Cisco Unity Connection .NET REST SDK

Contents

Overview

The .NET SDK library for the Unity Connection REST interfaces is a set of library code intended to make development of applications using .NET framework easier, faster and less error prone. Not every method available in every REST interface provided off the Unity Connection platform is included as a "wrapped" functionality in the SDK, but the majority of the commonly needed ones are.

This SDK project is targeted first at wrapping the most commonly used items in the **CUPI For Administrators** interfaces. It also includes **CUTI** (recording using the telephone) and **CUMI** (getting/sending messages), however coverage of those interfaces is far less extensive currently. No coverage of **CUNI** (message event notification framework) is provided at this time. A separate SDK project for covering the **CUPI For Users** interfaces is underway.

Requirements/Special Notes

The .NET REST SDK is written and tested against all versions of Unity Connection 8.6 and later. The REST interface is supported in 8.5 builds but is missing some functionality. The unit tests included in the project are tested against 9.1 and later – running the Unit tests against earlier versions will result in some tests failing – this is expected as functionality was added in later versions that will not exist in earlier builds. The SDK does not do version checking for you, if you call a method that's not supported the server error information that comes back will indicate this.

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. To be clear it simply wraps the existing supported REST APIs provided with the platform – it does not go outside the bounds of those protocols and applications developed using the SDK are just as safe and supported as those written directly to the API.

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… but you can also use Mono for development of applications on Mac or Linux platforms as well as mobile applications on iOS using MonoTouch or Android using Mono For Android. In fact the "Connection CoPilot" iOS application in the iTunes store was developed with the CUPI For Users SDK wrapper library. This is one of the reasons why the library is provided as a source code project instead of only binaries – you must rebuild the source for other platforms.

The library is build and tested using Visual Studio 2010 and Visual Studio 2012 in Windows, however I do validate it works with Mono Develop and Xamarin Studio on Mac.

Installing and Using the Library in Your Project

Currently the library is provided as a source code project you can download via SubVersion to a public read-only repository. You'll want to check for updates frequently as the project is being actively worked on and tested against regularly. To include the library in your project in Visual Studio 2010 or 2012 you need only to add the project file for the ConnectionCUPIFunctions library into your project.

NOTE: Very soon the REST SDK for Unit Connection will be available as a package download off NuGet. This will be very handy since all dependencies are included and updated versions of the SDK will notify you automatically when you bring your project up.

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 "ConnectionCUPIFunctions.csproj" file. This will pull the library into your solution and have it build when you rebuild your project. This will result in the "ConnectionCUPIFunctions.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 ConnectionCUPIFunctions project and add it. Then you only need to add a "using `Cisco.UnityConnection.RestFunctions`;" directive in your project and you're off to the races. The full project includes a couple different project examples such as a simple CLI application, an ASP project for a web based password reset application and a basic WinForms project demonstrating some of the basic capabilities of the library.

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 REST SDK requires the full ".Net Framework 4" setting. Be sure your project is configured for this or you'll get build errors.

You can, if you prefer, simply add the DLL to your project directly instead of including the entire project – it's much nicer for debugging and updating the library over time to keep it as a source project. We make an effort to ensure that changes to the library done over time are backwards compatible and won't break your existing projects but this, of course, cannot be guaranteed indefinitely so it's a good idea to keep the version of the library you're working with separately in case you need to stick with an older version temporarily.

Watching the traffic with Fiddler2

I highly recommend you download and install Fiddler2 on your development machine so you can watch the traffic going to and coming from Connection while you're using the library. This library is not intended to be the end-all-be-all of development against Unity Connection and you may want to extend it or do your own library for specialized functions. The best way to see how Connection's REST interface is working is to watch the HTTP traffic going back and forth and one of the best ways to do that is with Fiddler.

For customers that don't want to use the .NET wrapper library for their projects but want to get a jump start on seeing how commands and requests should be formatted and what they return this can be a very fast way of doing that. It's also a fantastic way to see if your application is being as efficient as it can be or if it's requesting/fetching more data than it needs to for the task at hand – unless you're watching the traffic it can be easy to get sloppy without realizing it. I'll discuss this more later when talking about how and why multiple interfaces are presented for doing essentially the same task (**hint**: one is more work but more efficient on the wire, one is easier and cleaner but results in more traffic).

HTTP Communication Changes in 10.x

With the release of the 10.x platform (which includes Call Manager and Unity Connection products among others) the base server core logic changed for HTTP based communication. In early release testing both internally and externally this has naturally tripped a lot of folks up. The changes are there to make the system less vulnerable to attack and are, on the whole, a good thing.

The very short version is sending basic authentication in every request header you issue is going to bring you to grief. In earlier versions this was allowed and, of course, that makes things very easy since you don't need to fiddle around with cookies, keep track of IDs or timeouts and such, just blindly shove the base 64 encoded authentication string into your header on every request and you're off to the races. Well, play time is over I'm afraid. The good news is if you're using the SDK, it's taken care of for you – no worries, you're welcome. If you're using the SDK as a learning tool and doing your own work, you'll want to pay attention to this and make sure you're not getting burned.

It's really not that difficult, you just need to understand what's going on. When you issue a REST API request the first time you need to send the login name and password in the standard base 64 string (I'll let you Google the details of that if you're doing it by hand for some reason or another). So looking at some Fiddler2 output here for the sequence that looks like this:

Request:

GET https://192.168.0.186:8443/vmrest/users?pageNumber=1&rowsPerPage=50 HTTP/1.1
Cache-Control: no-cache
Authorization: Basic Q0NNQWRtaW5pc3RyYXRvcjplY3NidWxhYg==
Content-Type: application/json
Accept: application/json, */*
Connection: Keep-Alive

Response:

HTTP/1.1 200 OK
Cache-Control: private
Expires: Wed, 31 Dec 1969 16:00:00 PST
Set-Cookie: JSESSIONIDSSO=188767B4B17C1F63B60F9D899C38AD75; Path=/; Secure; HttpOnly
Set-Cookie: JSESSIONID=DFB13585A6F71A70D9CE27A093102938; Path=/vmrest/; Secure; HttpOnly
Set-Cookie: REQUEST_TOKEN_KEY=7208623825002684176; Path=/; Secure; HttpOnly
Content-Type: application/json
Transfer-Encoding: chunked

Note the JSESSIONID and JSESSIONIDSSO values included in the response and the REQUEST_TOKE_KEY. The easiest way to deal with it is to just grab all the cookies and then include them in your requests to the server moving forward. If you send the basic authentication string in your header 3 times in a second, you'll get a 503 back from the server (service not available). Many, many folks claimed to be getting 503s "randomly" who swore they were not doing this. They were. Watch what's happening using Fiddler or WireShark before raising a flag that somehow the server is broken. Spoiler alert: I routinely runs loads for days that send piles and piles of REST requests to servers to check for this and they don't fail. Odds are you're missing something here rather than the platform is behaving "randomly" on you. Trust me.

So, your next request given the above sequence would look like this:

Request:

POST https://192.168.0.186:8443/vmrest/distributionlists HTTP/1.1
Cache-Control: no-cache
Cookie: JSESSIONIDSSO=188767B4B17C1F63B60F9D899C38AD75; Path=/; Secure;
HttpOnly,JSESSIONID=DFB13585A6F71A70D9CE27A093102938; Path=/vmrest/; Secure;
HttpOnly,REQUEST_TOKEN_KEY=7208623825002684176; Path=/; Secure; HttpOnly
Content-Type: application/xml
Accept: application/xml, */*

Response:

HTTP/1.1 201 Created
Set-Cookie: REQUEST_TOKEN_KEY=7208623825002684176; Path=/; Secure; HttpOnly
Location: https://192.168.0.186:8443/vmrest/distributionlists/vmrest/distributionlists/992a3272-4732-4a47-851c-7d733d84eb30
Content-Type: application/xml
Transfer-Encoding: chunked

And the server will be perfectly happy with that in your requests you send. For a while. The request tokens will expire after a time (a few minutes) and you'll start seeing something like this coming back from the server:

Response:

HTTP/1.1 204 No Content
Cache-Control: private
Expires: Wed, 31 Dec 1969 16:00:00 PST
Set-Cookie: REQUEST_TOKEN_KEY=-7208623825002684176; Expires=Thu, 01-Jan-1970 00:00:10 GMT
Set-Cookie: REQUEST_TOKEN_KEY=7972341737335967391; Path=/; Secure; HttpOnly

It's still taking your requests fine but notice that it's told you the token you're using is expiring and it wants you to request another one (by sending a basic authentication with your next request). You can either spot the "Expires" flag for your token in the response and take care of this on the next request or you can simply always send the basic authentication string with your request after a fixed period of time has passed. The SDK uses a simple mechanism of automatically expiring tokens 60 seconds after they've been issued – so if that amount of time has passed since we got our token from the server the next request issued with include the basic authentication string in it, new IDs and tokens will be issued by the server and you can reset your timer for that server. Rinse, lather, repeat. Either way works, the timer mechanism is just easier when dealing with communications to multiple servers out of the same client.

If you ignore the warning that the token is expiring, after a period of time the server with throw up the hand and you'll get a 401 error like this:

Response:

```
HTTP/1.1 401 Unauthorized
Set-Cookie: REQUEST_TOKEN_KEY=-7208623825002684176; Expires=Thu, 01-Jan-1970 00:00:10 GMT
Content-Type: text/html;charset=utf-8
Content-Length: 2186
```

If you see that in your testing you can be pretty sure the logic around requesting a new token/ID set is flawed in some way.

Using the .NET REST 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

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). The first thing to note is that ALL HTTP traffic goes out a single static method that has a monitor construct on entry/exit – this means all Connection servers you may be attached to in your application will all wait in line nicely and issue their HTTP request/response pairs in sequence, not in parallel. This avoid conflicts and "cross talk" issues cleanly, however for very heavy traffic applications wanting to talk to many servers at once, it's not ideal. So if you're looking to build a load test framework for REST targeting many servers at once, you will need to dig into your own HTTP library that will handle parallel traffic patterns.

The other thing the SDK handles for you is authentication and cookie management. If you are talking to, say 3 different Connection servers it will manage the authentication cookies for those servers for you. In Connection 10.0 and later this is critical as sending login/pw for each HTTP request will result in failures due to new security features on the platform. In short the `ConnectionServerRest` object handles this for you and will "flush" the cookie after 1 minute of inactivity to that server.

So, to log into a remote Connection server from your client, you need to create a ConnectionServerRest object and keep it around – this is your "handle" to the server which you will use when sending/receiving data to and from Connection. You will want to authenticate with an account that had the administrator role and, optionally, the mailbox delegate role if you want to be working with messages in other user's mailboxes. The attachment logic is all in the class constructor, there are no static methods off the class so you simply need to create a new instance providing the server name, login and password like this:

```
//attach to server -
ConnectionServerRest connectionServer;

try
{
    //insert your Connection server name/IP address and login information here.
    connectionServer = new ConnectionServer("cuc91.cisco.com", "login", "pw");
}
catch (Exception ex)
{
    //your error handling code here.
}
```

The ConnectrionServerRest instance has a few items off it but the most useful is the version interface that makes it easy to check for which version of Connection you've attached to, like this:

```
//the Connection server class ToString() override spits out the server
 //name and version number in the ToString function.
Console.WriteLine("Attached to Connection server: "+connectionServer);

//do a version check –
if (connectionServer.Version.IsVersionAtLeast(8,6,0,0)==false)
{
```

```
        Console.WriteLine("You are attached to an 8.5 or earlier server!"");
    }
```

At this point you have the details in place for communicating with that Connection server. The remaining samples simply assume the "connectionServer" reference is created and valid so I won't repeat this code chunk over and over again.

The WebCallResult Class

Throughout the library you will see most methods will return an instance of the WebCallResult instead of a simple integer or a bool. There is a method to the madness here – the WebCallResult 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.

