Taking Advantage of ADSI

Active Directory Service Interfaces (ADSI), is a COM-based set of interfaces that allow you to interact and manipulate directory service interfaces. OK, now in English that means it's a cool way of writing scripts and code that can do things like adding users, changing passwords, creating network groups, controlling IIS programmatically, and starting and stopping services. In this article I'm going to be covering the basic ADSI syntax, navigating the ADSI section of Microsoft's MSDN site to get the most bang-for-your-buck, and give you some example code that you can use in your own applications.

ADSI lends itself to automating a number of system administration tasks. When used in conjunction with the Windows Scripting Host (WSH, see John Petersen's article "The Windows Scripting Host" in the Spring 2000 issue of CoDe) it can give you nearly full automation control of your Windows system. Although perfect for system administrators, developers can leverage this technology to help do things such as automating software installations and controlling IIS to help deploy web-based applications. You can take advantage of NT's security system in a web app. by adding users via ADSI and letting NT do the work of validating them.

Background

ADSI provides a common set of interfaces to various network providers, so that you can have and use a consistent interface to these providers. Hence the "SI" part of ADSI. By default, the following ADSI providers are installed:

? WinNT: – Windows NT Networking provider
? LDAP: - Lightweight Directory Access Protocol (Exchange, Win2000)
? NWCOMPAT: - Novell NetWare 3.1
? NDS: - Novell Directory Services

? IIS : - Internet Information Server - installed during IIS installation

ADSI is available for Windows NT by installing the Option Pack (see online resources for more info), and is installed automatically in NT above SP3, and Windows 2000. Microsoft also has client components available for Windows 95 and 98 on the MSDN website. The examples show here assume Windows NT or 2000.

This article is going to be primarily centered around using the WinNT: providers, although most of the concepts and code will be nearly identical for any of the other providers.

Every ADSI object exposes at least 6 properties: GUID – a unique identifier, Parent – the parent object, AdsPath (a.k.a. the providers listed above), class – the class it belongs to, and Schema – the schema that defines the object, and name – our object's name.

Here's some code to see how this works. You will need to replace the name "OMNIBOOK_DEV", which is the name of my notebook with your system name. It seems like

we should be able to use LOCALHOST as the system name, but it's expecting our real system name, not an IP address (or a name that will be resolved into an IP address, as is the case with LOCALHOST). To find the system name under Windows 2000, go into the Control Panel and click on the System icon. Then click on the Network Identification tab, the system name is listed as "Full computer name:". Under NT 4.0 get to the Control Panel, double click on Network. You'll find the machine name and domain under the Identification tab.

```
oComputer = GetObject("WinNT://OMNIBOOK_DEV,computer")
? oComputer.guid
? oComputer.Parent
? oComputer.AdsPath
? oComputer.Class
? oComputer.Schema
```

On my system, this returns the following:

```
GUID:     {DA438DC0-1E71-11CF-B1F3-02608C9E7553}
Parent:   WinNT://WORKGROUP
AdsPath:  WinNT://WORKGROUP/OMNIBOOK_DEV
Class:    Computer
Schema:   WinNT://WORKGROUP/Schema/Computer
```

You can see from the above that Parent returns the parent of the current AdsPath, and the Class is the same name we used in the GetObject statement. So where did that WORKGROUP thing come from? The WinNT provider gives us access to various properties of the NT security model. In this security model users can be members of a Domain or Workgroups and servers can be Primary Domain Controllers (PDC), Backup Domain Controllers (BDC), or standard servers. We're getting a bit beyond the scope of this article, but suffice to say that in this case, "WORKGROUP" is the name of the workgroup my machine belongs to. It could just as easily been named something else. Because of these distinctions, sometimes it's necessary to specify which workgroup or domain the system/user belongs to as part of the path. NOTE: you can use the workgroup name in place of the domain name if, like me, you're part of a workgroup instead of a Domain. Depending on how your machine is configured, you will get different results. Knowing all this, hopefully the paths make a little more sense.

Finally, there's the GUID (Globally Unique Identifier). This is just the standard GUID Windows assigns to every COM object. You would use this to "bind", or create a reference to, the ADSI container object using the GUID. This is faster than letting the system do the lookup for you, but the number isn't very user-friendly, is it? Plus it makes distribution a problem, and has a few other limitations.

Now let's take a closer look at the syntax and the provider's "namespace". A namespace is a bounded area, or container, where names can be resolved into information or an object reference. In the sample, the code following the provider reference is called its namespace because it can be resolved into an object reference. In the GetObject call we referenced the ADSI provider, passed it a machine name, *the* prefix "computer". In this case "computer" is the name of one of the

objects available in this namespace. The actual syntax and available objects depends on the namespace you are using. Namespaces can be used as a way to implement polymorphism. Each ADSI provider has it's own set of namespaces that are specific to the provider, but can use the same sets of names as other providers in order to simplify their use. Information about the available objects can either be found via documentation such as MSDN, or through the Schema object that is part of every namespace. More on that in a bit.

