.Load( rui, null,
      CAPICOM.CAPICOM_KEY_STORAGE_FLAG.
      CAPICOM_KEY_STORAGE_DEFAULT,
      CAPICOM.CAPICOM_KEY_LOCATION.CAPICOM_CURRENT_USER_KEY
   );

   // Parse the name of the certificate was issued to. This
   // is the name that will be added to the hosts file.
   m_str_server_name =
      Regex.Match( cert.SubjectName,
         "CN=(?<cn>[^,]*?),",
         RegexOptions.IgnoreCase ).Groups[ "cn" ].Value );

   // Open the local computer's certificate store.
   CAPICOM.StoreClass store = new CAPICOM.StoreClass();
   store.Open(
      CAPICOM.CAPICOM_STORE_LOCATION.
      CAPICOM_CURRENT_USER_STORE,
      CAPICOM.Constants.CAPICOM_ROOT_STORE,
      CAPICOM.CAPICOM_STORE_OPEN_MODE.
      CAPICOM_STORE_OPEN_READ_WRITE );

   // Add the VC certificate and close the store.
   store.Add( cert );
   store.Close();
}

// This method inserts a new hosts file entry that
// points the CN on the VC certificate to localhost.
// This allows us to connect successfully to the localhost
// with SSL.
private void AddVMwareToHosts()
{
   stringhosts_file_path = string.Format(
      @"{0}\system32\drivers\etc\hosts",
      Environment.GetEnvironmentVariable( "SystemRoot" ) );

   m_str_original_hosts_file = System.IO.File.ReadAllText(
```

```
      hosts_file_path );

   System.IO.File.WriteAllText( hosts_file_path,
      string.Format( "{0}{1}127.0.0.1\t{2}",
         m_str_original_hosts_file, Environment.NewLine,
         m_str_server ) );
}

// These values are used when creating the server extension
// object. Ideally these values should be coming from a
// alocalized resource file, but for the purposes of this
// example we'll stick with my native tongue, gibberish.
private string m_str_server = GetServerNameFromCert();
private string m_str_original_hosts_file = "";
private string m_str_key = "com.lostcreations.svmotion";
private string m_str_version = "0.3.0";
private string m_str_client_url =
   "http://www.lostcreations.com/downloads/vmware/viplugins/
SVMotionClientSetup.msi";
private string m_str_name = "SVMotion";
private string m_str_description = "Adds a graphical
   SVMotion option to the VI 2.5 client.";
private string m_str_company =
   "l o s t c r e a ta i o n s";

// Add the local VC certificate to the list of trusted
// certificates so SSL connections are accepted.
AddVMwareCertificate();

// Create a local hosts file entry so the SSL
// connection to the local VC server succeeds.
AddVMwareToHosts();

// Logon to the server.
VimApi.ManagedObjectReferencevim_svc_ref = new
   VimApi.ManagedObjectReference();
vim_svc_ref.type = "ServiceInstance";
vim_svc_ref.Value = "ServiceInstance";
VimApi.InternalVimServicevim_svc = new
VimApi.InternalVimService();

// Important. For some reason the internal library times
// very fast if you do not set the timeout to around 5
// seconds.
vim_svc.Timeout = 5000;
vim_svc.Url = "https://" + m_str_server + "/sdk";
vim_svc.CookieContainer = new System.Net.CookieContainer();
```

```
VimApi.ServiceContent vim_svc_content =
   vim_svc.RetrieveServiceContent( vim_svc_ref );

// Substitute an administrative username and password.
vim_svc.Login( vim_svc_content.sessionManager,
   USERNAME, PASSWORD, null );

// There is no need to register it twice, so see if it
// already exists. Alternatively, if you are registering
// a new version you could choose to unregister the
// existing version here.
if ( vim_svc.FindExtension(
   vim_svc_content.extensionManager, m_str_key ) != null )
{
   return;
}

// This description object is used later by different
// extension objects.
VimApi.Description d = new VimApi.Description()
{
   label = m_str_name,
   summary = m_str_description,
};

// This object describes the client plugin associated with
// this sever extension.
VimApi.ExtensionClientInfoeci = new
VimApi.ExtensionClientInfo()
{
   company = m_str_company,
   description = d,
   type = "win32",
   version = m_str_version,
   url = m_str_client_url,
};

// Build the values that will be used to create the
// extensionrsource information object.
VimApi.KeyValue kv1 = new VimApi.KeyValue();
kv1.key = m_str_key + ".client.win32.label";
kv1.value = m_str_description;
VimApi.KeyValue kv2 = new VimApi.KeyValue();
kv2.key = m_str_key + ".client.win32.summary";
kv2.value = m_str_name;
VimApi.KeyValue kv3 = new VimApi.KeyValue();
kv3.key = m_str_key + ".label";
```

