Cisco Unity Connection .NET ODBC SDK

Contents

Overview....1

Requirements/Special Notes....2

Why ODBC, Isn't That Going Away?....2

Won't ODBC Projects Break With Every New Release?....3

Installing and Using the Library in Your Project....3

Using the .NET ODBC SDK....3

- Getting Started....4
  - Task 1: Install the IBM Informix Client SDK.....4
  - Task 2: Configure a User with the Remote Administrator and System Administrator roles....4
  - Task 3: Set the Database Proxy Service Shutdown Time....4
  - Task 4: Activate the Remote Database Proxy Service....4
  - Logging into Connection....5
  - The DbFetchResult Class....5
  - Logging and Debugging with the SDK....5
  - CUDLI Is Your Friend....6
- Fetching Data....6
  - Data Tables and Data Readers....6
  - Filling a Data Table....7
  - Getting a Count....8
  - Getting a Single Value....8
  - Filling a Data Reader....8
- Stored Procedure Overview....9
  - Stored Procedure Naming Strategy....10
  - Return Values from Stored Procedures....10
- Changing Database Focus....12
- Checking Connection Version....12
- Checking for Extension Conflicts....12
- Finding Objects by Name, Extension and/or Alias....13

Revision History....13

Overview

The .NET SDK library for the Unity Connection ODBC interfaces is a set of library code intended to make development of applications for Unity Connection in the .NET framework easier, faster and less error prone. While it does require you have IBM's Informix Client SDK installed for typical projects you do not need to include the IBM Informix class library in your project – you can do everything you need to do with the simple methods exposed off the SDK and the build in data tables and types available in .NET.

The idea is not just to simplify the interaction with the Informix client libraries (which is certainly the case) but to allow applications to be written against other database drivers in the future if and when Unity Connection moves off Informix. This is the same library used internally by the Unity Connection tools team for developing many of its applications that interact with Unity Connection via ODBC. Coupled with the REST SDKs already available they provide a powerful toolset for just about any application type you wish to create against Connection.

Requirements/Special Notes

The .NET ODBC SDK is written and tested against all versions of Unity Connection 7.0(2) and later. The ODBC interface is supported in 7.0(2) builds and later although it appeared in 2.x versions prior to that.

Use of the SDK is not supported by TAC – in other words if you need assistance with an application you are developing using the SDK, TAC is not going to provide that assistance for you. Support is through development services and the public support forums. Check the www.CiscoUnityTools.com site on the “links” page for current links to get to those forums. “Not supported by TAC” does not mean the SDK is somehow dangerous or risky to use. The ODBC interface has been around for many years and is not going anywhere.

Any .NET framework can use these libraries. This means you can, of course, develop desktop and web applications on Windows using C#, VB.NET, Iron Python etc… While technically possible to work in Linux and Mac using Mono, this is not currently supported given the differences in the IBM drivers on the client side needed to interact with Connection via ODBC. Separate implementations would have to be provided to work with these – which may happen if there’s demand, currently testing is limited to Windows platforms only.

Why ODBC, Isn’t That Going Away?

I get this one all the time so let me just address it here. There are a number of very good reasons for doing applications that use direct ODBC access and Unity Connection will continue to support ODBC via the proxy service into the future. There are no plans to deprecate this service and in fact with the Unity Connection 10.0 the service is being made easier to setup (fewer steps) and the service now stays on forever with no shut down timer.

Reason number 1 is for support on older versions of Unity Connection. As noted above you can go back to 7.0(2) (and actually into the 2.x versions if necessary). REST interfaces didn’t start appearing till late in the 8.x line.

Reason number 2 is that REST does not provide complete coverage for all administrative tasks and data. With the release of 10.0 the REST APIs for Connection have taken huge strides and the API is considerably more complete and robust providing access to the vast majority of what administrators need, there are still “corner case” items not exposed via REST. ODBC is, by definition, always complete since the administration and back end features are built using the views and stored procedures exposed.

Reason number 3, and this is a big one, is performance. For individual object editing and smaller data sets, user settings, inbox views etc… REST is an ideal solution given its cross platform capabilities and ease of use from just about any development platform on the planet. However, when you need to do a LOT of editing (think bulk editing of 10,000 objects) direct ODBC access will be 20 to 30 times faster. Yes, I said times, not percent. If I rewrote a tool like COBRAS to migrate data using only REST instead of ODBC migrations would take days.

That last item throws some folks. I often get “can’t you just make REST work faster”? Well, certainly there are efficiencies that can be obtained via any HTTP based protocol such as white listing which properties you want fetched and providing task-specific methods that “wrap up” numerous complex calls needed on the back end instead of requiring a series of HTTP request/response calls from the client. But there’s a couple things that will always make a typical REST interface much slower than direct database access via a binary transport.

The aforementioned HTTP request/response will always provide overhead – the data needs to be converted into a text representation of JSON or XML, sent across, the receiving party parses it out and converts it back into a dictionary or class instance (a process known as “serialization” and “deserialization”). This is always going to be slower than sending a well formed SQL query via ODBC and getting a dataset or record set back (I’ll explain the difference between these later).