Helping you to help yourself

The easiest way to take advantage of ADSI is by using the wealth of information available on MSDN (see the online resources listing). Granted, it's not always easy making sense of the information provided or figuring out why something doesn't quite work the way you expected. One area that can cause some confusion is wading through the number of terms used when talking about ADSI. We've already covered some of them, but here's a few more for good measure.

We've already seen how to bind to an ADSI provider and looked at the syntax of the GetObject. This syntax is called the binding string. Each provider will have its own unique way of calling it, just to keep you on your toes.

Each object available through ADSI implements a number of interfaces. Interfaces are collections of the Properties/Events/Methods (PEMs) we're all used to manipulating. A little bit of trivia here, all COM objects must, at a minimum, implement the IUnknown interface. By checking the interfaces that are available for each of the objects in your namespace, and the types of PEMs these interfaces support, you can determine the types of things you can actually do with each object. If you understand a normal class hierarchy where subclasses inherit the PEMs of their parents, you can understand interfaces. They're not *quite* the same thing, but very similar in that they are a way of extending the functionality of objects. Understanding this goes a long way towards being able to deciphering the MSDN docs and really making the most of ADSI.

Examples

Enough with the theory, let's see how we can put ADSI to work to actually do something useful. One thing to keep in mind when running these examples is that you must have the rights to actually perform these tasks. For example, things will not work right if you're logged in as a Guest. Logging in as the Administrator is probably your best bet while trying out these examples.

Listing all users on a machine:

VFP CODE

```
oComputer = GetObject("WinNT://OMNIBOOK_DEV,computer")

FOR EACH oMember in oComputer
     IF oMember.Class = "User"
```

```
          ? oMember.Class + ":" + oMember.name
     ENDIF
ENDFOR
```

You'll notice that we're using the Class property to filter out only the users. There is actually a Filter property that you could use instead, but it does not work in VFP. You pass it an array that holds the items your interested in, in this case "User". Unfortunately, the Filter property is expecting what is called a Safe Array, which VFP can't create. Since we have access to the class property, it's not too much of a loss. In VB, here's what that code might look like:

VB CODE

```
Dim oComputer

Set oComputer = GetObject("WinNT://OMNIBOOK_DEV,computer")
oComputer.Filter = Array("User")

For Each oMember In oComputer
     Print oMember.Class & ":" & oMember.Name
Next
```

Try the code without any filtering, it should give you some ideas of the other types of things you can manipulate using ADSI.

Take a second to compare this code to the first listing. The syntax changes between VFP and VB are very minor.

I previously mentioned being able to use the Schema object to determine what PEMs the various namespaces support. Grabbing the schema is pretty easy. NOTE: This example works only in VFP 7 – it appears they've enhanced support for Safe Arrays, although I was still unable to pass an array to the Filter property. You can also try this under VB with just a few syntax changes.

VFP CODE (VFP 7 only)

```
oComputer = GetObject("WinNT://OMNIBOOK_DEV,computer")
oSchema = GetObject(oComputer.Schema)

? "Computer Schema object: " + oComputer.Schema

IF oSchema.Container
     ? "This is a container object with the following children:"

     FOR EACH ObjectType IN oSchema.Containment
          ? "Child: " + ObjectType
     ENDFOR
ENDIF
```

```
?
? “This object has the following properties:”

FOR EACH ObjectProperty IN oSchema.MandatoryProperties
     ? ObjectProperty + “ (mandatory)“
ENDFOR

FOR EACH ObjectProperty IN oSchema.OptionalProperties
     ? ObjectProperty + “ (optional)”
ENDFOR
```

After running this, you should see that the computer object contains a number of children objects (such as User) that you can access. It also doesn’t have any mandatory properties and a few optional ones.

Creating a new user:

This one not only creates a new user, but it assigns that user to the Administrator account. One thing to watch out for here, you’ll want to make sure you’ve got a Guest account (and it’s not disabled). Otherwise things will fail.

VFP CODE

```
oDomain = GetObject("WinNT://WORKGROUP/OMNIBOOK_DEV")
oUser = oDomain.Create("user", “testuser”)
oUser.SetPassword(“MyPassword”)
oUser.FullName = “ADSI Test”
oUser.Description = “ADSI Test Account”
oUser.SetInfo()

oGroup = GetObject("WinNT://OMNIBOOK_DEV/Administrators,group")
oGroup.Add(oUser.AdsPath)
```

In this example, I specifically referenced the domain, when I could just have easily directly referenced the system name. Why? So I could make a point, of course. When creating or modifying a user on a network that has both primary domain controllers (PDC) and backup domain controllers (BDC), your application must reference the PDC. Attempting to SetInfo() on the BDC will return an error. Thanks to Randy Pearson for finding this one.

Editing a user:

To edit a user, just pass the account name in your GetObject call. In this example, we’re using the guest account. Pretty simple, isn’t it?

VFP CODE

```
oUser = GetObject("WinNT://OMNIBOOK_DEV/guest")
oUser.FullName = "Guest Account"
oUser.Description = "New description"
oUser.SetInfo()
```

Hey, this doesn't work!

