

Accessing eDirectory from Perl on NetWare

© 2006 by David Bank

by David Bank, CNE
dbank@trianglenug.org

This AppNote is a refresh and update of the February 2001 AppNote entitled “How to Program to NDS eDirectory on NetWare Using Perl” by Guruprasad S.

Contents:

- Introduction
- Review
- Prerequisites
- Perl on NetWare Peculiarities
- About Objects, Attributes and Values
- Practical Code Examples
- Coding Pitfalls
- Conclusion

Introduction

Since its introduction in the 1990s, eDirectory (then called NDS) has become a vital part of many networks, large and small. Starting in NetWare v4.11 (*aka* IntraNetWare), Novell began including the Perl interpreter, a common tool in the Linux/*NIX world, as part of the default NetWare install. The goal of this article is to show how you can combine the flexibility of Perl with the power of eDirectory to develop helpful tools and applications

Review

A series of AppNotes in October and November of 2000, and February of 2001, described how NDS/eDirectory could be accessed from Perl scripts running on NetWare, using an API known as the Universal Component System, or UCS. Since that time, all of the components – NetWare, Perl, eDirectory and the UCX/UCS APIs - have undergone significant changes. Some of the sample code and techniques presented in the 2000/2001 AppNote articles do not work with the modern software combinations.

Prerequisites

- Perl v5.8.x installed on NetWare v6.x or OES-NetWare
- The Universal Component System (UCS) APIs (included by default in NetWare)
- A text editor to write the scripts
- Console access to the NetWare server

Perl on NetWare Peculiarities

The Perl interpreter is an open source software project, and by design is very portable. Perl has been ported to virtually every *NIX/Linux platform in existence, and to OSes as diverse as OS/390 and, of course, NetWare.

Because Perl is developed in and still primarily aimed at the *NIX/Linux environment, it's not always possible to port all of its functionality to other platforms. This is especially true when there are major architectural differences between the environments.

Perl on NetWare has a number of peculiarities that may trip up even an experienced Perl programmer, and these should be kept in mind as one develops Perl code to run on NetWare. A more-complete listing can be found in the Perl on NetWare documentation, specifically

<http://developer.novell.com/ndk/doc/perl/index.html?page=/ndk/doc/perl/pr1584enu/data/bookinfo.html>

However, some items of particular interest to this article include:

It is mandatory to use lexically scoped variables (with help of the **my()** operator) whenever possible for scripts which are executed using **mod_perl**

Scripts that introduces infinite loop cannot be terminated

The Perl debugger *restart* option is not supported

While the first item says it's mandatory only for scripts executed using **mod_perl**, my experience has been that, in the NetWare environment, it's best to scope variables no matter what. The example code presented will be based on this practice.

Another peculiarity of Perl on NetWare, as it specifically relates to the Universal Component APIs, is that if you have a **require** or **use** statement for the UCS API module, but never instantiate an object provided by the module anywhere in your script, the server may **ABEND**.

Paths to server-local files are referenced using the syntax **VOLUME:/PATH/TO/FILE**

eDirectory context references are in the form **nds:\\TREE\\TOP O\\OU**

Including the **use strict;** statement in NetWare-based Perl scripts is problematic, because the UCS APIs seem to perform some redefinitions that the directive doesn't like. The **use warnings;** statement works, however, as do the **-c** and **-w** parameters on the Perl invocation.

Note that since “\” is a special character in Perl, it must be escaped, and so is represented in the code examples as “\\”.

With respect to eDirectory, note that eDirectory error codes are not available to your Perl scripts. Many UCS methods return, at best, only Boolean (“yes/no”, “TRUE/FALSE”, “OK/FAIL”) values, and you will have to use **DSTRACE.NLM** to capture eDirectory error information.

Novell does not publish separate Perl-oriented UCS documentation. Instead, you should use the Novell

Script for NetWare (NSN) UCS documentation. The Objects, Properties and Methods are the same, only the language syntax differs. Specifically, reference the Novell Developer Kit (NDK) **NSN Components (Parts One and Two)**, as well as the **NDK Novell eDirectory Schema Reference**. These are available on the Novell Developer website (<http://developer.novell.com>).

Finally, the Perl community has an impressive library of add-on Perl modules – the best-known repository is CPAN, the Comprehensive Perl Archive Network (<http://www.cpan.org>). However, adding the typical Perl module to NetWare's Perl installation is a non-trivial exercise, involving either establishing a NetWare development environment, or cross-compiling on another platform. Neither choice is for the inexperienced or faint-of-heart. For the vast majority of admins, you're limited to whatever Perl modules are included in the NetWare distribution.

About Objects, Attributes and Values

eDirectory is a quantum leap (arguably several leaps) in technology over Novell's original Bindery technology that was used through NetWare v3.x. As different as eDirectory is, there are some concepts and structures that the two environments share. These become evident when you start to analyze and manipulate individual Objects in eDirectory.