The other thing that tends to throttle REST based API is the REST philosophy of working with “nouns” and “verbs” – I’m not going to get into a dissertation on REST best practices here but the general idea is applications work with “nouns” (an object such as a user or a call handler in our case) using verbs (add, delete, modify, get). The idea is you don’t need to know much about the overall object model up front, you can “drill down” to what you need at the object level and part of the results will have URIs to where you need to go to get related information. For instance when you fetch a user as part of the result set you get a URI to go get the greetings or transfer rules associated with that user.

That last bit is part of the rub. Say for instance I want to take everyone in a Class of Service and set their standard transfer rule to not ring their phone. Via ODBC I can find the ObjectIds (unique identifiers) for all the transfer rules I want to update in one SQL query and then simply loop through them calling a stored procedure to update the transfer rules behavior one at a time (yes, you could just do a direct table update but then you’d be an evil hack who needs to step away from the keyboard immediately). In REST you need to get the list of users associated with the COS which sends a lot of data by itself, then you need to fetch the URI for the standard transfer rule for each user one at a time and then execute a POST for that rule to update its behavior one at a time. This takes much, much longer.

The short version is you send orders of magnitude more data back and forth across the wire and have to serialize/de serialize it on both sides over and over again. The idea that you can simply "speed this process up" somehow isn't really realistic. It can be optimized but you'll never get within spitting distance of direct database access. Yes, you can simply execute SQL queries via an HTTP protocol and call stored procs that way (think AXL type interfaces here) but then you aren't really conforming to REST standards and you'll still have significant HTTP processing overhead involved.

Yes, using ODBC is more difficult. It requires installation of a 3rd party client library to use and a fairly deep understanding of the target system's data model. It's not for everyone and not appropriate to all projects. But when you need to move a lot of data fast, it simply cannot be beat.

Won't ODBC Projects Break With Every New Release?

This is the 2nd most common question I get and I just want to address it in its own section here.

Short answer: No, of course not.

Longer answer: If you use views (instead of tables) when fetching data and always (always!) update/create/delete via stored procedures you will never have a problem here. I've been writing ODBC based tools for Unity and Unity Connection for 15+ years now and have never once gotten burned by a data model change between versions. Not once.

We always make sure views and stored procs are backwards compatible so if you stick to the rules of the road you'll be fine. It's exceedingly rare that any item be _removed_ from the object model entirely. Even if that does happen (I can think of perhaps 2 instances) the item is always just deprecated in the data dictionary, it's not actually removed from the table definition itself.

Installing and Using the Library in Your Project

The Easy Way

The SDK is available via NuGet package manager and by far the easiest way to install it is to manage NuGet packages for your project, select online and search on "Cisco Unity Connection Odbc SDK" and the project should come right up. After adding it all you need to do is add the "Cisco.UnityConnection.OdbcSdk" to your using list and you're off to the races.

NuGet is nice since it always grabs the latest and notifies you when an updated version of the library is available and gives you the option of installing it. It couldn't be easier.

The Hard Way

You can still download the source code for the project and include it manually. This can be handy if you prefer to debug into the SDK library code directly and see what's going on or the like or if you just don't trust NuGet or doing things the easy way makes you generally suspicious. Whatever works for you.

Using any SubVersion client you like you'll need to download the SDK project code off the site's download page:
http://www.ciscounitytools.com/CodeSamples/Connection/ODBC/ODBC.html

To add the project right click on your solution node in the solution explorer in Visual Studio. Select "Add" and then "Existing Project" and a file explorer window will appear. Navigate to where you downloaded the library code and select the "CiscoUnityConnectionServerOdbc.csproj" file. This will pull the library into your solution and have it build when you rebuild your project. This will result in the "CiscoUnityConnectionServerOdbc.dll" ending up in the target BIN output (debug or release) for your project. This is the only file you need to include in your setup for the library.

Once you've included the project you then need to add a reference to it in your project – in your project right click the "references" node in the solution explorer and select "add reference" – in the resulting dialog select "projects" and select the CiscoUnityConnectionServerOdbc project and add it. Then you only need to add a "using `Cisco.UnityConnection.OdbcSdk`;" directive in your project and you're off to the races.

NOTE: Visual Studio has the annoying habit of defaulting to the ".Net Framework 4 Client Profile" as the default for new projects. This will not work for us as the SDK requires the full ".Net Framework 4" setting. Be sure your project is configured for this or you'll get build errors.

Using the .NET ODBC SDK

This document uses a "task based" approach to demonstrating the use of the library – each major object class (user, call handler, name lookup handler, schedule etc…) has it's section and small code snippets are shown demonstrating the items you'd typically want to do with those objects. This does not attempt to document the entire data schema or get into too much theory. As a developer I know I learn faster with a simple “show me” approach so that’s what I endeavor to do here.