```
kv3.value = m_str_name;
VimApi.KeyValue kv4 = new VimApi.KeyValue();
kv4.key = m_str_key + ".summary";
kv4.value = m_str_name;
VimApi.KeyValue kv5 = new VimApi.KeyValue();
kv5.key = m_str_key + ".server.SOAP.label";
kv5.value = m_str_name;
VimApi.KeyValue kv6 = new VimApi.KeyValue();
kv6.key = m_str_key + ".server.SOAP.summary";
kv6.value = m_str_name;

// Create the extension resource information. Ours is
// English, but it could be any language here. This
// resource information is then saved on the server
// to the directoryC:\Program Files\VMware\
// Infrastructure\VirtualCenter Server\extensions\KEY\
// locales\LOCALE\.
VimApi.ExtensionResourceInfoeri = new
   VimApi.ExtensionResourceInfo()
{
   locale = "en",
   module = "extension",
   data = new VimApi.KeyValue[]
      { kv1, kv2, kv3, kv4, kv5, kv6 },
};

// Finally, create the extension object. The only
// property that you should not recognize (that we
// have not done anything with yet) is the
// lastHeartbeatTime property. To tell you the truth,
// we have not figured out what this property is for.
//  The database requires the field, so we guess it
// is used for something. Maybe to prop the coffee table
// up. We don't know.
VimApi.Extension ext = new VimApi.Extension()
{
   version = m_str_version,
   key = m_str_key,
   lastHeartbeatTime = DateTime.Now,
   description = d,
   client = new VimApi.ExtensionClientInfo[] { eci },
   resourceList = new VimApi.ExtensionResourceInfo[]
      { eri },
};

// Register the extension.
```

```
vim_svc.RegisterExtension(
   vim_svc_content.extensionManager, ext );
```

Full Source Code Link – VCCredentialsForm.cs

And that's it. The server extension has been registered and ready to use!

Creating Windows Installers

```
throw new NotImplementedException( "Nothing to see here,
move along." );
```

Client Plugin

```
throw new NotImplementedException( "Nothing to see here,
move along." );
```

Server Extension

The example will be coming later, but we wanted to go ahead and share one of the problems we ran into when creating a server extension installer. The purpose of the server extension installer is two-fold: 1) it should put all the right files in all the right places, but 2) it needs to register the server extension.

There is a problem.

Registering the server extension requires the use of a Custom Action. Custom Actions can be separate binaries, but they can also be loaded into the installer as a class inside of a Windows Class Assembly that implements the Installer interface. Except your custom action must link to the VMware assemblies that are required to register an extension. Except that those assemblies are not on the computer until your installer puts them there.

Get it? It is a chicken and egg kind of thing. The custom action will fail to run because its dependencies are not present when the installer is launched, and it is at that point that it loads dependencies.

There are two solutions:

1. Strongly type VMware's assemblies so that they can be loaded into the global assembly cache (GAC) and then link them. That way the installer will not feel compelled to bundle them as dependencies and when it is run on the remote machine it will look for them in the GAC as well. The problem with this is that these assemblies must actually **be** in the GAC on the remote computer. You cannot be sure that they will be, which is why you should rely on the second solution.

2. Instead of writing your custom action as a library, make it an executable. This way the custom application has its own application domain and when it is created the appropriate dependencies will be in place. You can see an example of this online at RegisterSVMotionServerExtension.

For more information on why side-by-side assemblies and custom actions are bad, and to prove we're not making stuff up, please see:

- Why a custom action may not run
- Generally though, EXE custom actions are bad.
- Missing dependencies are the exception

Client Plugin Examples

This section will list various client plugin examples.

Creating Views

The tabs on the right-hand side of the VI 2.5 client that appear when an inventory item is selected are called views. It is possible to create your own view, and this example will explain how. All the code from this example is from the Console plugin. Once again, some of the code comments, properties, and fields may be reduced or removed to make the code more readable inside this document format. To see the code and its comments in their original format please visit the Console plugin's website.

There are four basic steps to creating views:

1. Create a new VMware.VICient.Plugins.View object.
2. Attach a user control to the new view's Control property.
3. Create a context changed event handler for the view.
4. Display the view's data.

Creating a New View Object

Creating a new view object is not that different from creating a menu item object.