Further, the class will hold the ObjectId of an object you just created for you as well as providing pre parsed XML node details from the response if it's included. If you fetched a list of users, for instance, with a paging directive (which we'll cover) the total number of objects returned by the fetch is included in the class properties as well so you can easily do a "7 of 24" type list navigation interface for your users. In short this class can make your life one heck of a lot easier for only a little extra overhead on your part. It's worth it, I promise. For instance, this code snippet creates a new user and the results are returned as a WebCallResult instance:

```
WebCallResult res;

res=UserBase.AddUser(connectionServer, "template", "Alias", "8001",null);
if (res.Success==false)
    {
        Logger.Log("Error! Failed creating new user:"+res.ToString());
        return;
    }

Console.WriteLine("User created, new ObjectId="+ res.ReturnedObjectID);
```

The "ToString()" override for the class includes all details of the request, response, error code etc… in one shot – makes logging out failures a simple process. Notice also that on a success the ObjectId of the newly created user was included - you could have called the version of the AddUser method that returned an instance of the UserFull class all filled in with this (and much more) but that involves another fetch to the server and if you're trying to quickly add users that slows things down – nicer to just do a create call and fetch back the ObjectId of the newly created user and move on. More on that later.

To give you an idea of what logging with the WebCallResult class looks like the following is a dump of the contents of the class after a failed call to fetch a user by alias (the user did not exist):

```
WebCallResults contents:
URL Sent: https://192.168.0.186:8443/vmrest/users?query=(Alias%20is%20missinguser)
Method Sent: GET
Body Sent:
Success returned: False
Status returned 200:Ok
Error Text: No user found with alias=missinguser or objectId=
Raw Response Text: {"@total":"0"}
Total object count: 0
Status description: OK
```

You can see the full URI called, the method type and what the server returned. Although the HTTP response is 200 (OK) the call is a failure so the Success flag is false (as it should be). Normally the contents of this class is all you need to log to get an idea of any failed call that may have been made. There are other properties the class exposes that are not shown here that we'll be seeing and using in examples later on.

The UnityConnectionRestException

While most items in the SDK are easiest to use via their static methods that return object via out parameters and a WebCallResult class discussed above, you can also create new instances of objects as you normally would with any class. A few items must be created this way as they don't have static methods to leverage (for instance the Cluster class). The SDK uses a subclass of the Exception class so that it can return an instance of the WebCallResult class from a thrown exception. You aren't required to use it – a description of the error will appear in the standard Exception instance but it's best to always try and catch the UnityConnectionRestException so you can get the handy information in the WebCallResult structure.

Here's an example showing how to fetch a UserBase class instance (which you can explore more fully in the Users section below) using the "old style" class creation pattern:

```
UserBase oMyUser;
try
{
    oMyUser = new UserBase(connectionServer, "", "jlindborg");
}
catch (UnityConnectionRestException ex)
{
    Console.WriteLine("Failed fetching user:"+ex.WebCallResult);
}
catch (Exception ex)
{
    Console.WriteLine("Failed fetching user:"+ex);
}
```

As a rule it's good style to catch your more specific exception types first and fall back to the base Exception class - in this case it could throw an ArgumentException (i.e. if you passed a null ConnectrionServerRest in) or more commonly the UntiyConnecitonRestException class for any kind of failure at the HTTP/REST API layer. The WebCallResult class will have lots of useful information and can tell you if it was a case of not finding the alias or an HTTP timeout instance or any number of other potential problems with enough detail for you to track down the problem report reasonably easily. The base Exception class will only have a basic explanatory string; none of the other details will be reflected.

As noted you should be using the static calls off the classes when provided as it's much less code and a lot cleaner to read, but if you find yourself wanting to kick it old school, you can.

Death to Magic Numbers!

Throughout the SDK an effort has been made to use ENUM values for any integer (and string) property that is tied to a finite number of options. If you've invested many hours of your life memorizing the various property conventions for the Unity Connection object model I apologize, but for the rest of humanity this should make dealing with the object model considerably easier.

It's left as an exercise for the reader which of these two code chunks is easier to read:

```
        oRule.State = RoutingRuleState.Active;
        oRule.CallType = RoutingRuleCallType.Both;
        oRule.RouteAction = RoutintRuleActionType.Goto;
        oRule.RouteTargetConversation = ConversationNames.SubSignIn;
```

Versus:

```
        oRule.State = 1;
        oRule.CallType = 2;
        oRule.RouteAction = 2;
        oRule.RouteTargetConversation = "SubSignIn";
```

One exception to this is in language codes. There is a helper value for this and you'll see this pattern in the code samples:

```
        (int) LanguageCodes.EnglishUnitedStates
```

Languages are assigned to objects based on the integer value (i.e. 1033 for US English) – to keep your code readable you can use the above type of cast.

Updating Properties on Objects With Dirty Lists

Throughout the SDK you'll find classes that let you update properties on them directly and then execute an "Update" method to send all the pending changes to the Unity Connection server in a single command. For many types of application this provides an ideal mechanism for allowing users to change multiple properties, discard those changes or doing batch updates quickly and easily. Leveraging the strongly typed properties off these classes also allows for a much nicer experience finding what you wish to update and legal values for those properties. Considerably easier than using the REST API directly, which is sort of the point of providing the SDK in the first place.

That said it's important to understand what's happening under the covers. When you update a property that property and its new value are placed into a "dirty value" queue on the class itself. The property value in the class itself is NOT updated and neither is the Unity Connection server. No changes are actually applied until the Update method is called which writes the data back to the server. Further, even after you call Update() by default it will not update the local values you're working with unless you pass "true" as an optional parameter to the Update method. If you pass true then after the update to the server has succeeded it will turn around and re-fetch the data for that object from the server again. Since this involves another round trip HTTP request to the server it is not done automatically. If you were doing bulk updates, for instance, you may not want that extra overhead.

So for instance, let's look at the process of updating some CallHandler properties:

```
//fetch a call handler to edit
CallHandler oCallHandler;
var res = CallHandler.GetCallHandler(out oCallHandler, connectionServer, "",
        "Test Handler");

if (res.Success == false)
{
    Console.WriteLine("Could not find call handler");
    return;
}

//update some properties - call handlers have many
oCallHandler.DisplayName = "My new name";
oCallHandler.EnablePrependDigits = true;
oCallHandler.PrependDigits = "44";
oCallHandler.SendPrivateMsg = ModeYesNoAsk.Ask;

//will output "Test Handler" here.
Console.WriteLine(oCallHandler.DisplayName);

//apply the changes asking for a re fetch of data, too
res = oCallHandler.Update(true);
if (res.Success == false)
{
    Console.WriteLine("Failed updating call handler");
    return;
}

Console.WriteLine("Call handler updated");

//will output "My new name" here.
Console.WriteLine(oCallHandler.DisplayName);
```

Note that if you had not passed "true" to the Update method in the example above the call to write out the DisplayName property would still have produced "Test Handler" in the example above.

You can also clear pending changes and display or log any pending changes prior to calling an update, like this:

```
//update some properties
oCallHandler.OneKeyDelay = 1200;
oCallHandler.Language = (int)LanguageCodes.Japanese;

//dump all pending changes to the console - prints out as a simple
//name value pair list
Console.WriteLine(oCallHandler.ChangeList.ToString());

//flush any pending changes
oCallHandler.ClearPendingChanges();
```

Most of the heavily used classes for users, call handlers, interview handlers, name lookup handlers, distribution lists, contacts, greetings, menu entries etc... implement the above convention. There are, of course, ways to update properties using more traditional name value pair constructions and simple static methods that are more efficient for bulk update type

scenarios which do not involve loading objects and working with properties directly, but for typical applications the above construction is considerably easier to develop against.

Handling Clusters and Connectivity

One common problem developers have to wrestle with is when a server they are communicating with suddenly goes off line for whatever reason. If the system is setup in a cluster configuration (meaning it has a partner that can take over for it when it goes off line) conceptually it's just a matter of redirecting your calls to the partner server and picking up where you left off. Given REST is a stateless protocol you can simply check the failures you get back to see if it's a timeout (meaning the server did not respond).

One of the things the ConnectrionServerRest class gets you when you attach to a server is a list of VmsServer objects exposed as a generic list property you can use to determine if the server is part of a cluster or not. If there's more than one server in the list then you are part of a cluster – simple. You can create a 2nd instance of a ConnectrionServerRest object for the 2nd server that you can have on "hot standby" in your application easily enough using something like this:

```
//after logging in the connectionServer instance you created will have a list
//of servers that will be either 1 or 2 members.  If it has 2 that means there's
//a cluster mate involved - you can try and create a 2nd ConnectrionServerRest
instance
//up front you can use in the case of a timeout event.
ConnectrionServerRest connectionServerSecondary;
if (connectionServer.VmsServers.Count > 1)
{
    foreach (var oServer in connectionServer.VmsServers)
    {
        //Find the server that's not the one you're already connected to!
        if (oServer.ObjectId != connectionServer.VmsServerObjectId)
        {
            try
            {
                connectionServerSecondary = new ConnectionServer(oServer.HostName,
                            connectionServer.LoginName, connectionServer.LoginPw);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Failed to create secondary server:"+ex);
            }
        }
    }
}
```

Having that instance around you can then "fall back" to it easily enough. There are, of course, slicker ways to go about this employing a single ConnecitonServer reference used throughout your code that smartly "swaps out" for the primary or secondary server as necessary but this shows the general idea. Notably the WebCallResult class is key here in that the error code of -1 is reserved to mean specifically a timeout situation.

```
//you've attached to Connection and have been doing various items when we
//get to this call that will fail.
UserFull oNewUser;
res = UserBase.AddUser(connectionServer, "voicemailusertemplate", "newuseralias",
                        "33192", null, out oNewUser);
if (res.Success)
{
    return true;
}

if (res.StatusCode != -1)
{
    //normal error, log and bail
    Console.WriteLine("Error creating user:"+res);
    return false;
```

```
}

//if we're here the status code was -1 which indiates a timeout.  You can have a
//second ConnectrionServerRest instance for the secondary server already on standby
Console.WriteLine("Primary not responding, trying secondary server");

res = UserBase.AddUser(connectionServerSecondary, "voicemailusertemplate",
                       "newuseralias","33192", null, out oNewUser);
if (res.Success)
{
    return true;
}

Console.WriteLine("Failed creating user on secondary:"+res);
```

That's the general idea anyway. As noted you can get much slicker with the handling of this in your application which is why automatically handling is not done at the SDK level. There's simply too many ways you may want to handle the situation to have it be completely automatic under the covers.

As a side note, if you need to test this in your application, the easiest way to do this is by using the command line in the Unity Connection console to turn the network adapter on and off. Setup a cluster, attach to one of the servers, after you log in set a breakpoint in your application and then execute this command on the server you are connected to:

```
set network status eth0 down
```

The next command you issue via REST will result in a timeout – the WebCallResult class will indicate this with a StatusCode of -1 as noted above. To turn the network interface back on simply issue the same command as above but use "up" instead of "down" for the command.

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 WebCallResult class discussed above. Since the SDK can be (and is) used in a variety of application types such as desktop applications, web servers and mobile application 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 ConnectrionServerRest 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 all calls to static methods return a WebCallResult class instance that has all the error and details of what was sent/received from the server that you'd need. Similarly class constructors return a UnityConnectionRestException which contains a handle to a WebCallResult instance used for the same purpose. So the need to hook the error events off the server class is reasonably limited with one exception: JSON serialization errors.

When data comes in from Connection's REST methods, the SDK is taking that text in the body and "deserializing" it into an instance of a strongly typed object. So when a bunch of Json text for a user comes flying across from the server you get a nice handle to a UserBase class object out of it which is much easier to work with. Part of that process involves taking a property in that mess of Json text and stuffing it into a property on the class. If there's no property off the class to fit that data, we have a problem. This can happen if a new property is added on a Connection version that is not accounted for in the SDK – this piece of data is essentially "dropped on the floor" and the SDK carries on as best it can. In most cases this is not the end of the world, however knowing about this is handy (especially for me) so all my applications I used for testing and using the SDK all have a couple chunks of code in them that look like this after I've created and logged into my server creating a ConnectrionServerRest object (called "_server" here):

```
_server.ErrorEvents += ServerOnErrorEvents;
```

Then the definition for the method that fires when the error event is raised simply 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, including any serialization of JSON/XML data coming from the server (or vice versa) 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. Every item being sent to and handled from the Connection server will be dumped out this diagnostic event and you'll slow down your application and fill up logs very quickly if you're not careful. In most cases it's actually easier to just use Fiddler for monitoring traffic and watching what's going on since the debug data will be showing nearly identical information but in a less useful format. However if you need to see what's going on with a customer's system or the like and that's not an option, 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.