While eDirectory allows the definition of new and unique Objects (where Bindery only understood a small, static set), the notion of an Object is the same in both environments: a collection of groups of data of various types. Users are Objects. Printers are Objects. Groups are Objects. These things are true in eDirectory as much as they were in the Bindery world.

Similarly, it's true of both environments that Objects have Attributes. The Attributes of an Object are defined by its type – that is, a User Object consists of a different set of Attributes than a Group Object. A particular Attribute might appear in many different Object types.

Attributes have a Value – the data that the Attribute contains. Some Attributes are Multi-Valued Attributes (MVAs) and may contain more than one Value – the membership list of a Group is a common example.

In the eDirectory world, the schema defines (among other things) the available Attributes, the Attributes used by the various Objects (also called Layouts), the data type(s) of the Values (known as Syntax), and the Values associated with the various Attributes. Understanding these inter-relationships, and their hierarchical nature, is important to understanding how to access and safely manipulate eDirectory when using a direct tool such as the UCS API.

While this article confines itself to using UCS to access User Objects, that is an artificial limitation. Many other Object types exist and are accessible via the UCS APIs, and the APIs provide many other methods beyond those presented here.

Practical Code Examples

Example 1: Using UCS – The API for Perl

For any Perl script to access any UCS component, it must first include the Perl UCS module. Its also helpful to enable Perl's built-in warning checks:

```
use warnings;
require Perl2UCS;
```

The second statement will load **SYS:PERL\LIB\AUTO\PERL2UCS.NLM**, a NetWare-specific Perl module that provides an interface between Perl and the Universal Component System. Other NLMs that will get loaded as a consequence may include:

```
UCSCORE.NLM
UCS2UCX.NLM
UCXMGR.NLM
```

Our next task is to instantiate an eDirectory object. As noted above, scoping variables is a very good idea, and a UCS-provided object must be instantiated before the script ends, or the server may **ABEND**. So, let's initialize a local variable and then use it for the instantiation:

```
my $eDirObj;

$eDirObj = Perl2UCS->new("UCX:NWDIR") or die "\n ERROR: Unable
to instantiate eDirectory object \n Exiting... \n";

print "\n eDirectory object successfully created";
```

Code Example 2: Logging In to and Out of eDirectory

Without logging in to eDirectory, your script is limited to whatever operations the **Public** Object may perform. By default, **Public** is limited to searching and reading limited parts of the directory (and many savvy eDirectory administrators reduce, or eliminate altogether, access for the **Public** Object).

To get any useful work done, your script must login to eDirectory. This also raises a security issue. For your script to easily login without human intervention, it must have the login credentials. They must either be encoded in the script, or read from somewhere (like a file). Neither of those two options is especially appealing, so consider the consequences of how you design your script.

For the purposes of this article, we'll assume that the script is interactive, and relies on the invoker to supply valid credentials. We'll further assume that the login context is already known (alternatively, the invoker could supply a fully-qualified Object name). This code builds on Example 1:

```
my $Tree = "Tree";
my $Top_O = "Corp";
my $OU = "IT";

my $login_context = "\\$Top_O\\$OU";
```

Changing the eDirectory context for the script is merely a matter of writing a new value into the **FullName** Property of the eDirectory Object, although the context must be one into which the **Public** Object may

change context:

```
print "\n Setting eDirectory context to $login_context...\n";

$eDirObj->{"FullName"} = "nds:\\\\$Tree$login_context";
```

Note that it is possible to force your script to use a specific server in the Tree as its Preferred Server, if you want to ensure that you connect to the server holding a replica, or the Master replica, of the partition with the objects you're targeting:

```
my $ServerIP = "10.0.0.1";
$eDirObj->{"PrefServer"} = "$ServerIP";
```

Again, this is simply a matter of writing to the appropriate Property of the eDirectory Object. While any server in the Tree can execute the code presented, you might set the Preferred Server to avoid excess eDirectory traffic.

We're now ready to solicit credentials from the invoker:

```
my $loginID;
my $password;

print "\n\tEnter your login ID: ";
$loginID = <STDIN>;
print "\tEnter password: ";
$password = <STDIN>;

# Remove the trailing <CR> character from the variables; if
# you don't, they will cause authentication to fail
chop $loginID;
chop $password;
```

At this point, we've set a context for logging in, a preferred server, and have (hopefully) valid credentials for the eDirectory Tree. The easiest way to find out if they are valid is to try to login with the **login()** method; it will return **FALSE** if the login fails, although its not possible to tell why it failed (for example, to distinguish an invalid password from a non-existent user). The call is very similar to the eDirectory Object instantiation:

```
print "\n\tLogging into eDirectory Tree $Tree using context
    $login_context as User ID $loginID...\n";

$eDirObj->login( $loginID, $password ) or die "\n\nLogin
    failed, exiting...\n";

print "\t...login successful!\n";
```

Of course, it's also important to be able to log out. If you allow your script to end without logging out, the connection won't be cleared and resources may not be released. Logging out is very simple:

```
$eDirObj->logout();
```

Once you have successfully logged in, then during future method calls, you should check method return codes as opposed to using the **or die** construct, as you want to logout before killing your script.