```
/*
 * Add a new view to all inventory objects of type
 * HostSystem. This results in a new tab appearing on the
 *right-hand side of the VI client called "Console"
 * (this is not static, it is just what we are naming
 * the view as you can see from the line below).
 */
VMware.VIClient.Plugins.View view = VIApplication.Views[
VMware.VIClient.Plugins.TypeStrings.Inventory.HostSystem
].Add( "Console" );
```

Full Source Link – ThePlugin.cs

Creating the User Control

As seen above, creating a new view object is simple. Every view object has a property named Control. This property is a reference to a System.Windows.Forms.UserControl class and is what is displayed in the lower-right-hand corner of the VI 2.5 client. While you may want to stop reading, go create your own custom UserControl class and get to business, you would be missing out on more of our sage-like advice if you do not continue reading. So please, for the sake of rosemary and thyme, keep reading.

```
/*
 * Set the view's Control property to a new
 * System.Windows.Forms.UserControl(). The Control
 * property is the object that is show in the
 * bottom-right-hand side of the VI client. While it may
 * be tempting to set this property to a new instance of
 * your custom UserControl, *don't*! Having a parent
 * control is helpful as we are about to see.
 */
view.Control = new System.Windows.Forms.UserControl();

/*
 * Setting the parent control's Dock property to Fill
 * and the AutoSize to false and AutoScroll to true
 * results in a nice effect. If for any reason the
 * children of this control are out-of-bounds then scroll
 * bars will automatically appear allowing the user to
 * bring your custom controls back into view.
 */
view.Control.Dock = System.Windows.Forms.DockStyle.Fill;
view.Control.AutoSize = false;
view.Control.AutoScroll = true;
```

Full Source Link – ThePlugin.cs

Creating the Context Changed Event Handler

When a user clicks on another item in the inventory, the context of this view changes. This is just a really fancy way of saying that the managed object reference that it was using has changed. For example, clicking on host B after already looking at host A would result in a context change. Monitoring these context changes are important as they allow us to take actions as a result of a new inventory item being selected. This is very important as we will see in a minute.

```
// Add an event handler so that whenever a new host is
// selected, that is when the context of this view changes,
//we will know about it.
view.ContextChanged += new
   VMware.VIClient.Plugins.PluginEvent(
      view_ContextChanged );
```

Full Source Link – ThePlugin.cs

Displaying the View's Data

Displaying the view's data is the very reason we wanted to create a view in the first place! It is when you come to this stage that you realize there are actually two very different types of views:

- **Data driven**: These views use a single user control to display their data, having only to refresh their data upon a context change. For example, take a view that simply displayed the name of the selected host server. The name would most likely be displayed in a System.Windows.Forms.Label class. Well, when the context changes, the view's user control reference does not have to point to a **new** user control that contains a **new** Label with the name of the new host server. No, the value of the Label would simply be altered.

- **User control driven**: These views use a separate user control for each inventory item in order to display their data. For example, take the Console plugin – it uses a separate ConsolePanel custom user control for each host server that it connects to because it would be tedious and time consuming to attempt to switch out the backing SSH connection behind the actually display. So instead, a separate user control is used for each host server.

This example is using a user control driven view. We will point out what is necessary to implement these in the following code block:

```
// This is a list that maintains references to each
// host-specific user control for the Console view.
private Dictionary<string, ConsoleUserControl.ConsolePanel>
   m_consoles = new Dictionary
      <string, ConsoleUserControl.ConsolePanel>();

.
.
.

/// <summary>
///     This is the callback that is invoked when the view's
///    context has changed. Now what that means is quite
///    simply that the user has clicked on another host and
///    the view's managed object reference has been
///    updated.
/// </summary>
/// <param name="sender">
///     The view object.
/// </param>
voidview_ContextChanged( object sender )
{
```

```
// Get the view object.
VMware.VIClient.Plugins.View view =
   sender as VMware.VIClient.Plugins.View;

/*
 * Get the managed object reference for the updated
 * context. In this case it will be the currently
 * selected host system. I probably do not have
 * toinitmate that the context would be a virtual
 * machine if this was a virtual machine view. And so on.
 */
VMware.VIClient.Plugins.ManagedObjectReferencemoref =
   view.Context.Object as
   VMware.VIClient.Plugins.ManagedObjectReference;

// Get the host system from the suid.
stringsuid = VirtualInfrastructure.ManagedObject.ToSuid(
   moref.WsdlTypeName, moref.Id );
Vmomi.HostSystem host =
   Service.ManagedObjectIdentifiedBy[ suid ] as
   Vmomi.HostSystem;