Users

Users are, of course, the primary object of interest in most voice message applications. They are tied to a daunting number of other objects in the directory such as a primary call handler, phone system, class of service, alternate extensions, notification devices, private lists etc…easily the largest and most complex object as far as data schema and relationships in the Unity Connection platform. With that in mind we've designed the SDK to simplify finding and using these relationships and data items as possible.

Notably you'll find many "lazy fetch" references hanging off the User class. For instance the "AlternateExtensions()" method will fetch a generic list of all the alternate extensions (both user and system) for a user. The next time you call that same method it will return that pre constructed list for you again without fetching it from the server unless you pass "true" as a parameter to force it to reread the directory data. Yes, you could just create your own list using static calls off the AlternateExtension class which I cover here, but there's generally no need for that extra work. You'll see this same design pattern for many of the related objects for users – I recommend you leverage them instead of reinventing the wheel.

NOTE: these "lazy fetch" items are implemented as methods instead of properties so that they do not fire when, say, a generic list of User objects is bound to a grid for instance. Also, it provides a mechanism for being able to force a refetch of the data from the directory even if it's already been fetched previously.

Creating and Deleting Users

The first thing to note is that the User class is actually TWO classes. A **UserBase** and a **UserFull** class that inherits the functionality of UserBase and adds more properties. The reason for this is that the number of properties on the User class is enormous and Connection's REST interface presents a "short form" version when presenting lists of users and the "full version" when you fetch a single user by ObjectId. This burns a lot of developers that don't realize this and miss the fact that numerous properties they're getting when finding a user by name are not there. This is why. So as a rule you use UserBase when fetching/iterating over lists of users and when you want to fully interrogate them, you get their UserFull instance. The UserBase represents by far the most commonly needed items so having to fetch the full user should not always be necessary. Sounds a little confusing but really it's not – we try and hide much of that complexity from you in the SDK.

The next thing to understand is that most of the class libraries provided have multiple ways to go about creating new objects or finding existing objects. I didn't do this just for fun, there's reasons to use each of them which I'll discuss here while covering creating a new user.

Method 1: Static method with object returned.

```
//The user template name passed here is the default one created by setup and should
//be present on any system.
UserFull oUser;
res = UserBase.AddUser(connectionServer,"voicemailusertemplate","TestUserAlias",
  "8001",null,out oUser);
```

```
if (res.Success==false)
{
    Console.WriteLine("Failed creating new user:"+res.ToString());
    return;
}

Console.WriteLine("User created="+oUser.ToString());
```

Couple things to notice here: Yes, the static method is off UserBase even though the user details returned are UserFull. Since UserFull is derived from UserBase (which defines the AddUser method) Visual Studio will bark that you should be using the base class instead of the derived one. It's just a warning and things will work fine but I like my applications to build as warning free as possible so I stick to the base class reference here.

The one thing to realize here is that this will make a POST to create the new user and, if it succeeds, it will then fetch the entire set of user details with a follow on GET call using the newly created ObjectId for the user and fill in the UserFull instance for you and hand it back. You can then make updates to user properties and sub objects easily using that instance. Convenient if that's what you want but you need to be aware that follow-on GET isn't cheap – watching the transaction in Fiddler2 helps make this clear – there's a lot of data moving back from the server to fill in the UserFull details so if you don't need that object (for instance you're simply making a string of users quickly) you'll want to use the 2nd method.

Method 2: Static method with no object returned:

```
res = UserBase.AddUser(connectionServer,"voicemailusertemplate","TestAlias","8001",
  null);

if (res.Success==false)
{
    Console.WriteLine("Failed creating new user:"+res.ToString());
    return;
}

Console.WriteLine("User created="+ res.ReturnedObjectId);
```

Notice that the code samples are nearly identical other than the lack of a UserFull object being created. This is true, however there's another element here. The last "null" parameter there is a mechanism by which you can pass a series of name value pairs for user properties such that you can create the user with more custom data values than are allowed in the static method call alone which accepts only the bare minimum for creating a new user. If you are developing an application that will be adding bulk users as quickly and efficiently as possible you will want to use this so you can provide things like display names, first/last names, billing IDs etc… up front. This will be much faster than creating users, getting objects back, updating those properties on the objects and then saving them (which we'll cover in the update section for users). The following example shows what that would look like:

```
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("DisplayName", "Jeff Lindborg");
oProps.Add("FirstName", "Jeff");
oProps.Add("LastName", "Lindborg");
oProps.Add("BillingId","7714");
oProps.Add("ListInDirectory", true);
oProps.Add("PromptSpeed", 200);

res = UserBase.AddUser(connectionServer,"voicemailusertemplate","TestUserAlias",
  "80001",oProps);

if (res.Success == false)
{
    Console.WriteLine("Failed creating new user:" + res.ToString());
    return;
}

Console.WriteLine("User created, ObjectId=" + res.ReturnedObjectId);
```

Couple things to note here: The ConnectionPropertyList class is just a simple name value pair construction that has some nice syntactic sugar for handling different types for you (notice the overloads taking strings, bools and integers – this also works with date values). The names of the properties **ARE case sensitive** here. All property names consistently follow the standard "camel hump" construction – first letter is in capital then the first letter of each word in the identifier is capitalized. Check the UserFull instance using Visual Studio's auto complete function for a reference if you're unsure.

This operation is done in one single HTTP request – there is no subsequent fetch of the user details needed here, so for bulk add operations this method is considerably more efficient.

Finding and Fetching Single Users

Fetching single users in the system is easy provided you know their ObjectId (not likely) or their alias (more likely). If you don't know either of those properties you will need to get a list of users back using search criteria (see the next section). The UserBase class has a static method called "GetUser" that you can use to fill in either a UserBase or a UserFull class with information about the user given their alias. The reason the alias is the only item supported in this construction is because the alias is unique across all users in the directory. The primary extension is not (same extension can appear in multiple partitions) which makes a "single fetch" construction tricky – I relegate such searches to a construction that can return multiple matches and let you sort out which one you want.

The code construction for a single fetch is very easy, but filling out the UserBase or UserFull is very important. The code for doing both kinds of fetches looks like this:

```
UserFull oUserFull;
UserBase oUserBase;

res=UserBase.GetUser(out oUserFull, connectionServer, "", "jlindborg");
if (res.Success)
{
    Console.WriteLine("User found:"+oUserFull.ToString());
}
Console.WriteLine("User not found with that alias.");


res=UserBase.GetUser(out oUserBase, connectionServer, "", "jlindborg");
if (res.Success)
{
    Console.WriteLine("User found:" + oUserBase.ToString());
}
Console.WriteLine("User not found with that alias.");
```

Not much to it. You can use either the oUserBase or oUserFull for updating properties on the user, finding its primary call handler, switch assignment etc… the oUserBase simply has fewer properties (the more common ones) than the full list found in oUserFull. However, if you can get away with sticking to the common items in the UserBase definition, do so. What you don't see in the simple code above is what's going on in the background to get that data.

In the case of filling out the UserBase class a single HTTP GET request is made:
https://cuc91:8443/vmrest/users?query=(Alias%20is%20jlindborg)

And a single response is received with roughly 37 lines of data (I use the term "lines" here liberally as obviously the line breaks are not sent, but you get the idea). By comparison when passing the UserFull as an out parameter, the same GET request is sent and the same 37 lines are received back, but then another GET request is made:
https://cuc91:8443/vmrest/users/b083a973-c2a4-4373-aae9-34678ab08d32

And another response is processed, but this time with roughly 150 lines of data. In short roughly 200 lines of data get shuffled across the wire spread over two HTTP requests instead of about 40 with one. Always keep this in mind when developing your applications and decide "Do I really need ALL the user data for this?" and try to "stay skinny" when you can.

Finding and Fetching Lists of Users

Currently the querying capabilities offered by Connection's REST interface are rather limited. Notably compound queries are not supported – so you can't construct a query that says "all users that are in COS=a, primary extension starts with 123 and have first names that start with J". You get one clause to filter by. Please don't shoot the messenger. That said, normally this is enough to work with given the ease of sorting/filtering lists of objects in .NET's generic list classes. It would be nice to do more complex filtering on the server side and not dragging that extra response text over the wire, but such is life.

The list fetching methods for all class objects help you manage paging through lists of objects as well – as a rule you generally want to limit how many objects you fetch at one time – Connection will throttle you if you make requests that take too many cycles to fulfill if you are not economical in your requests when Connection is busy. Clearly it can't let you impact its call processing capabilities simply because you don't want to deal with handling paging. If you fetch 1000 users at a crack and you notice your application "randomly" failing during high traffic times, this is likely your issue. When in doubt, limit yourself to no more than 100 items at a time.

First, how not to do this:

```
//this will get all users on the system.  Don't do this unless you're just testing.
res = UserBase.GetUsers(connectionServer, out oUsers);
```

and now how to use paging like you should:

```
//get just the count of users, the oUsers list is empty and there's very little data
//dragged across the wire – the page number=0 is a special command here telling
//Connection that you don't want actual user data, but just the count of users.
res = UserBase.GetUsers(connectionServer, out oUsers, 0);
Console.WriteLine("Total users="+res.TotalObjectCount);

//fetch (up to) the first 5 users in the list
res = UserBase.GetUsers(connectionServer,out oUsers, 1, 5);
if (res.Success == false)
{
    Console.WriteLine("Error fetching users:"+res);
    return;
}

//The WebFetchResults class has another benefit: total objects count is included.
//In my server's case the values here are "26" and "5" respectively.
Console.WriteLine("Total objects on server:"+ res.TotalObjectCount);
Console.WriteLine("Objects returned:"+oUsers.Count);

//fetch the 2nd page of results and so on.
res = UserBase.GetUsers(connectionServer, out oUsers, 2, 5);
if (res.Success == false)
{
    Console.WriteLine("Error fetching users:" + res);
    return;
}

//fetching an invalid page number does not result in an error - it simply returns
//an empty list of users.
res = UserBase.GetUsers(connectionServer, out oUsers, 30, 5);
if (res.Success == false)
{
    Console.WriteLine("Error fetching users:" + res);
    return;
}

//The output here is "26" and "0"
Console.WriteLine("Total objects on server:" + res.TotalObjectCount);
Console.WriteLine("Objects returned:" + oUsers.Count);
```

So in short you can fetch the total object count from the server either stand alone (using a page number of 0) or simply use the first batch of users you fetch to get the value – do a little math and you can see how many pages you have to iterate over to provide a nice "6 of 26" type list presentation to your users or the like. There's a simple example of how to do such a presentation in the WinForms example in the "CUPIFastStart" project if you're interested.

Filtering and sorting, as noted, is rather limited. Only a single clause is allowed an only "is" and "startsWith" are supported – so no "Contains", "Between" or other types of similar operators you may be used to. Also note that when sorting and filtering you can only sort on the same clause you filter on – so you can't filter by extension and sort by alias in other words. You can leverage .NET list sorting for lists of manageable size – a UserSort clause is provided for that use which is shown here:

```
//to sort generic lists of users (either base or full) you can use the UserComparer
//class.  Here is how you can rearrange a list of users to sort ascending by their
//primary extension
UserComparer oCompareer = new UserComparer(UserComparer.UserSortElements.
  DtmfAccessId.ToString());

oUsers.Sort(oCompareer);

//...and by first name
oCompareer = new UserComparer(UserComparer.UserSortElements.FirstName.ToString());
oUsers.Sort(oCompareer);
```

You can sort by Alias, FirstName, LastName, DisplayName, DtmfAccessId (primary extension). Empty strings are sorted later than non empty strings (does not apply to Alias or DtmfAccessId since those cannot be blank).