Getting Started

To attach to Unity Connection from you Windows client you need to make sure you have:

1. The IBM Client SDK needs to be installed (at least the ADO .NET driver at a minimum)
2. The database proxy service turned on
3. An account configured with the remote administrator role
4. Port 20532 needs to be open between your client and the Unity Connection server. All communication via ODBC goes across this one port so if you’re not also using REST, SMTP or the like this is the only port that needs to be open.

The following 4 tasks need to be done.

Task 1: Install the IBM Informix Client SDK.

Go to the ODBC driver download page:
http://www.ciscounitytools.com/Applications/CxN/InformixODBC/InformixODBC.html

You can either download the current version recommend (at the time of this writing 3.7) or go to IBM’s site and download the latest. You can install the entire SDK if you like but only and .NET ADO drivers and localization files are necessary – the rest is optional.

All tools published off http://CiscoUnityTools.com for Unity Connection are strictly 32 bit to keep install and testing simple – however you can download and install the 32 or 64 bit drivers as you prefer. The SDK is compiled to work with either version. The need for 64 bit drivers is dubious, you will not experience any increase in speed or really any other advantage which is one of the reasons we keep our tools simple with 32 bit only.

Task 2: Configure a User with the Remote Administrator and System Administrator roles

1. Go to the Cisco Unity Connection Administration web interface for your installation. You can leverage a user with or without a mailbox for off box data access purpose.

3. Be sure the web administration password for this user is not configured to require a change at first login on the “Password Policy” page for that user.

4. If necessary, change the web administration password on the “Change Password” page. Note that only the web application password comes into play for remote data access.

5. Finally, on the “Role” page for the user, add the “**Remote Administrator**” and the “**System Administrator**” roles to the “Assigned Roles” list and save. You can assign any or all other roles as well but for the purposes of remote access to the database and making updates to users those two are necessary.

Task 3: Set the Database Proxy Service Shutdown Time

NOTE: If you are running Unity Connection 10.0 there is no shutdown time, this step is not necessary.

Out of the box the database proxy service is not running and if you try to start the service it will shut down right away. First you need to set the “Database Proxy: Service Shutdown Timer” value found in the System Settings -> Advanced -> Connection Administration section of the Cisco Unity Connection Administration page. By default this is 0. You can set it to as high as 999 days if you like. After the number of days configured here the remote database proxy service will shut down.

Task 4: Activate the Remote Database Proxy Service

1. Out of the box the service that listens to remote database requests is not active, you must turn it on. To do this, go to the “Cisco Unity Connection Serviceability” web admin page.

2. On the Tools menu, select the “Service Management” page.

3. The “Connection Database Proxy” item under the “Optional Services” section will be marked as “Deactivated” and stopped. Press the “Activate” button and it will be activated and started automatically.

Once you’ve started the proxy service you can connect with any tool that needs off box database access using the user name, web administration password and port “20532”.

NOTE: The service will automatically shut down after the number of days configured in step 2 above or if you restart the server unless you’re running Unity Connection 10.0 or later in which case there is no timer.

Logging into Connection

The SDK is designed to support multiply threaded applications that may be attached to more than one Connection server at a time (for instance a network of Connection clusters). As such you can create multiple instances of the **UnityConnectionServerOdbcSdk** class at the same time. If you have multiple threads acting on a single instance of the class then when building and running stored procedures it will be serialized such that one stored procedure is built and run at a time. The SDK hides the complexity from you here and so long as your application is reasonably quick about constructing your stored procedure and calling it you should have no problems if multiple threads are all calling stored procedures on a single instance.

The login sequence using the class is a simple two step process that looks like this:

```
UnityConnectionServerOdbcSdk server = new
        UnityConnectionServerOdbcSdk ("TestApplication");

var res= server.LoginDatabaseBlocking ("192.168.0.197", "dbdude", "labPw");

if (res.Successful == false)
{
    Console.WriteLine("Failed to log in:"+res);
    return;
}

Console.WriteLine("Logged into:" + server);
```

We'll look more at the return structure from calls into the SDK and general error handling later. The important items to take away here is the server construction line and the use of the LoginDatabaseBlocking call to do the attachment.

The first line passes the "TestApplication" string into the constructor for the main class in the SDK. This creates a new database functions interface in the server and the name there shows up in all Unity Connection audit logs for calls you make via ODBC – be sure to put a meaningful name in here for diagnostic purposes.

The LoginDatabaseBlocking should be pretty obvious – it's a blocking call vs. a background call. We may provide background versions but at this point I doubt it since it's easy enough to launch your own thread for such things in .NET, particularly in .NET 4.5 which makes this all but trivial. We'll look at what's returned by that method (and most methods in the SDK) in the next section.

The DbFetchResult Class

Throughout the library you will see most methods will return an instance of the DbFetchResult instead of a simple integer or a bool. There is a method to the madness here – the DbFetchResult class holds all the information about what was requested and what came back from the server so diagnosing what went wrong and what, exactly the error details were is much easier. The SDK itself does not have logging built into it, naturally, so it has to return to the calling application as much detailed and useful data as it can so YOU can log it properly in your application. That's the idea.