Example 3: Writing to a Server-local file

In the event you wish your script to record its activities to a file, which can be of great use with debugging, you can (subsequent to a successful login) write to a file local to the server where the script is running. Its also possible to read from or append to a file (using the usual Perl syntax for these operations), and non-local files can be accessed as well.

For the purposes of this article, we'll limit ourselves to opening a local file for writing progress messages and debugging information, and build on the code from Example 2:

```
my $LogFile = "DATA:/Perl/scripts/logfile";

if ( open ( LOGFILE, ">$LogFile" ) )
    { print "\n\t Logging to $LogFile...\n\n"; }
else
    {
    $eDirObj->logout();
    die "\n Unable to open $LogFile \n";
    }
```

Again, it is important that you not simply use the **or die** construct to exit your script if the file operation fails – remember to logout first. Similarly, if you open a file, then be sure to close it before logging out and exiting the script.

While it is possible for your Perl script to write its log to the **SYS:** Volume, I specifically recommend against doing so in practice, especially during initial development of your scripts. If your script gets stuck in a loop that includes writing to the file, then since infinite-looping scripts cannot be aborted, it is possible for your script to run the **SYS:** Volume out of disk space, which can crash the server and have far-reaching consequences to any eDirectory replicas it hosts. Write your files elsewhere whenever possible. If you must write to **SYS:**, consider applying an appropriate Directory Size Limit, which applies even to Supervisor-equivalent accounts.

Example 4: Changing Context

Once you've authenticated, as in Example 3, your script is free to change context within the Tree, as permitted by the credentials under which it is logged in. Again, changing context is simply a matter of writing the new context to the **FullName** Property of the eDirectory Object:

```
my $work_OU = "Sales";
my $work_context = "\\$Top_O\\$work_OU";

$eDirObj->{"FullName"} = "nds:\\\\$Tree$work_context";
```

There is no return code or error check. You'll only detect an error when you call a context-sensitive method and it fails.

Example 5: Searching the eDirectory Tree

Unless you know the exact object you want, it's useful to be able to search the eDirectory Tree. You can search on a number of criteria, but for this article, we'll generally limit ourselves to User Objects.

The first step in searching is to construct a **Filter** Object. This is done with a simple method call to the eDirectory Object, so let's use the eDirectory Object from Example 3:

```
my $Filter = $eDirObj->{"Filter"};
```

By default, the **Filter** will search in the current context of the eDirectory Object (the value of the **FullName** Property) when the **Filter** Object was instantiated. You may change this by writing a different value into the **SearchContext** Property of the **Filter** Object:

```
my $search_OU = "Engineering";
my $search_context = "\\$Top_O\\$search_OU";

$Filter->{"SearchContext"} = "nds:\\\\$Tree$search_context";
```

Next, we want to define a search **Scope**. Its possible to search your entire eDirectory Tree with a single **Filter**, but let's limit ourselves to just the current OU:

```
$Filter->{"Scope"} = $Filter{"SEARCH_SUBORDINATES"};
```

There are three possible **Scope** values, each with a pre-defined name and a corresponding integer value. The names should be interpreted with respect to the eDirectory Object, not the current context. This is a subtle and potentially confusing distinction:

<u>Scope Name</u>	<u>Integer Value</u>	<u>Note</u>
SEARCH_ENTRY	0	Search the current <u>object</u> only
SEARCH_SUBORDINATES	1	Search the current context, but not any sub-contexts
SEARCH_SUBTREE	2	Search the current context, plus all OUs below it

If you wanted to search the entire Tree, you'd simply set the eDirectory object's context to **[Root]** ("**nds:\\\\\$Tree**"), instantiate the **Filter** object, and set the **Scope** to **SEARCH_SUBTREE**.