On with the filtering and server side sorting story. The query and sort clauses can be added to the end of the GetUsers call along with the paging parameters – the SDK takes care of adding these to the URI parameters including escaping out spaces and such. Here are a couple examples of filtering and sorting

```
//simple fetch to get all users named "Jeff" - neither the name itself or the
//query parameters are case sensitive.
res = UserBase.GetUsers(connectionServer, out oUsers, "query=(firstname is jeff)");

//This gets all users that have primary extensions starting with 2
//with proper paging options included - the GET clauses can just keep getting
//stacked in.
res = UserBase.GetUsers(connectionServer, out oUsers,1, 5,
"query=(DtmfAccessId startswith 2)","sort=(DtmfAccessId asc)";
```

If you need to construct a very large list of users on the local client for whatever reason it's best to create a local object such as a Dictionary (.NET has a nice selection of containers to choose from) and use paging to "fill up" your container with users one page full at a time. A good rule of thumb is to stick to 100 objects at a time when fetching against production servers. In your test environments certainly more can be returned (with a somewhat long delay) but it's not a good idea. To keep the server from throttling your application and your users from twiddling their thumbs wondering if your application is locked up it's a better idea to fill up your local container in pages and provide a nice update status to your user (i.e. "Loading user details, %40") – since you get the total number of users in the first page full you'll know where you're at in the process. On the whole this is the preferred approach to handling large lists of objects in general, but users in particular since they are so large by comparison to other objects in the directory.

Updating Users

As with creating users there are a couple ways to update users in the SDK. The first way leverages the static method for updating users and passing in a property list similar to the option for creating new users and populating the list of user properties you wish to have applied to the user up front. This requires you know the objectId of the user – for instance if you had a list of userObjectIds fetch into a list or that were referenced (for instance as members of a public distribution list) or the like, this method would make sense in scenarios where you'd be bulk applying some properties to all users in that list. Something along these lines:

```
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("VoiceNameRequired",true);
oProps.Add("ListInDirectory", false);
oProps.Add("IsVmEnrolled", false);

foreach (string strObjectId in oListOfUserObjectIds)
{
    res = UserBase.UpdateUser(connectionServer, strObjectId, oProps);
    if (res.Success == false)
    {
        Console.WriteLine("User update failed:"+res);
        break;
    }
}
```

Notice that the above code only issues a series of POST commands, one for each user – it never has to do a GET to fill any data. Again, for processing similar changes for large groups of users this will be much more efficient than creating user objects first and then updating them.

Another method leveraging the User class instances instead is to simply make changes to properties off that instance and call the "Update" method directly on that instance. This is not as efficient for large lists of users but is considerably easier to use and maintain and can be very handy in building user interfaces where you can bind form elements directly to properties of a class as it can save you a bunch of time having to code up a "dirty list" of changed properties.

```
//First, fetch a user - we'll grab one by alias here
res = UserBase.GetUser(out oUserBase, connectionServer, "", "jlindborg");
if (res.Success == false)
{
    Console.WriteLine("Error fetching user:"+res);
    return;
}

//update as many properties as you like
oUserBase.ListInDirectory = false;
oUserBase.DisplayName = "New User Display Name";
oUserBase.VoiceNameRequired = true;
oUserBase.IsVmEnrolled = false;

//Apply the changes to the server.  Only changed properties are sent.
res = oUserBase.Update();

if (res.Success == false)
{
    Console.WriteLine("Error updating user:"+res);
    return;
}
```

The second method is handy for a couple reasons. First, you can make changes right off the instance of the class which makes finding the property you want as easy as searching the InteliSense list Visual Studio provides. This is also handy if you have the user presented with many UI elements they can change on an object – use binding to show the values of the User object, for instance, and allow them to make a single change when they're all done. If they change their mind the class includes a "ClearPendingChanges()" method that flushes the "dirty" property list for you.

Voice Names

Dealing with voice names and greetings for users and call handlers in the REST interfaces in Connection has easily been the item that's caused developers the most grief. With that in mind we've made an effort to make the stream file handling in the SDK as painless as possible. So to set the right tone here, let's see an example of how to set a voice name of a user from a WAV file on the local hard drive. This sample assumes you've already fetched a user instance (either base or full, doesn't matter):

```
res=oUser.SetVoiceName(@"c:\myvoicename.wav", true);
if (res.Success == false)
{
    Console.WriteLine("Failed updating voice name:"+res);
    return;
}
```

That's all there is to it. Note the "true" parameter on there – this is a very powerful capability in the SDK that will rip your WAV file into a raw PCM 16/8/1 recording for you before uploading the WAV file to Connection. If you've worked with Connection's REST interface at all you know it can be pretty fussy about the WAV format and you'll see a lot of "invalid media format" errors coming back. This eliminates that bit of pain which can save you a bunch of time. You're welcome.

And conversely, how do we fetch a voice name from a user on Connection to a local WAV file on my client? Looks almost identical:

```
res = oUser.GetVoiceName(@"c:\voicenameout.wav");
if (res.Success == false)
{
    Console.Write("Failed fetching voice name:"+res);
```

```
        return;
    }
```

OK, so that covers using WAV files from the local client for voice names – not so scary. What if you want to use the telephone as a recording device instead? This is referred to in Connection as "TRAP" (Telephone Record and Playback). Instead of using your microphone and recording a local file and uploading it, you record a stream file directly on the Connection server via your telephone and then assign that recorded stream file to a user's voice name. In the REST API pantheon this is referred to as "CUTI" for the "Telephone Interface" API. The SDK provides a PhoneRecording class to help you with this interface.

This is pretty easy – you need a phone to use and in our example here we'll assume it's 1003. It's assumed that the Connection server you're attached to can dial it directly, of course. Again, in this example our "oUser" object has already been fetched.

```
//First, establish a call to extension 1003 (your extension presumably)
//This is done in the construction of the class instance.
PhoneRecording oPhone;
try
{
    oPhone = new PhoneRecording(connectionServer, "1003");
}
catch (Exception Ex)
{
    Console.WriteLine("Failed to establish phone call:"+Ex);
    return;
}


//now start recording - you'll only hear a beep here that indicates
//you can start talking - press # and the recording will terminate
res=oPhone.RecordStreamFile();
if (res.Success == false)
{
    Console.WriteLine("Error recording stream:"+res);
    return;
}

//for fun, play the currently recorded stream out
res = oPhone.PlayStreamFile();


//Now set that stream as the voice name for your user
res= oUserBase.SetVoiceNameToStreamFile(oPhone.RecordingResourceId);

if (res.Success == false)
{
    Console.WriteLine("Error setting voice name :" + res);
}

//hang up the phone
oPhone.Dispose();
```

Wow. That was pretty easy, right? You'll see this class come up again when we talk about greetings for call handlers later.

PINs and Passwords

Another pain point for folks using the CUPI interface in the field has been managing PIN and Passwords. For clarity "PIN" is a Personal Identification Number and is a password you enter via the phone (numbers only). A Password is alphanumeric string used for logging into GUI clients (i.e. with access to a full keyboard interface). The ResetUserPin and ResetUserPasswords methods are provided as both static and instance methods off the user class (either full or base, it doesn't matter).

```
//use the static class if you have the objectID and don't need to create
//a user object first.
```

```
//Reset the Pin of a user that you have the ObjectId for.
res = UserBase.ResetUserPin(connectionServer, strObjectId, "1324523");

//More commonly you're leveraging a user object to update PINs and passwords.
//just reset the password and nothing else
res = oUser.ResetPassword("RainySunday");

//clear the hacked count and unlock the user's account
//Passing a blank PIN means it will be skipped - you cannot set a PIN or a
//Password to blank via the SDK.
//The null values mean leave their corresponding values alone
res = oUser.ResetPin("", false, null ,null ,null , true);

//reset the PIN and unlock the account.
res = oUser.ResetPin("019012", false);
```

It's sometimes also desirable to check the PIN and Password settings for a user. Are they locked out (hacked), is it set to never expire, when was it changed last? This is offered through the Credential class – each user has a PIN and a Password credential associated with them. As usual there is a static method to go about this and a convenient "lazy fetch" option off a user instance. Both are shown here:

```
//use the static class if you have the objectID and don't need to create
//a user object first. Resetting PINs for large numbers of users would be
//quicker using this method for instance.

//Fetch the credentials for a user's PIN
Credential oCredential;
res = Credential.GetCredential(connectionServer,strObjectId,CredentialType.PIN,
  out oCredential);
Console.WriteLine("PIN hacked="+ oCredential.Hacked);

//As usual you can also fetch this same information off the user instance using
//a lazy fetch mechanism
Console.WriteLine("PIN hacked="+ oUser.Pin().Hacked);

//or for the password
Console.WriteLine("PW last changed="+ oUser.Password().TimeChanged);
```

Private Lists

Each user can have up to 99 private lists (limit is configurable in the user's Class of Service). A private list can contain users, public distribution lists, remote contacts and other private lists. The SDK allow for creating of new lists, recording a voice name for that list, adding and removing members from lists and, of course, reviewing membership and list details. Note that the SDK currently only supports adding users and public lists and private list members. Adding private lists to other private lists is a little strange and isn't plumbed into the library.

The easiest item is reviewing the list details for a user. As you've probably come to expect, there's a lazy fetch method for private lists off the User instance that you can leverage to quickly review the list information and its membership. Here's an example of showing all the lists and each list's membership details for a user you've already fetched:

```
//Dump all private lists and membership details for all private lists
//associated with a user.
foreach (var oList in oUser.PrivateLists())
{
    Console.WriteLine(oList.ToString());
    Console.WriteLine("Memebers:");

    foreach (var oMember in oList.PrivateListMembers())
    {
        Console.WriteLine("    "+oMember.ToString());
    }
}
```

The “ToString” override on the private list object shows the lists name and its number (lists are addressed using their number, from 1 to 99). The “ToString” override on the list member object shows its type (local user, distribution list or private list), their alias and display name.

Of course you can also fetch lists and membership information via static calls as shown here:

```
//fetch all the private lists for a user via static call
List<PrivateList> oLists;
res = PrivateList.GetPrivateLists(connectionServer, strUserObjectId,out oLists);

//fetch members of a list via a static call - need the ObjectId of the user that
//owns the list and the objectId of the list itself for this.
List<PrivateListMember> oMemebers;
res = PrivateListMember.GetPrivateListMembers(connectionServer,strListObjectId,strUs
erObjectId,out oMemebers);
```

To create new lists you use static methods to create a new instance of a list object which you can then edit in much the same way you can users and other objects in the directory. In this example we create a new private list for a user that we’ve already created a User object for. Note that when you create a new list you have to assign a number to it – this is that id between 1 and 99 (depending on your maximum count allowed). You need to assign the ID to one that’s not in use – it’s not required that you add them sequentially (i.e. you can have list 1, 7 and 19 if you like) but this makes managing private lists via the API difficult since the user may be making private lists via the web interfaces and you have no control over how “orderly” they are about it. If you add a list that already exists the server will return an error. Similarly if you add a list beyond the number supported you will also get an error. Be sure to check the return values for the WebCallResults at each step (which I’m not doing here for brevity).

```
//get the count of lists - we'll add one after that number
PrivateList oPrivateList;
int iListCount = oUser.PrivateLists().Count;

res = PrivateList.AddPrivateList(connectionServer,oUser.ObjectId,"My new list",
  iListCount+1,out oPrivateList);

if (res.Success == false)
{
    Console.WriteLine("Error adding private list:"+res);
    return;
}

//set the voice name of the private list - heard durring the message
//addressing conversation when the list is selected as a message target.
res=oPrivateList.SetVoiceName(@"c:\myVoiceName.wav");

//Find a public list in the directory and add it to our new list
DistributionList oPublicDl;

res=DistributionList.GetDistributionList(out oPublicDl,connectionServer,"",
  "allvoicemailusers");

res = oPrivateList.AddMemberPublicList(oPublicDl.ObjectId);

//Find a user in the directory and add it to our new list
UserBase oUser;
res = UserBase.GetUser(out oUser, connectionServer, "", "jlindborg");

res = oPrivateList.AddMemberUser(oUser.ObjectId);
```

To remove a member you need to fetch the ObjectId of the PrivateDistributionListMember object first. This sample shows how to fetch a specific private list by number off a user, remove a member from that list and then delete the list entirely:

```
//fetch list #3 for our user (the one we added above).
```

```
res = PrivateList.GetPrivateList(out oPrivateList, connectionServer, oUser.ObjectId,
 "", 3);
if (res.Success == false)
{
    Console.WriteLine("Failed fetching list:" + res);
    return;
}

//get a particular member from the list (the user jlindborg) and remove them.
//There is no individual member select opition, you need to get the list and
//find the member you want here;
foreach (var oMember in oPrivateList.PrivateListMembers())
{
    if (oMember.MemberType == PrivateListMemberType.LocalUser &
        oMember.Alias.Equals("jlindborg"))
    {
        res =oPrivateList.RemoveMember(oMember.ObjectId);
        break;
    }
}

//if you want to iterate over the list of members again be sure to use the
//"refetchData" flag to make sure the list is refetched and you're not using
//stale data from an earlier fetch.
oPrivateList.PrivateListMembers(true);

//now delete the list
res=oPrivateList.Delete();
```

Alternate Extensions

Each user in the directory has a primary extension which shows up on their user object instance. This is not optional and every user has one. There can, however, be alternate extensions (both user added and admin added) totaling up to 20 numbers. These are normally for home phones, cell phones and such that allow the user to be automatically logged in when they call to check voice mail from one of those lines or for callers that forward into the Connection voice mail system from one of those numbers to be forwarded to the user's greeting. The SDK has a set of methods for fetching, reviewing and managing these extensions for you.