The "ToString()" override for the class includes all details of the query and results, error code etc… in one shot – makes logging out failures a simple process. The query string returned in the class includes all parameters and their values (for either a query or a stored procedure call) so it's a simple matter of dropping the results to your log file or console or whatever using the ToString() and you have everything you need for running down the problem.

Logging and Debugging with the SDK

Since I've been asked a few times, let me just state up front here that the SDK is not *supposed* to log to a file on the hard drive for you. Most of what you need for your own error handling and logging is passed back as part of the DbFetchResult class mentioned above. Since the SDK can be (and is) used in a variety of application types such as desktop applications, services and web servers it cannot assume access to the local file system for logging purposes. It does provide a few event handles you can wire up to provide more "dialog like" logging in your application if you prefer and/or can provide more diagnostic output you can handle as you like at your application level as disused in this next section. The UnityConnectrionServerOdbcSdk object exposes a couple of events you can use if you wish to be notified of any error and, optionally, debug event data that you can "hook" in your application to provide a more "dialog" logging output for instance. As noted above most calls in the SDK return a DbFetchResult class instance that has all the error and details of what was sent/received from the server that you'd need. For any errors that may take place on the back end which are not tied directly to a method call with a DbFetchResult return, you can hook the ErrorEvents event off the UnityConnectrionServerOdbcSdk class.

This is easy to setup in .NET. After creating the server object you can add this line of code:

```
        server.ErrorEvents += ServerOnErrorEvents;
```

Then the definition for the method that fires when the error event is raised looks like this:

```
private static void ServerOnErrorEvents(object sender,
            ConnectionServer.LogEventArgs logEventArgs)
        {
            Logger.Log("[ERROR]:"+logEventArgs.Line);
        }
```

Nothing too fancy – Any and all errors that are encountered on the server class will show up in the log now where you can spot them.

Similarly you can wire up the debug event that can also be useful, however you should only do this if you're having a specific problem you're trying to diagnose – you should NEVER have this enabled in a production application because the debug output is VERY chatty. However if you need to see what's going on with a customer's system or the like you can dump the traffic information out by wiring up the event like this:

```
server.DebugEvents += ServerOnDebugEvents;
server.DebugMode = true;
```

Notice that you have to turn debug mode on – if you don't do that (it's off by default) nothing will be raised out of the event. Then a very similar signature is used for the event that is fired on the debug event as the error event:

```
private static void ServerOnDebugEvents(object sender,
            ConnectionServer.LogEventArgs logEventArgs)
{
    Logger.Log("[DEBUG]:" + logEventArgs.Line);
}
```

Not too tricky. Again, though, I highly encourage folks to wire up and alert/log on error events but leave the debug events out of the picture unless you have a driving need for them in a particular scenario.

CUDLI Is Your Friend

If you're going to be doing anything with SQL queries into Unity Connection your first stop needs to be CiscoUnityTools.com and pick up the latest version of the data link explorer application CUDLI. This application contains a complete data dictionary for all major releases of Unity Connection and has many, many helpful functions for finding the data or procedure you want to run, what parameters it takes, what values are legal for them etc… etc… it will save you much time and make you considerably more effective developing applications. It's on every desktop I develop on without exception and its up and running most of the time. Get your copy here:

http://www.ciscounitytools.com/Applications/CxN/CUDLI/CUDLI.html

Fetching Data

Data Tables and Data Readers

Before we get started I'd like to briefly cover the mechanics of getting data off a server and to a client via ODBC. Naturally you can fetch individual properties or a scalar value (number) that's just a single item – however in most cases you're doing a fetch of more than one row of data consisting of more than one column. There are two approaches to doing this: fill a dataset which is essentially a two dimensional array with information from the query or establish a data reader for reading through all the rows.

On the surface these two look fairly similar; however under the covers they behave very differently. In short unless you have a very specific reason for doing so, always use data tables.

The SDK is designed to fetch either single values, scalar values or return data tables filled with the results of queries. This follows a very simple pattern of fetching data, filling a structure, passing it back and being ready for the next call. No pervasive connection is maintained to the server via ODBC, it's a strict "in and out" operation and scales well and is nice and clean. Further this uses only native data types in .NET – the DataTable class is part of the standard system data library and this works across different drivers and database engines without issue.

A DataReader on the other hand is an implementation of the old “firehose cursor” some of you old school database folks may remember. It’s a forward only, read only data reader that maintains a connection to the server until you destroy the reader. It’s got all kinds of intelligence built into it to prefetch rows of data as you iterate over the result set. It’s quick, it’s easy and it can handle iterating over an enormous return set (think hundreds of thousands of rows) reasonably efficiently.

The down side, however, is that it uses a proprietary type (for instance the IfxDataReader for the Informix .NET driver) that requires you include their development library in your project and until you destroy the object it maintains its own separate connection to the server via ODBC. If you open 10 data readers you will have 10 separate connections to the database if you’re not careful. Unity Connection will stop allowing more attachments after 10 so then your application will start behaving oddly.