Now that you've set a **Scope**, you need to formulate the **Search**. A **Search** is made up of **Expressions**, and can be complex, using multiple criteria with Boolean logic. For the purposes of this article, we'll construct a very simple **Search** that looks for User Objects only:

```
$Filter->AddExpression( $Filter->{"FTOK_EQ"}, "Object Class", "User" );
```

Once you have listed all the **Expressions** in the **Search**, you need to terminate the list of **Expressions**:

```
$Filter->AddExpression( $Filter->{"FTOK_END"} );
```

In preparation to execute the **Search**, we're need to initialize a pointer to the list of the eDirectory Objects we expect to be returned:

```
my $Entries = $eDirObj->Reset();
```

Finally, we conduct the **Search** by applying the **Filter** to the eDirectory Object:

```
$Entries = $eDirObj->Search($Filter);
```

The return value is a pointer to the list of matching Objects, or **NULL** if the **Filter** failed to find any

objects. In the former case, we must reset the list pointer to the start of the list. We can also see how many Objects matched the **Search** criteria:

```
my $ObjCount;

if ( $Entries )
{
    $Entries->Reset();
    $ObjCount = $Entries->{"Count"};
    print "\nSearch successful - ", $ObjCount, " Objects Found\n";
}
else
{
    print LOGFILE "\nERROR: Search resulted in no objects...\n";
    close(LOGFILE);
    $eDirObj->logout();
    die "\nERROR: Search resulted in no objects...\n";
}
```

Again, it is important to detect and handle error conditions gracefully, rather than just using the **or die** construct.

Once you have obtained a list of matching Objects, you may step through them using a **while** loop, controlled using the return value of the **HasMoreElements()** method:

```
my $Entry;
while ( $Entries->HasMoreElements() )
{
    # Get the next Entry
    $Entry = $Entries->Next();

    # DO SOMETHING HERE

    # End of while loop
}
```

The **HasMoreElements()** method will return **FALSE** when the end of the list of Objects is reached. You can use the **Reset()** method to return the pointer back to the start of the list. **\$Entry** is a pointer to the current Object.

Example 6: Finding a Specific Object

As an alternative to a **Search**, if you know the Value of the **Full Name** Attribute of an eDirectory Object, you can use the **Item** method of the **\$Entries** Object to find it. The example code below leverages the code in Example 3, and works in whatever happens to be the current context of the eDirectory Object (so if you want to look elsewhere, first change context by altering the **FullName** Property of the eDirectory Object, as shown in Example 4).

After declaring our local variables for pointers and to hold the **Full Name** of the user we're looking for, we'll set a pointer to the list of Objects – all Objects, not just User Objects – that are in the current context of the eDirectory Object. We'll then use the **Item** method to find our specific target in that list. The result will be **NULL** if no match is found, or else a pointer to the specific Object we're looking for:

```
# Pointer to list of Objects in current context
my $ObjectList;
# Pointer to specific Object
```

```

my $Object;
# Full Name of Object we want to find
my $TargetObject = "Dave Bank";

# Get a pointer to the list of all Objects in the current context
$ObjectList = $eDirObj->{"Entries"};

# Use the Item method to locate a specific Object
$Object = $ObjectList->Item($TargetObject);

if ( $Object )
    { print "\nFound ", $Object->{"Full Name"}, " in context ",
      $eDirObj->{"FullName"}, "\n"; }
else
    { print "\nContext ", $eDirObj->{"FullName"}, " does not contain ",
      $TargetObject, "\n"; }

```

You can also use this code example to locate a specific Object within the list of Objects returned by a **Search**. If we were to assume that the **Search** in Example 5 was successful and **\$Entries** was a pointer to the list of Objects returned, then the **Item** method will work against that list:

```

my $TargetObject = "Dave Bank";

$Entry = $Entries->Item($TargetObject);

if ( $Entry )
    { print "\nFound ", $Entry->{"FullName"}, " in Search results\n"; }
else
    { print "\nSearch results do not contain ", $TargetObject, "\n"; }

```

Note that because the **Search** presented in Example 5 encompassed a sub-tree, we don't know the exact context in which **\$TargetObject** was found. We only know the context of the **Filter** Object when the **Search** was performed.

Example 7: Enumerating Layouts and Attributes

Every Object in eDirectory is associated to a Layout, defined by the eDirectory schema. You can enumerate all of the Layouts in your eDirectory environment by obtaining the list of Layouts from the eDirectory Object. Here, we build on Example 3 by declaring some local variables for accessing the list of Layouts, and displaying some of their Properties:

```

my $LayoutList;
my $LayoutProp;

# Get the list of Layouts
$LayoutList = $eDirObj->{"Layouts"};

# Reset the pointer to the top of the list
$LayoutList->Reset();

print "\neDirectory defines the following Object Layouts\n";
print "\tName\tBased On\tRemovable\n";
# Step through the list and print selected Properties of
# each Layout
while ( $LayoutList->HasMoreElements() )
    {
    $LayoutProp = $LayoutList->Next();
    print "\t", $LayoutProp->{"Name"}, "\t", $LayoutProp->{"BasedOn"};
    if ( $LayoutProp->{"Removable"} )
        { print "\tYes\n"; }
    else