We'll start with listing the alternate extensions for a user that we've already fetched into an oUser object – as you've come to expect by now this leverages a simple lazy fetch method to get the alternate extensions list off the user directly:

```
List<AlternateExtension> oAlternateExtensions;

//output all alternate extensions - what's returned will depend on the user's COS
//settings - admin added alternate extensions may or may not be included.
foreach (var oTempExt in oUser.AlternateExtensions())
{
    Console.WriteLine(oTempExt.ToString());
}
```

To add and remove an alternate extension uses similar design patterns we've already seen – use the static method to create a new instance as shown here:

```
//Adding an alternate extension can be restricted by the user's class of service so expect
//that this call can fail.
AlternateExtension oAltExt;
res = AlternateExtension.AddAlternateExtension(_connectionServer, oUser.ObjectId, 3,
"1234",out oAltExt);

if (res.Success)
{
   Console.WriteLine(oAltExt.DumpAllProps());
```

```
        //delete the alternate extension you just added.
        res = oAltExt.Delete();

    }
```

The “DumpAllProps” there is just a handy debug mechanism that drops all the values of any instance of just about all the classes in the SDK – it exports every property in the class and it’s corresponding value in a simple name/value pair table format for easy review.

Notice, again, the index number there (3 in this example) – that’s the position of the alternate extension, it’s up to you to pick the right one to add (one that does not exist) – it may be necessary to iterate over them all to find an open one since the user can certainly create/remove them via the web interface and you cannot assume a contiguous progression of IDs in the list.

Notification Devices

In Connection 9.0 and later there are 6 default notification devices that are associated with all users and that cannot be deleted. Four of these 6 are available via the phone interface to edit (the phone based devices as opposed to the SMTP and HTTP notification devices). Users and administrator can add additional notification devices of any type – there is technically no limit to these. This makes dealing with the administration of these devices a little tricky at times – you can count on those 6 devices (5 in versions prior to 9.0 since HTTP did not exist) but beyond that you have no idea how many may be there and there are no defined “slots” for how many additional devices can be created. So you’ll be doing a lot of walking of lists in other words – and fortunately generic lists in .NET are easy to work with in this regard.

As usual we’ll start with listing the notification devices out – since devices are always tied to users it’s easiest to do this off the User object – you can use the static NotifcationDevice class and fetch them that way, of course, but you’ll need to pass in the objectId of a user to get them (i.e. you cannot iterate over all notification devices in a system, it must be done by user).

```
        //list all devices for a user - this includes their name, type and if they're active
        foreach (var oDevice in oUser.NotificationDevices())
        {
            Console.WriteLine(oDevice.ToString());
        }
```

Not too tricky. You can also fetch a specific notification device by name like this

```
        //fetch a specific device for a user by name
        res = oUser.GetNotificationDevice("Home Phone", out oNotificationDevice);
        if (res.Success == false)
        {
            Console.WriteLine("Failed to find home phone device:" + res);
            return;
        }
```

You can also create different types of devices – the SDK supports creating a phone, pager, SMS, SMTP or HTML based devices. Here’s a bit of code that shows how to create a new phone based device, fetch it, update it and then delete it.

```
        //create a new phone device for a user
        NotificationDevice oNotificationDevice;
        res = NotificationDevice.AddPhoneDevice(connectionServer, oUser.ObjectId, "New Phone
            Device",oUser.MediaSwitchObjectId, "5551234","NewUrgentVoiceMail",true,
            out oNotificationDevice);

        if (res.Success == false)
        {
            Console.WriteLine("Failed creating phone device:"+res);
            return;
        }

        //Update some of the notification device details.
        oNotificationDevice.DisplayName = "New Display Name";
```

```
oNotificationDevice.EventList = "AllMessage";
oNotificationDevice.Active = false;

//apply changes
res = oNotificationDevice.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update new device:"+res);
}

//delete the device we just added
res = oNotificationDevice.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed to delete device:" + res);
}
```

Couple of things to notice about this code sample. First, when creating a new notification device you must always provide a user's ObjectId, of course, but also a phone system association (media switch objectID) – this is true for all phone and pager based devices since they require a dial out which needs to know which switch definition to go out on. Also notice the event trigger string – this is a comma separated string that you can pass multiple types of messages that will cause the device to trigger. These include and of: AllMessage,NewFax,NewUrgentFax,NewVoiceMail,NewUrgentVoiceMail,DispatchMessage,UrgentDispatchMessage. Although strange, it won't hurt anything to include, say "AllMessage" and "NewVoiceMail" in the same list even though they are redundant.

Creating an HTML based device (added in Connection 9.0) requires you pass in a Notification Template ID as part of the creation. To this end there is a NotificationTemplate class provided that makes fetching and enumerating the list of templates defined on the system easy for you. The following chunk of code shows how to fetch the list of templates and then create a new HTML notification device for a user:

```
//First, fetch the HTML templates on the server - there are two by default.
//Here we only check if at least one is returned and we blindly use it for brevity.
List<NotificationTemplate> oTemplates;
res = NotificationTemplate.GetNotificationTemplates(connectionServer,
  out oTemplates);

if (res.Success == false || oTemplates.Count<1)
{
    Console.WriteLine("Failed fetching templates:"+ res);
    return;
}

//Now add the device for our user - the display name needs to be unique here
res = NotificationDevice.AddHtmlDevice(connectionServer, oUserTestDude.ObjectId,
  oTemplates[0].NotificationTemplateID, "New HTTP","testguy@test.com",
  "NewVoiceMail", true);

if (res.Success == false)
{
    Console.WriteLine("Failed adding device:"+ res);
    return;
}

//you can turn around and fetch the device by name if you like.  Clearly it's easier
//to just use an out parameter on the AddHtmlDevice above but if you need to fetch
//by name you can use this technique
NotificationDevice oTest;
res = NotificationDevice.GetNotificationDeivce(connectionServer,
  oUserTestDude.ObjectId, "", "New HTTP", out oTest);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching device"+res);
}
```

A special note to consider when it comes to notification devices about scheduling. By default any new device created will be associated with the default system schedule (all hours). You can assign any notification device to any schedule in the system by assigning a schedule set's ObjectId to the notification devices "ScheduleSetObjectId" property and updating it (see the Schedules section for more). If, however, you wish to assign a custom schedule to a device, the SDK greatly simplifies this rather daunting task for typical scenarios. If you want to create a simple schedule that's active for selected days from the same time to the same time each of those days, you can do this in one line of code (a long line admittedly). If you wish to create more complex schedules with different start/stop times or with "breaks" in the middle of days etc… you'll need to look at the Schedules section for details on how to do that.
In this example we'll update a notification device that's already been loaded to have a new, custom schedule that's active Monday, Wednesday and Friday from 10am to 9pm:

```
//Create a new custom schedule in one call using AddQuickSchedule.
res = ScheduleSet.AddQuickSchedule(connectionServer, "Phone Device Schedule", "",
  oUser.ObjectId, Schedule.GetMinutesFromTimeParts(10, 0),
  Schedule.GetMinutesFromTimeParts(21, 0),
  true, false, true, false, true, false, false);

if (res.Success == false)
{
    Console.WriteLine("Failed creating schedule:"+res);
    return;
}

//Assign the newly created schedule to the notification device and update
oDevice.ScheduleSetObjectId = res.ReturnedObjectId;

res = oDevice.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed assigning schedule:"+res);
}
```

At this point the notification device that was created earlier is now associated with a custom schedule. To be clear this schedule is stored in the same location as all system schedules but because it's assigned to a user instead of a location (see the Schedules section for more details here) it will not show up in the CUCA web admin interface in the schedules section. As a side note, the "Schedule.GetMinutesFromTimeParts" call is just a simple helper to convert an hour/minute in 24 hour format into minutes-from-midnight which is what the schedule interface needs for start and stop times – you could replace that with just an integer if you wanted to do your own math there.

Message Waiting Indicators

Users can have up to 10 MWI devices defined – although it's very rare to ever see more than 2 other than for some edge case scenarios. With that in mind the SDK does not support finding MWIs by name or the like – you can add, delete and fetch MWIs for users but to find the one you want you'll need to iterate through them. As usual we'll start with a simple example of listing the MWIs for a user that's been fetched already

```
//dump out all MWIs defined for a user - includes the name, extension if it's
//Enabled and if the lamp is on or not.
foreach (var oMwi in oUser.Mwis())
{
    Console.WriteLine(oMwi.ToString());
}
```

And now an example of creating a new MWI device, fetching it, updating it and finally deleting it:

```
//Create a new MWI device and set it active
res = Mwi.AddMwi(connectionServer, oUser.ObjectId, "New MWI", oUser.MediaSwitchObjec
tId, "1234", true);
if (res.Success == false)
```

```
{
    Console.WriteLine("Failed adding MWI:"+res);
    return;
}

//fetch the new MWI using the objectId returned on the
//WebCallResults class instance.
Mwi oMwiDevice;
res = Mwi.GetMwiDevice(connectionServer, oUser.ObjectId, res.ReturnedObjectId,
  out oMwiDevice);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch new MWI:"+res);
    return;
}

//update some properties on the Mwi device we just added and save it
oMwiDevice.DisplayName = "New Display Name";
oMwiDevice.IncludeTextMessages = true;
oMwiDevice.MwiExtension = "4321";
oMwiDevice.Active = false;

res = oMwiDevice.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update MWI:"+res);
    return;
}

//finally, delete the device we just added
res=oMwiDevice.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed to delete MWI:"+res);
}
```

Mailbox Information

The MailboxInfo class provides information about the size, mounted state, quota limits and if the deleted items folder is enabled for the user. There's only one way to get this data and that's to create a new instance of the MailboxInfo class using the ObjectId of a user – there are not static methods or "lazy fetch" options off the user for this:

```
//Fetch the mailbox information for a user
MailboxInfo oMailboxInfo;
try
{
    oMailboxInfo = new MailboxInfo(connectionServer, oUser.ObjectId);
}
catch (Exception ex)
{
    Console.WriteLine("Failed fetching mailbox info:"+ex);
    return;
}

Console.WriteLine("Over send limit="+oMailboxInfo.IsSendQuotaExceeded);
Console.WriteLine("Mailbox size={0}, send limit={1}, recieve limit={2}",
    oMailboxInfo.CurrentSizeInBytes, oMailboxInfo.SendQuota,
    oMailboxInfo.ReceiveQuota);
```

The class also contains a helper function to fetch the message counts (in addition to the full inbox size in byes provided in the attributes).

```
int iInbox, iSent, iDeleted;
res = oMailboxInfo.GetFolderMessageCounts(out iInbox, out iDeleted, out iSent);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch folder counts:"+res);
    return;
}

Console.WriteLine("Inbox message count="+iInbox);
```

This is a read only informational class, there's no updating capabilities included.

Class of Service

Every user in the system is associated with one and only one Class of Service that dictates system access in the admin, which phone numbers the user is allowed to enter and limits access to licensed features in Connection. The SDK allows for finding, fetching, reviewing, creating and deleting classes of service. For more on working with Class of Service objects, see the Class of Service section.