// Define the variable that will hold a reference to the
// actualConsole user control.
ConsoleUserControl.ConsolePanel cp = null;

// If the console has already been defined, that is, it
// isin the list of consoles that is global to this
// class, thenuse that one.
if ( m_consoles.ContainsKey( suid ) )
   cp = m_consoles[ suid ];

// Otherwise create a new console and add it to the list.
else
{
   cp = new ConsoleUserControl.ConsolePanel();
   m_consoles.Add( suid, cp );
}

// Set the console's hostname / IP address to that
// of the the currently selected host.
cp.HostNameOrIPAddress = host.GetName();

/*
 * Hide the view's parent control while we clear
 * the child controls (our custom controls). Hiding
 * the parent control while we do this prevents
```

```
    * flickering. Once the child controls are cleared
    * add the custom control that is relevant to this
    * host.
    */
   view.Control.Visible = false;
   view.Control.Controls.Clear();
   view.Control.Controls.Add( cp );
   view.Control.Visible = true;
}
```

Full Source Link – ThePlugin.cs

Creating Global Views

A global view is visually represented as the icons on the primary tool strip in the VI 2.5 client. The *Inventory* and *Administration* buttons or icons, along with the others, display a global view when pressed. Global views are not that much different than regular views, except for perhaps visual appearances. Take for instance the following scenario:

You want to add a new view for VMs so that it will list the last three people to power cycle the VM. You could create a view called "PowerLog" and then simply add the appropriate controls and populate them with the necessary data. No special steps are required to make the view look like it belongs, to make it visually consistent with other, VMware views. But what if you were asked to create a global view that displayed the last 10 people that have logged into the server? Although the process can be identical, doing so results in a visual that rips the user away from the VI 2.5 client because of its stark contrast to the rest of the application.

The question then becomes, "How do we create a global view that is visually consistent with the rest of the VI 2.5 client? The answer: we steal, er, borrow, from VMware. VMware has packaged significantly large number of custom controls in the two assemblies *vpxClientCommon.25.dll* and *VMware.CustomControls.25.dll*. We actually do **not** want to dig into these libraries, as they are not part of the plugin architecture. However, to best serve the interest of our users, we must.

These two assemblies contain two classes that we will use to build out global view: *VpxClientCommon.PropViewBase* and *VMware.CustomControls.ViewEx*.

PropViewBase is a container that you are familiar with, even if you do not know what it is. This class is responsible for displaying the tab views on the right-hand side of the VI client or on the bottom half of the client in a global view. This is the class to which you add said tabs. ViewEx is the class that VMware itself uses to display tabbed content.

The following code examples come from the VI plugin *Invoke*, of which the entire source code is available online. If you are having trouble reading the code, please try to read it online in a wider viewing format.

```
// Create the classes that perimeter container
// (PropViewBase) the content container (ViewEx)
VMware.CustomControls.ViewEx view = new
   VMware.CustomControls.ViewEx();
VpxClientCommon.PropViewBase pview = new
   VpxClientCommon.PropViewBase();

// Add a new tab called "Settings" and configure the tab.
// You'll notice that third argument to this function is
// a Type class. The function is asking us what type
// of object will this tab display. In our case it is
// a ViewEx object.
pview.AddTabButton( "Settings",
   VMware.CustomControls.ViewID.GettingStarted,
typeof( VMware.CustomControls.ViewEx ) );

pview.SetButtonsTextOptions();
pview.SetDockPadding();
pview.DoAutoArrangeButtons();

// Associate a view with this tab. I just picked a ViewID
// of "Getting Started". It is not really important for our
// purpose.
pview.AddView( view,
   VMware.CustomControls.ViewID.GettingStarted,
   VMware.CustomControls.ViewType.Normal);

// Add a global view called "Invoke" to the icon bar at the
// top. This special global view collection is accessed
// through the special type string of
// "VMware.VIClient.Plugins.
//   TypeStrings.Inventory.Global"
VMware.VIClient.Plugins.Viewgview =
   VIApplication.Views[
     VMware.VIClient.Plugins.TypeStrings.Inventory.Global
   ].Add( "Invoke" );

// Set the global view's user control to that of the tabbed
// control (the PropViewBase) that we created earlier.
// Did you forget already?
gview.Control = pview;


// Attach a new user control to the ViewEx's Control
// property. At this point laying out a global view is just
// like laying out a regular view (see the last section).
// Or, at least it is until we figure out something
```

```
// that contradicts that statement : )
view.Control = new UserControl();
```

Full Source Link – ThePlugin.cs