```

```

        { print "\tNo\n"; }
    }

```

Each Layout specifies the Attributes of the associated Objects. The **Layout** and **Fields** methods can be used to enumerate the structure of an Object. This code example assumes that **\$Entry** is already pointing to a User object (as it would in Example 5). We'll declare some local variables, retrieve the Layout for the User Object, then enumerate the Fields (Attributes) in the Layout:

```

my $Layout;
my $Fields;
my $Attribute;

# Find the Layout associated to the Object
$Layout = $Entry->{"Layout"};

# Retrieve the list of Fields in this Layout
$Fields = $Layout->{"Fields"};

# Move the list pointer to the start of the list
$Fields->Reset();

print "\nObject ", $Entry->{"Full Name"}, " has the following Attributes: \n";

# Enumerate the Object's Attributes (Fields)
while ( $Fields->HasMoreElements() )
{
    $Attribute = $Fields->Next();
    print "\t", $Attribute->{"Name"}, "\t";
    if ( $Attribute->{"Optional"} )
        { print "Optional\n"; }
    else
        { print "Required\n"; }
}

```

Finally, you can retrieve the eDirectory Syntax, the description of each Attribute defined in eDirectory. This can tell you if a specific Attribute is a simple data type (e.g. an integer, a BOOLEAN) or a complex data type (e.g. a path, time, an ACL). Additionally, you can find out if an Attribute is single-Valued or multi-Valued, and removable or non-removable. Here, we build on Example 3 and declare some local variables to contain the data we access, obtain the list of Attribute types, then step through the list, displaying the properties of each:

```

my $TypeList;
my $TypeProp;

$TypeList = $eDirObj->{"FieldTypes"};

if ( $TypeList )
{
    $TypeList->Reset();

    print "\neDirectory defines the following Attributes:\n";
    print "\tName\tSyntaxType\tSyntaxName\tRemovable\n";
    while ( $TypeList->HasMoreElements ( ) )
    {
        $TypeProp = $TypeList->Next();
        print "\t", $TypeProp->{"Name"}, "\t",
            $TypeProp->{"SyntaxType"},
            "\t", $TypeProp->{"SyntaxName"}, "\t";
        if ( $TypeProp->{"Removable"} )
            { print "Yes\n"; }
        else
            { print "No\n"; }
    }
}

```

```

    }
else
{ print "\nERROR: Could not get list of Attribute types\n"; }

```

Even for a basic modern eDirectory Tree, the outputs from these three code examples could be quite extensive. For instance, just the User Object Class is defined, by default, with about 200 Attributes (including the ones it inherits). Modern eDirectory easily exceeds a total of a thousand defined Attributes in most cases, and defines over 170 Layouts. Knowing and understanding this information is crucial to being able to leverage eDirectory to its full power as a business advantage.

Example 8: Viewing an Object's Attributes

The Values contained in the Attributes of Objects may be read using the **GetFieldValue** method. To read an Attribute's Value(s), you must know the name of the Attribute (which can be discovered by enumerating the Layout, or looking at the eDirectory Schema documentation). It's important to note that the name of an Attribute in eDirectory does not necessarily correspond to the name of the associated field in a tool such as ConsoleOne, nor is there always a one-to-one correspondence between an Attribute and a field in tools like ConsoleOne. For example, the **workforceID** Attribute is displayed by ConsoleOne in both the **Employee ID** and **Personal ID** fields of the user's Properties (**User Profile** tab, **Business Info** and **Personal Info** panels).

The **GetFieldValue** method takes two parameters: the name of the Attribute, and a BOOLEAN indicating if the data is being returned to a scalar (**FALSE**) or an array (**TRUE**). The default for the second parameter is **FALSE**, and so it may be omitted for retrieving the Value of a Single-Valued Attribute.

The method returns **NULL** if the Attribute's Value contained no data. It is also possible for the result to be **NULL** if the Attribute had an ACL that prevented the script from accessing the Value, or that the method failed for some other reason; however, it is not possible, within the script, to distinguish these conditions from an empty Attribute.

In this code example, we'll read the CN (*aka* Common Name), **Full Name** and **Description** Attributes of each User Object in our list (as returned by the **Search** in Example 5, and referenced by **\$Entry**). Because **Description** is a Multi-Valued Attribute (MVA), we must read it into an array. We can also print out the Values, if there are any (since **CN** is a required Attribute of a User Object, we know the User Objects will have one):

```

my $UserCN;
my $UserName;
my @UserDescription;
my $cntrA;

# The second parameter of GetFieldValue defaults to FALSE
# and can be omitted if reading a single-valued Attribute
$UserCN = $Entry->GetFieldValue("CN");
print "\nUser Object: $UserCN\n";

# Here is an example of including the second parameter of
# GetFieldValue even though it is the default
$UserName = $Entry->GetFieldValue("Full Name", false);
if ( $UserName )
    { print "\tFull Name: $UserName\n"; }
else