As usual let's start with simple fetches of lists of class of services:

```
//get and list all COSes
List<ClassOfService> oCoses;
res =ClassOfService.GetClassesOfService(connectionServer, out oCoses);

foreach (var oCos in oCoses)
{
    Console.WriteLine(oCos.ToString());
}


//fetch all COSes that have a display name that starts with "voice"
res = ClassOfService.GetClassesOfService(connectionServer, out oCoses,
  "query=(DisplayName startswith voice)");

foreach (var oCos in oCoses)
{
    Console.WriteLine(oCos.ToString());
}
```

The query clause construction is the same as was presented in the user searching section and can include a sort clause and paging. Normally with class of service objects it's not likely there will be so many that paging is a concern – if you have thousands of class of service objects in your design, you're doing it wrong. That said you can certainly construct your application to use such features, they work in the same way here.

There's also a lazy fetch handle for class of service references for users that can be used off an instance of the UserBase or UserFull class:

```
Console.WriteLine("Can send to DL-" + oUser.Cos().CanSendToPublicDl);
```

And you can also find a class of service using its name if you need to. The display names for classes of service must be unique system wide so you can use them for this purpose:

```
ClassOfService oCos;
res = ClassOfService.GetClassOfService(out oCos,connectionServer,"",
  "Voice Mail User COS");
```

Greetings

Greetings for users are managed through the users primary call handler. See the Greetings section for Call Handlers for details on how to use this in the SDK. As with many references on the user you can leverage a simple "lazy fetch" path to get the call handler for a user in one operation:

```
// show the greeting name, play what setting and active/inactive setting
// for all 7 greetings associated with a user.
foreach (var oGreeting in oUser.PrimaryCallHandler().GetGreetings())
{
    Console.WriteLine(oGreeting.ToString());
}
```

Transfer Options

Similar to greetings, the 3 transfer rules for a user are stored on the primary call handler associated with the user. See the Transfer Options section for Call Handlers for details on how to use this in the SDK. Similar to other references, there is a "lazy fetch" option for transfer rules off the primary call handler fetch on the user which can be used like this:

```
//Dumps the transfer details (including active/inactive and action setting)
//for all 3 transfer options associated with a user.
foreach (var oTransfer in oUser.PrimaryCallHandler().GetTransferOptions())
{
    Console.WriteLine(oTransfer.ToString());
}
```

Menu Entries

Similar to greetings and transfer options, the menu entries (mappings for actions taken when callers press 0-9, * or # during the user's greeting) are associated with the user's primary call handler. See the Menu Entries section for Call Handlers for details on how to use this in the SDK. This can be accessed off a user object (either full or base) using a lazy fetch reference to the user's primary call handler like this:

```
//Dumps the basic configuration for each of the 12 keys in the user's menu
//entries.
foreach (var oMenu in oUserFull.PrimaryCallHandler().GetMenuEntries())
{
    Console.WriteLine(oMenu.ToString());
}
```

Phone System

All objects that can handle calls (users, call handlers, interviewers, name lookup handlers) and many sub objects such as notification devices are assigned to phone systems. Connection can have numerous different phone systems defined at the same time and assigning users and phone notification devices and such to different phone system is how administrators can segment, say, tenants that are sharing a single Connection installation and/or dictate which ports notification dial outs for particular users will be used.

Currently there is no option to create new phone systems or port groups (this is coming in a later version of Connection) so the Phone System is informational (read only). As usual you can list all phone systems via a static method call or leverage simple "lazy fetch" calls off objects like the user to review the details of the phone system they are associated with.

```
//get all phone systems
List<PhoneSystem> oPhoneSystems;
res = PhoneSystem.GetPhoneSystems(connectionServer, out oPhoneSystems);

foreach (var oPhoneSystem in oPhoneSystems)
{
    Console.WriteLine(oPhoneSystem.ToString());
    Console.WriteLine("Details:");
    Console.WriteLine(oPhoneSystem.DumpAllProps("--->"));
}
```

```
//Get the phone system details for a particular user
Console.WriteLine(oUser.PhoneSystem().ToString());
```

Exiting the User Mailbox Conversation

So, where does the user go when they hit * to exit out of their inbox conversation? Certainly you can customize the conversation to simply not allow that, and some sites do, but normally the users can exit out and by default they end up going to the Opening Greeting call handler created by setup – mostly because they have to go somewhere and that's as good a place as any. Like many other settings in Connection, the user's exit destination is controlled by the "Action trinity" that lets you set an action, conversation and destination so that you can define the exit action to be anything from hang up immediately (very rude but efficient), launch a special conversation or send the call to another user, call handler, interview handler or name lookup handler. Lots of options and you control it all by editing three properties on the FullUser object:

- ExitAction
- ExitTargetConversation
- ExitTargetHandlerObjectId

This is the same 3 property construction used in many places in Connection to do the same thing – route the call somewhere based on an action. Anything from where to go when a menu entry is pressed to where to go after a greeting finishes playing to how to handle a caller not entering anything in a name lookup handler. Rather than repeat the details of how to use these three properties here and everywhere else I've included a general discussion on how to use the "Action" properties as these are referred to in a consolidated Setting Actions section.

LDAP Users

The UserLdap class is provided to allow you to find and import users from an LDAP integration. So for instance if you have Active Directory LDAP synchronization configured for Unity Connection, the users available for import are provided as part of the static calls to "GetLdapUsers" off this class. Only those users available for import from LDAP are presented via this interface and once you import a user they will no longer be returned in that search. If you then turn around and delete that user from Unity Connection's directory they will once again show up as available for import (or should – there have been a few bugs in this area in some versions of Unity Connection).

So what's going on here? Behind the scenes there's another database you cannot access directly via REST or ODBC that is responsible for holding all the user's found in your LDAP configuration(s) for a server. This table maintains a mapping of users from the LDAP integration to Unity Connection users in the directory. When you pull a list of users to import they are pulled from this database behind the scenes, not from the LDAP directory itself – the synching of that database is done separately in the background. When a user is imported the ObjectId of that user in Connection is added to the table in that background database which indicates this user is "mapped" and should not be offered for import again. When you delete that user the mapping should be removed, thus making it available again.

All that really isn't important for getting your job done, however, so lets just look at a typical example of something you'd want to do in an application. Find a user to import and import them. The UserLdap class contains only a few methods off of it since you can't create/delete/edit LDAP users, of course, only find and import them. LDAP users have only a few properties on them: first and last name, alias and pkid (unique ID from that mapping table mentioned above). Notably they do not have extensions populated on them so you have to provide that when importing them along with the alias of a template, similar to creating a new user in the previous section.

Here's a code chunk showing finding a user to import with the alias "jlindborg" and then importing them:

```
//Find the user
List<UserLdap> oLdapUsers;
res = UserLdap.GetLdapUsers(connectionServer, out oLdapUsers, 1, 20,"query=(alias is
            jlindborg)");

if (res.Success == false)
{
    Console.WriteLine("Failed fetching LDAP user:"+res);
    return;
}

if (oLdapUsers.Count != 1)
{
    Console.WriteLine("Failed finding jlindborg");
    return;
}
```

```
//Import the user
UserFull oImportedUser;
res = oLdapUsers[0].Import("3388192345", "voicemailusertemplate",out oImportedUser);

if (res.Success == false)
{
    Console.WriteLine("Failed importing user:"+res);
    return;
}

Console.WriteLine("User imported:"+oImportedUser.ToString());
```

You can, of course, also use the static method to import a new imported user if you happen to know the pkid and alias of the user to import – this is pretty unusual though, it's much easier to find the user as we did above and them import them off the instance method which has the alias, first/last name and pkid values already populated for you and you only have to provide the extension and the alias template.

Note that the option to pass back out the FullUser object on the import used in the above sample is optional there. If you're just importing users en mass and don't need to get a user object back out, you can do that and save that round trip HTTP fetch needed to fill in the user properties.

As a side node – users created via import from an LDAP source will behave like any other user you create directly. Specifically you can edit their first/last names and such and the API (and by extension the SDK) will happily allow for that. However when the LDAP sync takes place next time any values "owned" by the LDAP integration will be overwritten. Just be aware that the API does not prevent you from editing these fields on local users. You can tell a user is LDAP synched by checking the **LdapCcmPkid** property on the UserFull instance – it will not be empty in the case of an LDAP integrated user.

One last note, be careful about alias conflicts. The alias from LDAP may very well conflict with another user in Connection's database and the import will fail. Be sure to check your error reasons during import, particularly if you are working with a mix of locally created users and users imported from an LDAP provider.

Messages

To access messages for either review or sending on behalf of other users the account you log into Unity Connection with must have the "Mailbox Access Delegate Account" role assigned to it. If you attempt to send a message from a user or review messages for a user with an account that does not have this role, Connection will return "authentication required" errors back.

Fetching Messages for a User

First, let's look at fetching messages for a user – this is similar to fetching users from the directory but has a bit more sugar added to help with filtering, sorting and paging.

```
//Fetch the unread and urgent messages sorted by the oldest first.
//Gets the first 20 messages in the stack that match this criteria
res = UserMessage.GetMessages(connectionServer, oUser.ObjectId, out oMessages,1,20,
  MessageSortOrder.OLDEST_FIRST, MessageFilter.Read_False |
  MessageFilter.Priority_Urgent);
```

The first thing to notice is the paging parameters are passed as simple integers – 1 and 20 in this case. So the page number (0 gets count, 1 is the first page) is first, followed by the number to fetch per page. Just like with items in the directory to total count of messages returned is stored in the WebCallResult class passed back so you can setup "x of y" type paging structures in your UIs easily. If you do not pass these parameters the SDK defaults to 1 and 10 respectively.

Also note that you can set the sort order to sort by oldest first, newest first or urgent first (regardless of read state). The default sort order if nothing is passed in here is newest messages first. The filter options can be compounded by separating with me "or" operators – in the example above it's fetching urgent unread messages. There are a dozen different filters you can apply alone or together. By default it uses the "none" filter to get all messages of all states and types.

There's also the ability to get information about individual messages and fetch attachments (usually WAV files of course) off of them. There's static and instance versions of these but usually you're creating a UserMessage class instance and working with that. This example shows fetching the first 10 messages from a user's inbox, outputting the top level details of each message along with the attachment count and then fetching the first attachment off each message that has at least one and dumping it to a local hard drive location:

```
//Fetch the messages for a user - the default for this method is to fetch
//the first page of results 10 at a time if you pass no additional parameters.
List<UserMessage> oMessages;
res=UserMessage.GetMessages(connectionServer, oUser.ObjectId, out oMessages);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching messages:"+res);
    return;
}

//iterate over all messages fetched returning top level information, attachment
//count and then save the first attachment (if present) into c:\temp\ using
//the GUID of the message.
foreach (var oMessage in oMessages)
{
    Console.WriteLine(oMessage.ToString());

    //get how many attachments are present in the message - usually 1
    int iCount;
    oMessage.GetMessageAttachmentCount(out iCount);

    Console.WriteLine("    Attachment count="+iCount);

    //Extract the first message attachment to a local file
    if (iCount > 0)
    {
        //note that the MsgId includes a "0:xxx" tacked onto the beginning which
        //is used for forwarding scenarios - Windows does not like colons in paths
        //so if you want to use this construction you must remove it
        string sFile = string.Format(@"c:\{0}.wav",oMessage.MsgId.Replace(":",""));
        res = oMessage.GetMessageAttachment(sFile , 0);

        if (res.Success == false)
        {
            Console.WriteLine("Failed saving attachment:"+res);
        }
    }
}
```

Most of the time you will be dealing with the inbox messages which is why the SDK defaults to that folder - however you can fetch messages from the deleted items folder and/or the sent items folder as well. The following example shows fetching the first 10 deleted items messages and restoring the first one to the inbox as an unread message - this scenario might be encountered if a user accidentally deleted a message they didn't mean to for instance. Remember that the user's class of service may be configured to not stored deleted messages in the deleted items folder - in which case the folder will always be empty.

```
//get deleted messages for user
List<UserMessage> oMessages;
res = UserMessage.GetMessages(oServer, oUser.ObjectId, out oMessages, 1, 10,
                              MessageSortOrder.NEWEST_FIRST, MessageFilter.None,
                              MailboxFolder.DeletedItems);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch deleted messages:"+res);
    return;
}

if (oMessages.Count < 1)
{
    Console.WriteLine("No deleted messages found for user");
```

```
    return;
}

UserMessage oMessage = oMessages.First();

//mark the message unread.
oMessage.Read = false;
res=oMessage.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update read status of message:"+res);
    return;
}

//restore it to the inbox folder
res = oMessage.Restore();
if (res.Success == false)
{
    Console.WriteLine("Failed to restore deleted message:"+res);
    return;

}
```

Send a New Message Using Wav File

This example shows how to send a new voice message from the mailbox of a user to an SMTP address. The SDK only uses SMTP as destination addresses since this is all that's needed. The API does support other address constructions such as target ObjectId strings accompanied by object type designations, but these are unnecessary in my opinion. All addressable objects in Connection contain a usable SMTPAddress that can be used for addressing and this simplifies the interface quite a bit. The CreateMessage methods both allow for multiple addressing targets that can be either "TO", "CC" or "BCC" address types; however the address itself is always just a plain SMTP string.