The vast majority of my applications do not use data readers at all for any reason. One notable exception is the User Data Dump – it uses one data reader to iterate over all users in the system and then does supplemental data fetches for additional information via data tables. Given the large amount of data associated with a user and the large numbers of users (20,000) possible, it made sense to employ the efficiency of a data reader for this purpose. To be clear, then, User Data Dump actually uses two ODBC connections to the server for the duration of its run.

For these edge case needs the SDK does provide a mechanism for leveraging data readers in your applications. That said, be sure to only use it if you absolutely need it. A data table can easily accommodate thousands of rows of data without significantly impacting performance or memory in a typical application and is a much better design choice on the whole.

Filling a Data Table

Let’s start by looking at one of the most common tasks you’ll want to do with the SDK and that’s filling a data table based on an SQL query. Here’s a typical example where we’re pulling the alias and ObjectId (unique GUID) for all subscribers (users with mailboxes) that have gone through the first time enrollment conversation and are in Seattle:

```
DataTable oTable;
const string strSql = "SELECT Alias, ObjectId FROM vw_Subscriber WHERE " +
                      "fn_tolower(city)=? AND isVmEnrolled=?";

res = server.FillDataTableBlocking(strSql, out oTable, false, "seattle",true);

if (!res.Successful)
{
    Console.WriteLine("Fetch failed:"+res);
    return;
}

Console.WriteLine("Seattle users:");

foreach (DataRow oRow in oTable.Rows)
{
    Console.WriteLine("Alias="+oRow["Alias"]);
}
```

There are a couple important items in here to point out in this example.

First, the use parameterized query structure. Those “?” in the query serve as placeholders that get replaced with values in the FillDataTableBlocking method. In this case the string “seattle” and “true” Boolean value. It’s important that you always use parameterized queries both for safety and for efficiency. If you’re looping through and issuing the same query but with different values if you’re using parameterized queries the server will be quicker since it compiles the query itself and keeps it in memory even though the parameters passed to the query can change. It’s good style and good for you.

Second, you can mix types for your parameters. Notice that you can pass strings, Booleans and numbers of various types in your parameter list – they are converted and added as necessary.

Third, the use of the “fn_tolower(xxx)” function. Unlike some other databases such as Access, queries against string fields cannot be assumed to be case insensitive with Informix. You need to use the fn_tolower() function to force the comparison to happen with lower case and, of course, remember to pass your parameterized string as lower case as well to get matches.

Fourth, fetch specific values. In the above example only the Alias and ObjectId are being pulled off the vw_Subscriber view which has many columns. Try and avoid “SELECT *” if you can. There are times where it may be appropriate but rarely.

Typically you’d make a special check for no rows being returned. This is the purpose of the first Boolean flag in the FillDataTableBlocking parameters which is passed as false by default. A value of false means it won’t return a failure if there are no matches. If you pass this as true a query you're expecting to have at least one match would return a failure if the data table returned was empty.

Getting a Count

Executing a "scalar query" is simply getting a count using a SQL query – these are extremely fast on the server side and light on the wire since only a single integer gets passed back. In cases where you just need to grab an up-front count for an operation this is much faster than getting a data table and counting the rows up if you don't need the data.

```
const string strSql = "SELECT COUNT(*) FROM vw_Subscriber WHERE ListInDirectory=?";

int iCount = server.GetCount(strSql,true);

Console.WriteLine("Directory list count="+iCount);
```

Notice that there's no DbFetchResult class returned in this case, just the count. This is a good reason to have the ErrorEvent wired up so if there is a malformed query passed you don't just report 0 and move on without note.

Yes, you can just execute fill data table method instead and the result is returned in the first column of the first row (only row) returned. It's a bit simpler using this call, however, as you don't need to convert types or the like. However doing it this way returns the DbFetchResult class and the table as an out parameter which is a little more code but may be more appealing to you depending on the application.

Getting a Single Value

Another shortcut method provided for convenience is the ability to fetch a single string value from a query. Yes, you can just fill a data table and pull it out of the rows/columns in the returned table – however it's something that needs to be done so often that it's provided as a one line shorthand method that looks like this:

```
const string strSql = "SELECT OjectId FROM vw_Subscriber WHERE Alias=?";

string strObjectId = server.DatabaseFunctions.GetSingleValueFromQuery(strSql);

Console.WriteLine("ObjectId="+strObjectId);
```

Again, note that a DbFetchResult class is not returned, just a string value. No matter what field you're passing back it's always passed back as a string. If no match is found or there's an error a blank is returned. Again, you can always just fill a data table via the FillDataTableBlocking method and fish the value out of the first row and first column instead, this is provided merely as a shorthand time saver for very common needs such as pulling values based on unique identifiers or the like.

Filling a Data Reader

If you intend on using a data reader in your application you need to add the IBM.Data.Informix library to your application. This means browsing to the DLL which can be found by default under **c:\program files(x86)\IBM Informix Client SDK\bin\netf20\** (or wherever you installed it if you didn't take the defaults).