```

```

        { print "\tFull Name Attribute is blank\n"; }

# The data is being returned to an array, the second
#   parameter of GetFieldValue must be true
@UserDescription = $Entry->GetFieldValue("Description", true);
if ( @UserDescription )
    {
    for ( $cntrA = 0 ; $cntrA < @UserDescription ; $cntrA++ )
        { print "\tDescription [$cntrA]: $UserDescription[$cntrA]\n"; }
    }
else
    { print "\tDescription Attribute is blank\n"; }

```

The output should look something like this made-up example:

```

...
User Object: tsmith
  Full Name: Ted Smith
  Description [0]: Promoted to Sales Manager August-15

User Object: bjones
  Full Name: Bill Jones
  Description Attribute is blank

User Object: jmoore
  Full Name: Jim Moore
  Description [0]: Retiring in September
  Description [1]: Promoted to Sales Manager Jan-16

User Object: SysAdmin
  Full Name Attribute is blank
  Description Attribute is blank
...

```

Example 9: Adding a New Object

The same APIs that allow you to access and read eDirectory also allow you to add new Objects. You should be sure you understand the schema of the Object Class, and what Attributes are required, before attempting to do this.

Since you can read files in your Perl script, you can use the API to do mass Object creation – for example, populating student accounts into an eDirectory Tree at the start of a school year, or generating a large number of accounts for temporary/seasonal workers. Our example will be limited to creating a new User Object for a single new user, but the code presented is easily expanded to a larger scale.

We'll build on Example 3, declaring some variables that are simply hard-coded with the data for the new User Object, and getting a pointer to the list of Objects in the current context. We'll create the Object with the **AddElement** method, then populate its Attributes with the **SetFieldValue** method. Finally, the **Update** method will commit the populated Object to eDirectory:

```

# Mandatory Attributes
my $NewObjectCN = "fjohnson";
my $NewUserSurname = "Johnson";

# Optional Attributes
my $NewUserFullName = "Fred Johnson";
my $NewUserTitle = "Consultant";

```

```

my $NewUserDescription = "Windows-to-Linux desktop migration planner";
my $NewUserEmail = "fjohnson\@company.tld";

my $ObjectList;
my $NewObject;

# Get the Object entries for the current context
$ObjectList = $eDirObj->{"Entries"};

# Create an Object of type User
$NewObject = $ObjectList->AddElement( $NewObjectCN, "User" );

# Check success of AddElement method
if ( $NewObject )
{
    # AddElement was successful, populate Attributes

    # Surname is a Mandatory Attribute for User Objects
    if ( $NewObject->SetFieldValue("Surname", $NewUserSurname ) )
    {
        # All Mandatory Attributes populated, now do optional ones
        $NewObject->SetFieldValue("Full Name", $NewUserFullName);
        $NewObject->SetFieldValue("Title", $NewUserTitle);
        $NewObject->SetFieldValue("Description", $NewUserDescription);
        $NewObject->SetFieldValue("Internet Email Address",
            $NewUserEMail");
    }
    else
    {
        # SetFieldValue failed for Mandatory Attribute, exit gracefully
        print LOGFILE "\nERROR setting Surname ", $NewUserSurname. " for
            Object $NewObjectCN\n";
        close (LOGFILE);
        eDirObj->logout();
        die "\nERROR setting Surname ", $NewUserSurname, " for Object
            $NewObjectCN\n";
    }
}
else
{
    # AddElement failed, exit gracefully
    print LOGFILE "\nERROR adding Object $NewObjectCN\n";
    close (LOGFILE);
    eDirObj->logout();
    die "\nERROR adding Object $NewObjectCN\n";
}

# Object fully populated - commit to eDirectory
if ( $NewObject->Update() )
{ print LOGFILE "\nSuccessfully created User Object $NewObjectCN\n"; }
else
{
    # Update failed, exit gracefully
    print LOGFILE "\nERROR committing Object $NewObjectCN to eDirectory\n";
    close (LOGFILE);
    eDirObj->logout();
    die "\nERROR committing Object $NewObjectCN to eDirectory\n";
}

```

The **AddElement** and **SetFieldValue** methods work on the Object in memory; however, **AddElement** will fail if an Object with the same CN already exists. Not until the **Update** method has been successfully invoked is an Object actually created in eDirectory. Note that you can only determine success/failure of a method invocation. eDirectory error codes are not returned by the UCS APIs.

Objects are created in the current context, as stored in the **FullName** Property of the eDirectory Object.

Make sure the context is properly set before creating the Object.