All the properties of a message can be included as flags in the call – these include urgent, secure, private, dispatch, read and delivery receipts as well as an optional callerId instance which can be passed as NULL if you don't want to include the caller ID details with the message.

This example flags a message for urgent and private as well as having the "c:\test.wav" file converted into raw PCM before being uploaded as a message attachment.

```
//fetch user with alias of "jlindborg" - we will be sending the message from his
//mailbox.
UserFull oUser;
res = UserBase.GetUser(out oUser, connectionServer, "", "jlindborg");

if (res.Success == false)
{
    Console.WriteLine("Could not find user in database by alias=jlindborg");
    return;
}

//create a recipient - you must include at least one and can include many.
MessageAddress oRecipient = new MessageAddress();
oRecipient.AddressType = MessageAddressType.TO;
oRecipient.SmtpAddress = "jsmith@lindborglabs.com";


//set the message for urgent and private.  Note that the callerId instance can be
//passed as null here if none is desired.
res = UserMessage.CreateMessageLocalWav(connectionServer, oUser.ObjectId,
    "Test Subject", "c:\\test.wav", true, SensitivityType.Private , false,
    false, false, false, null, true, oRecipient);

if (res.Success == false)
{
    Console.WriteLine("Error uploading voice message:" + res);
```

```
    return;
}

Console.WriteLine("Message sent");
```

Send a New Message Using Phone (CUTI)

As with greetings and voice names you can use CUTI for recording voice messages as well. An alternative version of the CreateMessage method is provided to handle this use of the CUTI interface. The advantage here is that you can record a voice message using the phone as a media device and the media never leaves the Unity Connection server itself. It's created there and "turned into" a voice message attachment right where it stands: no need to produce recorded media locally to the client or upload binary files to the server. Particularly for mobile clients or for applications where media security is a concern this can be a real advantage.

In this example we'll record the message using the phone with CUTI and address it to multiple targets – including both a remote user and CC'ing the message to the sending subscriber.

```
//fetch user with alias of "jlindborg" - we will be sending the message from his
//mailbox.
UserFull oUser;
res = UserBase.GetUser(out oUser, connectionServer, "", "jlindborg");

if (res.Success == false)
{
    Console.WriteLine("Could not find user in database by alias=jlindborg");
    return;
}

//create a recipient
MessageAddress oRecipient = new MessageAddress();
oRecipient.AddressType = MessageAddressType.TO;
oRecipient.SmtpAddress = "jsmith@lindborglabs.com";

//CC the message to the sender
MessageAddress oCc = new MessageAddress();
oRecipient.AddressType = MessageAddressType.CC;
oRecipient.SmtpAddress = oUser.SmtpAddress;

//use the CUTI interface to leave a message using extension 1001.
PhoneRecording oPhone;

try
{
    oPhone = new PhoneRecording(connectionServer, "1001");
}

catch (Exception ex)
{
    Console.WriteLine("Failed to connect to phone extension:" + ex);
    return;
}

//record the message itself - ends when the user hits #
res = oPhone.RecordStreamFile();
if (res.Success == false)
{
    Console.WriteLine("Failed recording message:" + res);
    return;
}

//include caller ID details in the message
CallerId oCallerId = new CallerId();
oCallerId.CallerName = "Jeff Lindborg";
```

```
oCallerId.CallerNumber = "2065551234";

//convert the recorded stream into a voice message.
res = UserMessage.CreateMessageResourceId(connectionServer, oUser.ObjectId,
    "my subject", oPhone.RecordingResourceId,false, SensitivityType.Normal,
    false, false, false, false, oCallerId, oRecipient, oCc);


if (res.Success == false)
{
    Console.WriteLine("Failed uploading message:"+res);
    return;
}

//hangup
oPhone.Dispose();


Console.WriteLine("Message sent");
```

Forwarding a Message With Intro

The SDK provide methods off the message object for easily forwarding a message with an introduction. You can provide the recording for the introduction either via a WAV file on the local hard drive or using the CUTI interface to record it on the Connection server and reference it there. You can also forward without an introduction by using the wav file call and passing a blank path instead.

This example grabs the first message from a user's inbox and forwards it to another user with an intro using a wav file recorded on the local hard drive.

```
//get inbox messages for user
List<UserMessage> oMessages;
res = UserMessage.GetMessages(oServer, oUser.ObjectId, out oMessages);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch deleted messages:"+res);
    return;
}

if (oMessages.Count < 1)
{
    Console.WriteLine("No deleted messages found for user");
    return;
}

//construct an address for the recipent to forward to
MessageAddress oAddress= new MessageAddress();
oAddress.AddressType = MessageAddressType.CC;
oAddress.SmtpAddress = "jsmith@testdomain.com";

//notice the special construction for handling urgent (priority) and private
//(sensitivity).  The SDK uses simple Boolean flags for these however the API uses
//more complex string values - use the enums provided for handling that.
UserMessage oMessage = oMessages.First();

res = oMessage.ForwardMessageLocalWav("FW:" + oMessage.Subject,
    oMessage.Priority == PriorityType.Urgent,oMessage.Sensitivity,
    oMessage.Secure, false, false, "c:\\intro.wav", true, oAddress);

if (res.Success == false)
{
    Console.WriteLine("Failed to forward message:"+res);
}
```

Replying to a Message

The SDK provides a way to reply or to reply all to any message. It will allow you to attempt the reply no matter what the senders information, it's up to you to make sure that makes sense. The list of recipients is provided for review off the UserMessage class instance as is the FromSub boolean flag which indicates if the message was sent from a subscriber vs. the outside caller mailbox (which cannot be replied to). The SDK will not stop you from replying to invalid addresses or the like - the mailbox you send the message on behalf of will simply get a NDR message for the failed attempt.

The reply can be constructed using a wav file from the local file on the hard drive or via a stream recorded on the Connection server via CUTI, just as with forwarding or creating a new message. The following chunk of code just does a simply reply to a message - all that's needed for a reply all is to pass the flag as true instead of defaulting to false. In this case we will use the CUTI interface to record a reply using the phone.

```
//get inbox messages for user
List<UserMessage> oMessages;
res = UserMessage.GetMessages(oServer, oUser.ObjectId, out oMessages);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch deleted messages:"+res);
    return;
}

if (oMessages.Count < 1)
{
    Console.WriteLine("No deleted messages found for user");
    return;
}

//just use the first message in the list
UserMessage oMessage = oMessages.First();

//Establish a phone connection
PhoneRecording oPhone;
try
{
    oPhone = new PhoneRecording(oServer, "1002");
}
catch (Exception ex)
{
    Console.WriteLine("Failed to establish phone connection:"+ex);
    return;
}

//now record a reply over the phone
Console.WriteLine("Record your reply, press # to end.");
res = oPhone.RecordStreamFile();

if (res.Success == false)
{
    Console.WriteLine("Failed to record reply:"+res);
    return;
}

//reply to all recipients
res = oMessage.ReplyWithResourceId("RE:" + oMessage.Subject,
    oPhone.RecordingResourceId, oMessage.Priority == PriorityType.Urgent,
    oMessage.Sensitivity,oMessage.Secure, false, false, true);

if (res.Success == false)
{
    Console.WriteLine("Failed to forward message:"+res);
    return;
}
Console.WriteLine("Reply sent!");
```

```
//hang up
oPhone.Dispose();
```

Deleting Messages and the Deleted Items Folder

The UserMessage class has a delete method off it just as many other class definitions do, however the UserMessage version includes a boolean flag that lets you "hard" vs. "soft" delete a message. "Soft" deletes will leave a copy of the message in the deleted items folder provided the user's class of service is configured to do that. A "hard" delete will never put a copy in the deleted items folder regardless of what the class of service is configured to do.

There's also a static method to clear the deleted items folder entirely. The following code chunk shows a message being "hard" deleted and then the deleted items folder being cleared out.

```
//hard delete the message
res = oMessage.Delete(true);
if (res.Success == false)
{
    Console.WriteLine("Failed to delete message:"+res);
}

//clear all messages from the deleted items folder
res =UserMessage.ClearDeletedItemsFolder(oServer, oUser.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed to clear the deleted items folder");
}
```

Helper Methods

Finally there are some helper functions off the UserMessage class that can be useful when dealing with messages – notably the static methods for converting milliseconds from 1970 into local time and vice versa. The details on a message object returned from Connection have the ArrivalTime and ModificationTime stored as longs – this is milliseconds from midnight on 1/1/1970 – it's a common format for storing date/times, particularly in Linux land. You can use the "ConvertFromMillisecondsToTimeDate" method on the static UserMessage class to take that long and convert it either into GMT or into local time on the box you're running on. Similarly you can use the "ConvertFromTimeDateToMilliseconds() to go in the other direction.

User Templates

[Editing/adding requires Unity Connection 10.0 or later]

When creating a new user you are required to provide a user template's alias to do it. Users and their related sub objects constitute hundreds of individual properties that span many sub collections and templates allow administrators to setup default behavior and settings for many, many properties up front. The idea that you could manage all of them on your own with each create is, of course, highly unrealistic.

A user template is very much like a user but has somewhat fewer properties. Anything that is only filled in for a specific user instance such as their extension or their SMTP address for instance are not present on the user template. Outside of that they behave very similar to a user including their sub objects such as their primary call handler with associated transfer and greeting rules etc. As such I'm not going to cover all that again for templates; just the basics and you can review the User section for more details on how the object model hangs together for users as a whole.

Finding and Fetching User Templates

You can always count on at least one user template to be in the system since the installer creates one and marks it undeletable. This is the "voiceMailUserTemplate" alias and should always be available to you. In fact in many of my examples I simply use this rather than fetching the list of available templates and selecting one. Nothing wrong with that but typically you'll want to provide users with a list of templates to choose from when creating a new user. The SDK provides the usual list fetching methods accompanied by easy paging capabilities if there are very large numbers of templates on your system. This example simply selects the first 20 (default count limit) and dumps the top level information for each out to the console:

```
List<UserTemplate> oTemplates;
res = UserTemplate.GetUserTemplates(_server, out oTemplates);
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch templates:"+res);
    return;
}

foreach (var oTemplate in oTemplates)
{
    Console.WriteLine(oTemplates.ToString());
}
```

You can, of course, include filter/sort clauses as optional parameters to the GetUserTemplates method as you can with most other GetXXX calls in the SDK. Typically for templates such filtering is not necessary; however it's provided for completeness and consistency.

This example shows how to fetch a specific template by alias instead of fetching them all as a list:

```
UserTemplate oTemplate;
res = UserTemplate.GetUserTemplate(_server, "", "myTemplate", out oTemplate);

if (res.Success == false)
{
    Console.WriteLine("Failed to find template:"+res);
    return;
}

Console.WriteLine("Template found:"+oTemplate.ToString());
```

Creating, Updating and Deleting User Templates

This example shows how to create a new user template, edit a few properties in it and then turn around and delete it. Not a terribly practical example but you get the idea:

```
UserTemplate oTemplate;

res = UserTemplate.AddUserTemplate(_server, "voicemailusertemplate",
    "myNewTemplate","New Template Name",null, out oTemplate);

if (res.Success == false)
{
    Console.WriteLine("Failed to create template:"+res);
    return;
}

Console.WriteLine("Template created:"+oTemplate.ToString());

oTemplate.DisplayName = "Updated Display Name";
oTemplate.Department = "Engineering";
oTemplate.PromptSpeed = 150;
res = oTemplate.Update();
if (res.Success == false)
{
    Console.WriteLine("Failed to update template:"+res);
    return;
}

Console.WriteLine("Updated template");
```

```
res = oTemplate.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed to delete template:"+res);
    return;
}

Console.WriteLine("Template deleted");
```

One thing to notice here: Yes, when you create a new user template you must provide the alias of an existing user template! In our example above we used the system created “voicemailusertemplate” template since we know it’s always there. That may seem a little circular but again, there is a LOT of user properties to be carting around, and starting with a template of defaults is very handy.