Filling a data reader looks a lot like filling a data table we've already seen - it would look like this:

```
const string strSql = "SELECT Alias, ObjectId FROM vw_subscriber";

IfxDataReader oReader;

res = server.FillDataReaderInformix(strSql,out oReader);

if (res.Successful == false)
{
    Console.WriteLine("Fetch failed:"+res);
     return;
 }

while (oReader.Read())
{
    Console.WriteLine("Alias="+oReader["alias"]);
}
```

```
        oReader.Dispose();
```

This is a pretty similar process to using a data reader except you don't need to iterate over the rows of a table, instead you need to execute "read" until it returns false (indicating the end of the record is reached). Notice that you need to execute a read first before fetching data from the current row – when the reader is loaded the row is empty until you execute the Read() method.

Also note that the reader provides no "count" method – you have no way of knowing up front how many rows you'll be processing so if you need that information (for instance for reporting progress through a large set) you'll need to issue a separate COUNT(*) query before getting your reader.

Finally notice the explicit dispose of the reader – technically this will be done when you exit scope but when using a reader I like to be explicit about this to avoid accidents. Remember, it maintains its own separate connection to the database via ODBC until it's destroyed.

Stored Procedure Overview

Stored procedures are your friend. I know many folks don't want to deal with stored procedures and want to just "get to the metal" and update data in tables directly. Mostly I think this is because they don't really understand what a stored procedure does for them or how they work. The SDK is written to assume ALL updates, creates and deletes ware done via stored process exclusively. If you're doing anything other than that you're doing it wrong, period.

So, what exactly is a stored procedure? Basically it's a function call that takes a variable number of parameters. There's quite a bit of logic in the stored procedures for Connection that check for legal value, conflicts and such which is one of the reasons they're so much safer to use than direct table edits. For instance when you delete a user via a stored procedure you can pass in a replacement user's ID to fix up any external references to the user being removed automatically. Very handy. Without it the foreign key constraints on the table definitions would simply not allow you to remove the user until you manually went around and removed all references to them first. Not fun.

You can execute a stored procedure just like you can a query. For instance the stored procedure call to create a new subscriber in Unity Connection could look something like this:

> execute procedure csp_subscribercreate (pAlias='newguy', pDtmfAccessId='555123', pTemplateAlias='voicemailusertemplate')

It will return the GUID (ObjectId) of the newly created user if it succeeds (i.e. if there's not a conflicting alias or extension or the like).

So that's not so scary. Where it gets a little more challenging is when you have to "cast" values such as LVARCHARs and handle the rather fussy date/time format used by Informix. The SDK provides a full set of methods to create stored procedures and add parameters to them to help isolate you from all the complexities here so your code should be nice and clean and not have to change if and when providers and/or databases change out.

So let's see the code that would generate the same stored procedure to add a new user seen above:

```
        server.StartNewCommand("csp_SubscriberCreate");
        server.AddCommandParam("pAlias", DbParamType.VarChar, "NewGuy");
        server.AddCommandParam("pDtmfAccessId", DbParamType.VarChar, "555123");
        server.AddCommandParam("pTemplateAlias",DbParamType.VarChar,"myTemplate");

        string strNewObjectId;
        res = server.ExecuteProc(out strNewObjectId, "ObjectId");

        if (!res.Successful)
        {
            Console.WriteLine("Proc failed:"+res);
            return;
        }

        Console.WriteLine("New user created, id="+strNewObjectId);
```

Pretty straight forward – and yes, I had CUDLI open and was looking at the parameters the csp_SubscriberCreate procedure accepts including what their types were (all VarChars in the example here). As noted you'll want CUDLI installed and open while you work.

Also note that the SDK will automatically add the **AuditAlias** and **AuditComponent** fields in every stored procedure that supports them (almost all of them do). The AuditComponent is the string you passed into the database functions constructor when you created the UnityConnectionServerOdbcSdk class instance. The AuditAlias is the currently logged in user name and domain on the Windows client you're running on. This helps you be a good Unity Connection citizen by logging those details in the audit log on Connection for later troubleshooting purposes.

Stored Procedure Naming Strategy

All stored procedures added by Unity Connection start with "csp_" – which stands for Connection Stored Procedure. There are, of course, system level stored procedures also present in the database but these are not for public use and tools like CUDLI will not show them to you.

If you pop open CUDLI and browse the stored procedures for Unity Connection you'll see there are quite a lot of them. Depending on the version there are 800+ procedures. However, not to worry – a good chunk of them can be ignored and the rest are easily understood based on a simple naming convention.

First, any procedure that contains the sub string "**_gen_**" in it can be ignored. So for instanced "**csp_timezone_gen_tr_insert**" or "**csp_user_gen_tr_update**" can be ignored, these are not intended for use in applications, they get used during installation/configuration or are used on the back end. Using CUDLI you can still see them and, in fact, review the source code for the stored procedures themselves if you're curious but you shouldn't be worrying about these for use in your applications.

The remaining stored procedures in the list come in sets of 3 for create, modify, delete methods and the stored procedure name always starts with the name of the object you're working with. There's a few special functions in addition to the basic 3 here and there (for instance csp_BroadcastMessagePurge) but for the most part all stored procedure names start with the object type followed by the 3 functions.