Example 10: Deleting an Existing Object

The UCS APIs also give you the ability to delete Objects from eDirectory, using the **Remove** method. As with adding an Object, the methods operate in the current context of the eDirectory object. Any logic can be used to determine that Object to be deleted, but in this example, we'll simply specify one, and re-use variables from Example 9:

```
# The CN of the Object to be deleted
my $TargetObject = "bjones";

# List of Objects in the current context
$ObjectList = $eDirObj->{"Entries"};

print LOGFILE "\nDeleting $TargetObject\n";

# Does the target Object exist in this context?
$Entry = $ObjectList->Item($TargetObject);
if ( $Entry )
{
    print LOGFILE "\n\t$TargetObject is in context ",
        $eDirObj->{"FullName"}, "\n";
    # Target Object found - delete it
    if ( $ObjectList->Remove($TargetObject) )
        { print LOGFILE "\tSuccessfully deleted $TargetObject\n"; }
    else
        { print LOGFILE "\nERROR attempting to delete $TargetObject\n"; }
}
else
    # Target Object not in this context
    { print LOGFILE "\n$TargetObject does not exist in context ",
        $eDirObj->{"FullName"}, "\n"; }
```

It's important to note that there is no “undo” method. Regrettably, a mistaken Object deletion can only be regretted.

Example 11: Modifying an Existing Object

Using the **GetFieldValue**, **SetFieldValue** and **Update** methods, you can also modify existing Objects in eDirectory. When changing an existing Object, the **SetFieldValue** method may take a third parameter, a syntax not shown in the previous code examples. This parameter is important if the Attribute already has a Value – if so, the Value should be passed as this third parameter. Otherwise, it should be omitted.

Important

Attributes of *string* types (e.g. SyntaxType 3, **CASE IGNORE STRING**; SyntaxType 2, **CASE EXACT STRING**), such as **Description**, may not be set to a zero-length string. See C below.

These code examples assume that **\$Entry** is a pointer to a User Object, already set (as shown in Example 5). We'll start with declaring variables for use in modifying the Object:

```
# Return value of method calls
my $Return;
```

```

# Looping counter for setting MVA Values
my $cntrB;

# New and old Values of Attributes
my $OldTitle;
my $NewTitle = "Vice President";
my @OldPostalAddress;
my @NewPostalAddress = ( " ", "101 Main St", "#10", "Anycity",
    "Earth", "123");
my @OldDescription;

```

A) Change the Value of a Single-Valued Attribute

Changing the Value of an Attribute is done with the **SetFieldValue** method. It is important to determine if the Attribute already has a Value, using the **GetFieldValue** method, as an existing Value must be passed as the third parameter in the **SetFieldValue** call.

```

$OldTitle = $Entry->GetFieldValue( "Title" );
if ( $OldTitle )
    # There is an existing Title, pass it as the
    #   third parameter
    { $Return = $Entry->SetFieldValue("Title",$NewTitle,$OldTitle); }
else
    # The Title Attribute is empty, no third parameter
    { $Return = $Entry->SetFieldValue( "Title", $NewTitle ); }
if ( $Return )
    {
    $Return = $Entry->Update();
    if ( $Return )
        { print LOGFILE "\nSuccessfully changed Title from ",
            $OldTitle, " to ", $NewTitle, "\n"; }
    else
        { print LOGFILE "\nUpdate failed for Title field\n"; }
    }
else
    { print LOGFILE "\nSetFieldValue failed for Title field\n"; }

```

If you know the Attribute has no Value, perhaps because you have just created the Object, then you can skip the step of checking for it and simply omit the third parameter.

B) Change a Value of a Multi-Valued Attribute

The original AppNote article on this particular subject demonstrated using **SetFieldValue** for MVAs and passing the arrays by reference. The APIs have changed, and when setting new Values for MVAs, its necessary to set each Value individually. Thus, to change the six Values of the **Postal Address** Attribute, **SetFieldValue** must be called six times.

```

# Get the user's Full Name for the first line of the Postal Address
$NewPostalAddress[0] = $Entry->GetFieldValue( "Full Name", false );
if ( ! $NewPostalAddress[0] )
    # Full Name Attribute is blank, substitute CN, because
    #   an Attribute Value cannot be set to NULL, and since
    #   CN is a required Attribute, we know it will have a Value
    { $NewPostalAddress[0] = $Entry->GetFieldValue( "CN", false ); }

# Get the old Value of the Postal Address; since this is an MVA, it
#   returns an array, and the second parameter must be TRUE
@OldPostalAddress = $Entry->GetFieldValue( "Postal Address", true );
if ( @OldPostalAddress )

```

```

    {
    # The Value is currently empty
    for ( $cntrB=0 ; $cntrB < @NewPostalAddress ; $cntrB++);
        {
        $Return=$Entry->SetFieldValue("Postal Address",
            $NewPostalAddress[$cntrB]);
        if ( ! $Return )
            # Problem in SetFieldValue - exit loop
            { $cntrB = @NewPostalAddress; }
        }
    }
else
    {
    # The old Value must be overwritten
    for ( $cntrB=0 ; $cntrB < @NewPostalAddress ; $cntrB++);
        {
        $Return=$Entry->SetFieldValue("Postal Address",
            $NewPostalAddress[$cntrB],$OldPostalAddress[$cntrB]);
        if ( ! $Return )
            # Problem in SetFieldValue - exit loop
            { $cntrB = @NewPostalAddress; }
        }
    }
# Only attempt an Update if SetFieldValue was successful
if ( $Return )
    {
    $Return = $Entry->Update();
    if ( ! $Return )
        # Problem in Update
        { print "\nERROR: Update failed for Postal Address\n"; }
    else
        # Success!
        { print "\nSuccessfully changed Postal Address\n"; }
    }
else
    {print "\nERROR: SetFieldValue failed Postal Address [$cntrB]\n";}
}