While using ADSI, you're bound to come upon examples that seem to work under VB, or ASP, but fail under VFP. One thing that sometimes seems to help is using the Get() and Put() methods instead of accessing properties directly. How did I know that there are even Get() and Put() methods? Check out the interface that every ADSI object inherits from – IADs and you'll have your answer.

If you just can't get something to work under VFP, try running it under VB. If it works you may have run into a compatibility issue with VFP (such as with Safe Arrays). If it doesn't, at least you have narrowed the list of possibilities. If it did work, you might want to consider writing an ActiveX "wrapper" in VB. See the sidebar for an example.

Another aspect of using ADSI that's easy to overlook is that it runs under the NT security system. That means it's bound by the same security limitations of any normal user account. This should make sense, since if it didn't, it would be a huge security issue. If you think your code looks good but it is still failing to run properly, make sure that you (or your application) actually have the rights to do what you're attempting to do. This is especially a concern when running scripting code through ASP on the web. Check those rights!

Here's another thing that's easy to miss, and it's a biggy – the provider names are case-sensitive. WinNT: is not the same as winnt: ! Case-sensitive coding errors can be a real bear to track down, so keep this one in mind.

Summary

ADSI gives developers and system administrators powerful control over many different directory services. We've only had the opportunity to cover a few of the ways it can but used, but I hope that you can see how ADSI's interface simplifies tasks that used to require a C++ or Win32 interface guru. Please do yourself a favor and make sure you check out the Online Resources - There are a lot more code examples available. If you come up with any cool uses of this technology, drop me a note. I'd love to hear your ideas.

Online Resources:

MSDN:
http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/adsi/adsistartpage_7wrp.htm&RLD=407

Win32 Scripting:
http://cwashington.netreach.net/script_repository/faqs.asp?topic=adsifaq

West Wind technologies:
http://www.west-wind.com/presentations/IISadmin.htm

VB-World:
http://www.vb-world.net/activex/

Creating a simple VB wrapper

Here's a simple example of how you might go about creating an ActiveX wrapper in Visual Basic. This example shows one possible way to get around VFP's inability to pass an array to the ADSI filter property. To start, fire up VB. When you're first prompted to create a new project, select ActiveX DLL and click on Open. You will be prompted with a window where you can add the following code:

```
Dim aSafe As Variant

' Set our variant variable to an empty array.
' A variant is an undefined variable. VFP variables
' are considered variant because their type isn't set
' until we assign a value to them.

Public Sub Clear()
   aSafe = Array()
End Sub

' Create a public method called Add. We accept an
' item called vNewItem, passed by value, as a variant variable.

Public Sub Add(By Val vNewItem As Variant)
  ' Get the array size, resize it then save the passed in value
  ' to the array.
  nASize = Ubound(aSafe)
  nASize = nASize + 1
  ReDim Preserve aSafe(nASize)
  aSafe(nASize) = vNewItem
End Sub

' Let/Gets are similar to VFP's assign & access methods.

Public Property Let aArray(ByVal vNewArray As Variant)
  If IsArray(vNewArray) Then
    aSafe = vNewArray
  End If
End Property
```

```
Public Property Get aArray() As Variant
  aArray = aSafe
End Property

‘ Do the real work, we pass in an object reference to the ADSI
‘ object. Then we set its filter property to our local array.

Public Sub SetFilter(ByRef oADSI As Object)
  If IsObject(oADSI) Then
    OADSI.Filter = aSafe
  End If
End Sub

‘ Like VFP’s Init() method. This just resets our array.

Private Sub Class_Initialize()
  Me.Clear
End Sub
```

After typing in the code, select the root node of the treeview in the Project Window on the right hand side of the screen. It should be labeled “Project 1 (Project 1)”. If you take a look at the properties window, you should see a property called “Name” that is set to Project1. Let’s change that to: vbutils. This is the name of the ActiveX control that will be created. Now click on the Class node of the project tree. Set the name property here to SafeArray. Now save this project by clicking on File, then Save Project. Set the class name when prompted to SafeArray, and the Project name to VbUtils.

We’re finally ready to compile and test our control. Click on File, then on Make VbUtils.dll.

Let’s exit out of VB and start up VFP to test our new control. From the command window in VFP, try the following code. Remember to replace the OMNIBOOK_DEV machine name with your own.

```
oComputer = GetObject(“WinNT://OMNIBOOK_DEV,computer”)

* Create an instance of our new ActiveX control
* Notice how we reference the control: Project Name.Class Name
* this is exactly the same way it works in VFP when creating
* COM objects.

oSafe = CreateObject(“VBUtils.SafeArray”)

* Add the item “User” to our filter
oSafe.Add(“User”)

* Set the ADSI filter property
oSafe.SetFilter(oComputer)

* Walk through each member object

FOR EACH oMember IN oComputer
   ?oMember.Class
ENDFOR
```

If everything is correct, you should see a listing of only the “User” classes when you run the above code. Wasn’t that easy?