So for instance if you want to create, modify or delete a subscriber (a user with a mailbox) you would use these three procedures:

csp_SubscriberCreate

csp_SubscriberDelete

csp_SubscriberModify

You'll also notice that all parameters for all procedures are prefixed with a lower case "p" – this is a database naming convention and I know it trips folks up now and again, so don't forget when passing your parameters to include the p in there.

You'll also see some stored procedures that have "get" embedded in them – around 20 of them. These are shorthand methods used by the web based administration interface (CUCA) and most of them are not going to be of interest to application developers. For instance "csp_GetLocalizedText" is used by the admin pages to pull resource strings in different languages for the administration interface. There are a few you may find interesting if you look but I don't use these in my applications as I prefer to use explicit database queries to get what I need. For instance "csp_GetObjectByAliasAndLocation" might be useful however I prefer to fetch this information with an explicit query in my applications.

Each procedure will have a set of required parameters – in CUDLI these show up with the "REQ" column checked – if you call a stored procedure and miss a required parameter you'll often get an error message about the stored procedure name not being found or recognized which may throw you off – it's not the stored procedure NAME it's not finding, it's not finding the full method signature which includes the required parameters so if you see that don't tear your hair out about the spelling of the proc and instead look at your required parameters.

Return Values from Stored Procedures

As a rule, all the create stored procedures return 1 or more values and all the rest of the stored procedures do not return anything. Why do you care? If you call a stored procedure that returns a value without getting that value the call will fail. Trying to fetch a return value when one is not present will also fail of course. So you need to be sure to call the procedure you are using with the correct expectation.

The vast majority of the create stored procedures return a "pObjectId" string indicating the new unique identifier for the newly created object. If you're curious about a particular stored procedure's behavior you can simply select the procedure in CUDLI and then use the "Source" view button to see the stored procedure source code. The Informix style stored procedure code may look a little odd to you but up at the top where the "CREATE PROCEDURE" is seen should be fairly obvious as a method signature. For instance the csp_BroadcastMessageCreate procedure starts like this:

```
CREATE PROCEDURE csp_BroadcastMessageCreate (
                pAuditAlias varchar(192) DEFAULT NULL,
                pAuditComponent varchar(64) DEFAULT NULL,
                pEndDate datetime year to fraction DEFAULT NULL,
                pObjectId char(36) DEFAULT NULL,
                pStartDate datetime year to fraction DEFAULT NULL,
                pStreamFile varchar(40),
                pSubscriberObjectId char(36),
                    pVideoSessionId varchar(191) DEFAULT NULL
)
RETURNS
        char(36) AS pObjectId;
```

It should be fairly obvious this stored procedure returns a string (36 characters long) named "pObjectId". Similarly the csp_BroadcastMessageDelete looks like this:

```
CREATE PROCEDURE csp_BroadcastMessageDelete (
                    pAuditAlias varchar(192) DEFAULT NULL,
                    pAuditComponent varchar(64) DEFAULT NULL,
                    pObjectId char(36)
)
```

Note the lack of a "RETURNS" clause there – this procedure does not return anything.

So, what does this mean for the SDK? Let's see how we call the above two stored procedures using the SDK framework and see how they differ. First, let's create a new broadcast message. This requires we already uploaded a WAV file as a stream file (done using CUPI REST API which is not covered here) and you have a subscriber's ID to act as the owner (sender) of the message. We'll set the broadcast message to live for 5 days max. The code looks like this:

```
server.StartNewCommand("csp_BroadcastMessageCreate");
server.AddCommandParam("pSubscriberObjectId", DbParamType.Char, strUserObjectId);
server.AddCommandParam("pStreamFile", DbParamType.VarChar, strStreamFileId);
server.AddCommandParam("pEndDate", DbParamType.DateTime, DateTime.Now.AddDays(5));

string strNewObjectId;
res = server.ExecuteProc(out strNewObjectId);

if (!res.Successful)
{
    Console.WriteLine("Proc failed:"+res);
    return;
}

Console.WriteLine("New broadcast message created, id="+strNewObjectId);
```

Of note here is the out parameter for the new objectId – if that is not passed the call to the stored procedure will fail since it's expecting to pass back a value for the new Id here. If we then turn around and delete that same broadcast message the call would look like this:

```
server.StartNewCommand("csp_BroadcastMessage");
server.AddCommandParam("pObjectIdObjectId", DbParamType.Char, strNewObjectId);

res = server.ExecuteProc();

if (!res.Successful)
{
    Console.WriteLine("Proc failed:"+res);
    return;
}

Console.WriteLine("Broadcast message deleted.");
```

Not too bad, right? For stored procedures that don't return a value simply don't pass an out string parameter and you're good to go.

So at this point you've seen how to fetch data as a table, number or string and how to create/delete/modify data using stored procedures using the SDK. You now know everything you need to for writing applications using ODBC – what follows is some examples to help flesh out common tasks but for the most part everything you need to know is contained in the CUDLI interface via the data dictionary notes and the stored procedure and view/table interfaces.