```

While **SetFieldValue** must be called for each Value of the MVA, the **Update** method only needs to be called once per Object being changed. This is because **SetFieldValue** only changes the Object in memory, and **Update** commits the fully-changed Object to eDirectory.

C) Delete a Value from a Multi-Valued Attribute

The original AppNote omitted this concept entirely, and the API documentation does not explain this operation at all. If your set a *string*-type Value to a zero-length string with **SetFieldValue**, the subsequent call to **Update** will fail.

Deleting a Value is accomplished with the *undef* constant that is part of Perl, as shown in this code that removes the first Value in the Object's **Description** Attribute, while preserving any other Values:

```

@OldDescription = $Entry->GetFieldValue( "Description", true );
if ( @OldDescription )
    {
    $Return = $Entry->SetFieldValue( "Description", undef,
        $OldDescription[0] );
    if ( $Return )
        {
        $Return = $Entry->Update();
        if ( $Return )
            { print "Successfully changed Description\n"; }
        else

```

```

        { print "\nERROR: Update failed for Description\n"; }
    }
    else
    { print "\nERROR: SetFieldValue failed for Description\n"; }
}
else
{ print "Description Attribute has no Values\n"; }

```

Example 12: Detecting Selected Object Parameters

Certain parameters of an Object can be useful in deciding how to handle or process the Object. For example, if you are counting Objects in containers as your script moves through the eDirectory Tree, you may wish to omit Alias Objects from your counts. Or you may want to log User Objects that have been *Disabled* or subject to *Intruder Lock Out*.

Again building on Example 5, we'll assume **\$Entry** is pointing to a specific User Object. We'll declare a few local variables for recording information about the Object:

```

my $IsAlias;
my $IsLockedOut;
my $IsDisabled;

```

A) Alias

Helpfully, the **Alias** Property is a BOOLEAN indicating if the Object is an Alias:

```

$IsAlias = $Entry->Alias();
print "\nObject ", $Entry->GetFieldValue( "CN", false ), " is ";
if ( $IsAlias )
    { print " an Alias\n"; }
else
    { print " not an Alias\n"; }

```

B) Locked Out

Using the **GetFieldValue** method, you can retrieve the BOOLEAN Value indicating if the User Object has been locked out by Intruder Detection:

```

$IsLockedOut = $Entry->GetFieldValue( "Locked By Intruder", false);
if ( $IsLockedOut )
    { print "\nObject ", $Entry->GetFieldValue( "CN", false ),
      " is locked out\n"; }

```

C) Disabled

Similarly, there is a BOOLEAN Value showing if the User Object has had its logins Disabled:

```

$IsDisabled = $Entry->GetFieldValue( "Login Disabled", false);
if ( $IsDisabled )
    { print "\nObject ", $Entry->GetFieldValue( "CN", false ),
      " is disabled\n"; }

```

Coding Pitfalls and Debugging

The lack of Perl-specific documentation for the UCS APIs is perhaps the most limiting aspect of coding tools to use them. An example is the fact that using the *undef* Perl constant to delete a Value from an MVA is, as of this writing, completely undocumented.

It is important to note that the **SetFieldValue** method only changes the Object in memory. The changes are not actually made until the **Update** method is called. Its also possible for the **SetFieldValue** method to succeed and a problem not show until **Update** is used.

Another limiting factor is the fact that eDirectory errors are not returned via the APIs, only simple success/failure. You will need to use **DSTRACE.NLM** to capture the eDirectory error codes. These NetWare console commands will help you do that:

```
load dstrace
dstrace +areq
dstrace +abuf
dstrace +auth
dstrace screen on
dstrace file on
```

Setting the **PrefServer** Property of the eDirectory Object, as shown in Example 2, can be used to limit where you would need to run **DSTRACE.NLM**. Optimally, the Preferred Server will host a Master or Read/Write Replica of every Partition holding an Object that your script will access.

Conclusion

This article has only scratched the surface of what's possible. System administrators can script mass operations and perform large-scale changes in an efficient manner. Using Apache with *mod_perl*, its possible to create web-enabled applications can access and manipulate eDirectory.

Finally, eDirectory is only one of the UCS APIs. The ability to access and control the NetWare OS, NetWare-hosted file systems, UPSes, Fax Servers, communications ports, and services such as FTP and NNTP are all part of the UCS APIs.