Changing Database Focus

By default when you attach to the Unity Connection server you are connected to the **UnityDirDb** database. This is by far the most common database and holds all directory information for users, call handlers, interviewers etc… for creating, editing and deleting directory objects this is the database you need to be attached to. There are, however, 3 other databases you can attach to (and explore in CUDLI if you like). Here's the list of all database names you can attach to:

- **UnityDirDb**. The main directory database which is the default when attaching to a Connection server.
- **UnityDynDb**. This is not a table you'd normally need to attach to as it contains transient data used for network synchronization and other service tasks used by Connection. There isn't anything in here that a typical application would be interested in and certainly nothing you'd need to create or modify.
- **UnityRptDb**. All reporting data used for generating any of the Unity Connection reports available gets pulled from the logs into tables in this database by a scavenger process about every 30 minutes or so.
- **UnityMbxDb1**. Actually there can be up to 6 of these, 1 through 6. This stores the mailbox and messages data for users. I'm often asked why it's necessary to support up to 6 of these and it has nothing to do with capacity (total messaging capacity is limited by the disk size of the server). You'd create separate mail databases such that backing the up can be broken up. If all messages are in one giant mailbox database for a large site the time it takes DRS to back it up can exceed the window a site has to get a backup completed. As such they may wish to break message databases up and spread the backup of those databases over multiple nights for instance. There is no performance or capacity advantage, it's all about backup time and size considerations.

To switch the currently attached database, simply use the ChangeActiveDatabase method off the server object using the database name to attach to. The current database name that the server object is attached to can be fetched using the CurrentDatabaseName property:

```
if (!server.ChangeActiveDatabase("UnityRptDb"))
{
    Console.WriteLine("Failed changing databases");
     return;
}
Console.WriteLine("Dbname=" + server.CurrentDatabaseName);
```

Checking Connection Version

One common task you may need to do (I certainly do) is check the version of Unity Connection your application is attached to. Based on which version you may need to expose more/fewer features or fetch different data. The SDK comes with handy methods to make this just about trivial.

When you log into Unity Connection via the SDK it fetches a number of important pieces of data from the server and makes them available to you. Items such as the primary location object identifier for the server, the greetings administrator role id (needed for adding owners to call handlers for instance) and the version information is parsed and broken out into its major/minor/rev/build/es components.

Also included is a handy function that can be used to see if the version of Unity Connection is at least at a specified level or higher. So for instance to check if the Unity Connection server attached is at least 8.6(2) or later you can do that like this:

```
if (server.IsConnectionVersionAtLeast(8, 6, 2,0))
{
    Console.WriteLine("Version is greater than 8.6(2)");

}
```

You can also provide build and ES levels, although the need to know that level of detail is pretty rare.

Checking for Extension Conflicts

Another common task when creating users (or call handlers perhaps) is to check if there's an extension within the partition you are creating the user in that conflicts. So let's see a typical example here. We want to create a new user with extension 1234 in the partition of the default user template – we need to fetch the partition ID and then check for conflict presenting a usable description of the conflicting object to the user (for instance "distribution list 'all subscribers'").

```
string strSql = "SELECT PartitionObjectId from vw_SubscriberTemplate WHERE Alias=?";
string strPartionId = server.GetSingleValueFromQuery(strSql, false, "",
         "voicemailusertemplate");

string strDescription;
```

```
if (server.ExtensionExists("1234", strPartionId, out strDescription))
{
    Console.WriteLine("Conflict:"+strDescription);

}
```

Pretty straight forward – the description string will contain the object's alias/name and its type so the user can present a usable message to the admin or user.

Finding Objects by Name, Extension and/or Alias

The SDK provides several "**Find**xxx" methods off the UnityConnectionServerOdbcSdk class that can save you time when searching for objects. They all work similarly; let's look at finding a user. You can search by alias or extension or both and you can specify if the search is local only (meaning homed on the server you are attached to) or anywhere in the global directory if there is a network of Connection servers involved. A count of matches is returned as an out parameter so you can check if there's more than 1 match found (can happen with extensions when partitions are in use for instance). Again a description is also passed back as an out parameter for easy presentation options.

```
int iCount;
string strDescription;
string strObjectId;

//find by alias, can only be one
if (server.FindUserByAliasOrExtension("jlindborg", "", true, out iCount,
        out strObjectId, out strDescription))
{
    Console.WriteLine("User found, ID="+strObjectId);
}

//find by extension, globally - can be multiple
if (server.FindUserByAliasOrExtension("", "1234", false, out iCount,
        out strObjectId, out strDescription))
{
    if (iCount > 1)
    {
        Console.WriteLine("More than one user found in directory");
    }
    else
    {
        Console.WriteLine("One user found:"+strDescription);
    }
}

//find by both
if (server.FindUserByAliasOrExtension("jlindborg", "1234", true, out iCount,
        out strObjectId, out strDescription))
{
    Console.WriteLine("User found, ID=" + strObjectId);
}
```

Revision History

Version 3.0.2 - 1/8/2014

- First "full" release of SDK functionality