Global Users

The GlobalUser class behaves very much like the UserBase class with respect to finding, fetching and iterating users. The primary difference is you don’t create, edit or delete global users, you can only find them. The global user collection represents a small amount of data for all users across all Connection servers in your network and is used for addressing purposes for the most part. All users created on all Connection servers replicate around to all other Connection servers and show up in the global users collection eventually.

This can be used when addressing messages to other users on the network for validation or for more complex scenarios such as cross server login or the like. For an example on how to use global users and locations to “hop” to different Connection servers in a digital network to get to the home server for remote servers all over your network, see the Locations topic.

System Contacts

System contacts are user objects that do not have mailboxes on Unity Connection and can be used to route calls to destinations and such without using a seat license (they are free). At the time of this writing the REST API does not support deleting these objects, however you can create and edit them. This example just runs through creating a contact, adding a voice name and updating some properties on it:

```
//create a new system contact using the default system contact template
//created by install
Contact oContact;
res=Contact.AddContact(oServer, "systemcontacttemplate", "Test Contact",
                    "Test","Contact", "testcontact", null, out oContact);


if (res.Success == false)
{
    Console.WriteLine("Failed to create new contact:"+res);
    return;
}

//add a voice name to the contact from a wav file
res = oContact.SetVoiceName("c:\\voiceName.wav", true);
if (res.Success == false)
{
    Console.WriteLine("Failed to set contact voice name:"+res);
    return;
}

//update some contact properties
oContact.DisplayName = "New display name";
oContact.ListInDirectory = true;

res = oContact.Update();
if (res.Success == false)
{
    Console.WriteLine("Failed to update contact:"+res);
    return;

}
```

Call Handlers

Call handlers come in two flavors: a “system call handler” which are the ones you see in the CUCA web interface in the system call handlers section such as the “opening greeting” and “say goodbye” handlers. Then there are “primary call handlers” which are tied to a user with a mailbox. These have the same functionality but the primary handlers are exposed in the user pages in the CUCA web interface and not on their own. Much of the call processing functionality is in the call handler which is why many of the tasks you do on a user you are actually doing on their associated primary call handler. Understanding this makes the model much easier to grasp.

The key piece to understand is that fetching call handlers will, by default, include all call handlers - both system and primary - which may not always be what you want. When fetching call handlers it's a good idea to filter by name (or at least by non primary flag - which we'll see here in a minute).

Creating and Deleting Call Handlers

First, let's just create a simple system call handler here. You never "create" a primary call handler, they are created automatically when you create a user with a mailbox (admin users do not have call handlers or mailboxes). One wrinkle with system call handlers is when you create one you must pass in the ObjectId (GUID) for a call handler template as part of the creation. With User creation you can use the alias of the template which gets passed on the URI but that's not allowed for display name fields (call handlers do not have an alias, but display name must be unique). So before creating a new handler you must first fetch the template you wish to use. For more details on managing call handler templates, see the Call Handler Template section.

```
//First, we need to fetch a call handler template to work with.
//This is the default template that should be present on any system.
CallHandlerTemplate oTemplate;
res = CallHandlerTemplate.GetCallHandlerTemplate(out oTemplate, connectionServer,
        "","System Call Handler Template");

if (res.Success == false)
{
    Console.WriteLine("Failed fetching handler template:"+res);
    return;
}

//Create a system call handler with no extension - the name must be
//unique among all handlers on the system or this will fail
CallHandler oHandler;
res = CallHandler.AddCallHandler(connectionServer, oTemplate.ObjectId,
        "My New Handler", "", null,out oHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed creating handler:"+res);
    return;
}

Console.WriteLine(oHandler.ToString());
```

As with most other objects that support creation you can also pass in a set of parameters to establish values for at the time of creation. Coupled with the ability to create a handler without returning an instance of one (thus skipping the follow-on GET call to the server) you can do bulk creation of call handlers much more efficiently if large numbers of them are needed. The following chunk of code shows how to go about that - each create involves a single POST request to the server and a very small response that contains the ObjectId of the newly created handler and that's all. This assumes we've already fetched the template ahead of time.

```
//construct name/value pair class
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("DispatchDelivery",false);
oProps.Add("Language",(int) LanguageCodes.FrenchStandard);
oProps.Add("SendSecureMsg",true);
oProps.Add("MaxMsgLen",60);

//Create a system call handler with no extension - the name must be
//unique among all handlers on the system or this will fail
```

```
CallHandler oHandler;
res = CallHandler.AddCallHandler(connectionServer, oTemplate.ObjectId,
        "New Handler", "", oProps);

if (res.Success)
{
    Console.WriteLine("New call handler ObjectId="+res.ReturnedObjectId);
}
```

Finding and Fetching Call Handlers

First, fetching just a single call handler by name is a common task. As noted the display names for all local call handlers are unique so you can search on their names (not case sensitive) to fetch one like this:

```
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, connectionServer, "",
        "Opening greeting");

if (res.Success == false)
{
    Console.WriteLine("Failed finding handler:"+res);
    return;
}
```

Fetching lists of call handlers works very similar to fetching users - paging support is build in along with the query and sort commands. One wrinkle with call handlers is that it will be returning primary call handlers along with system call handlers in these searches. If you're not filtering by name or DTMFAccessId (extension) you'll want to at least filter using the IsPrimary flag (0 means get only system call handlers which is what you want). And, as ever, always be good boys and girls and leverage paging and keep your fetch size reasonable. Call handlers are not as large as users but it's still a good idea to restrict yourself to no more than 100 at a time when operating in production environments. The following code chunk fetches all system call handlers in the system in 20 handler chunks:

```
List<CallHandler> oHandlers;
int iPageNumber = 1;

do
{
    res = CallHandler.GetCallHandlers(connectionServer, out oHandlers, iPageNumber,
         20,"query=(IsPrimary is 0)");

    if (res.Success == false)
    {
        Console.WriteLine("Failed fetching handlers:" + res);
        break;
    }

    foreach (var oHandler in oHandlers)
    {
        Console.WriteLine(oHandler.ToString());
    }

    iPageNumber++;

} while (oHandlers != null && oHandlers.Count > 0);


Console.WriteLine("Done!");
```

Notice that when the fetch for a page number that results in no results is called it does not kick an error out, but returns an empty list instead. No results is not considered an error condition in this case. Also, technically the check for "oHandlers

!= null" is not necessary as it will never be null unless the GetCallHandlers returns a failure and we exit from that path - but old habits die hard when dealing with class instances...

Updating Call Handlers

There are a couple ways to update call handlers as you probably suspect by now. If you've already got an ObjectId of a handler to work with you can skip the unnecessary GET involved with creating a CallHandler object first and pass in a set of properties in a name/value pair construction. This can work well for bulk update scenarios for instance.

```
//construct name/value pair class
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("DispatchDelivery",false);
oProps.Add("Language",(int) LanguageCodes.FrenchStandard);
oProps.Add("SendSecureMsg",true);
oProps.Add("MaxMsgLen",60);

//Update an existing handler if you've already got an ObjectId
res = CallHandler.UpdateCallHandler(connectionServer,strHandlerObjectId, oProps);

if (res.Success==false)
{
    Console.WriteLine("Failed updating handler:"+res);
}
```

More commonly however you've already fetched a call handler by name or are iterating through a list of constructed handler instances in a generic list or the like. In that case leveraging the convenient "dirty property" feature in the CallHandler class works well.

```
oHandler.DisplayName = "New display name";
oHandler.Language = (int) LanguageCodes.FrenchStandard;
oHandler.SendSecureMsg = true;
oHandler.MaxMsgLen = 60;

res = oHandler.Update();
if (res.Success == false)
{
    Console.WriteLine("Failed updating handler:" + res);
}
```

Voice Names

Voice names for call handlers are not used as commonly as for users but they do pop up here and there. They get played out in the greetings administration conversation (this allows users to change greetings for system call handlers they are listed as owners for via the phone) and they can also be used when identifying messages from outside callers that were left for particular handlers which can be helpful in disambiguating where they came into your mailbox from.

Similar to users the CallHandler class has some helper functions to make updating voice names for call handlers easier. The following code chunk shows fetching a call handler and updating its voice name from a WAV file on the local hard drive and then turning around and fetching the voice name to another local file:

```
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, connectionServer, "",
        "Jeffs Call Handler");

if (res.Success == false)
{
    Console.WriteLine("Failed fetching handler:"+res);
    return;
}

//update the handler's voice name off a wav file - convert to PCM 16/8/1 first.
res = oHandler.SetVoiceName(@"c:\VoiceNameIn.wav", true);
```

```
if (res.Success==false)
{
    Console.WriteLine("Failed updating voice name:"+res);
}

//now fetch the name to a local wav file
res = oHandler.GetVoiceName(@"c:\VoiceNameOut.wav");
if (res.Success == false)
{
    Console.WriteLine("Failed fetching voice name:"+res);
}
```

As shown for users you can also use the phone for updating voice names for call handlers (or greetings) - review the Updating Voice Names for Users section for a description of how to go about that, the same SetVoiceNameToStreamFile() method used for this.

Greetings

There are 7 greetings for all call handlers in the system, which of course applies to users as well since they are tied to a primary call handler. The greetings included in the collection (in order or precedence) are as follows:

Alternate - if active, always plays

Holiday - if active and the handler's schedule is a holiday, it plays.

Internal - if active and the call is from a known user's extension, it plays

Busy - if active and the call forwarded busy to the handler's extension, it plays.

Off Hours - if active and the schedule of the handler indicates off hours, it plays.

Standard - if no other greetings kick in, it plays. You cannot deactivate this greeting.

Error - special greeting, plays when an invalid option is entered during another greeting. It also cannot be deactivated.

By default only the standard greeting is activate and it'll effectively play all the time.

The greetings interaction is in some way simpler than some other objects - you cannot create or delete a greeting - ever. Each handler gets 7 and only 7 and they cannot be removed. So we only need to worry about fetching them, reviewing them and updating them. Easy. First, let's look at fetching the greetings for a call handler - in this case I'll show both the system call handler and how you'd do it from a user object - after this I'll trust that you can figure out you need to go through the user's "PrimaryCallHandler" reference to get to these functions.

```
//get all greetings for a call handler - this should always get all 7 greetings
foreach (var oGreeting in oHandler.GetGreetings())
{
    //outputs the greeting type, if its enabled and what it's set to play
    Console.WriteLine(oGreeting.ToString());
}


//For a user it would look like this - just need to use the "lazy fetch" primary
//call handler reference off the user object.
foreach (var oGreeting in oUser.PrimaryCallHandler().GetGreetings())
{
    //outputs the greeting type, if its enabled and what it's set to play
    Console.WriteLine(oGreeting.ToString());
}
```

...and to fetch just a specific greeting if you don't want to iterate through them to find the one you want, it would look like this:

```
res = oHandler.GetGreeting(GreetingTypes.Busy, out oGreeting);
if (res.Success == false)
{
    Console.WriteLine("Failed fetching greeting:" + res);
```

```
}
```

Ok, now on to editing greetings. There's two big things you want to do with greetings and dealing with the API for this is very error prone and annoying. The SDK wraps most of the complexity for you and whisks you to a happy place of programming efficiency so you can get on with your life.

The first thing is, of course, updating the greeting recording itself (i.e. what plays to caller when they hear this greeting). One thing to understand, however, is that each greeting can be recorded in separate languages. So you can have a French and English busy greeting that play to callers depending on which language their call it set to. This is very helpful in constructing multiple language auto attendant trees since you only have to make one tree and just record different language greetings for each to handle all those languages. However it's up to you to make sure the language you're uploading is actually installed on the Connection server. Yes, that's correct - you can upload greetings and set languages for them that are not installed on Connection and it'll happily let you do that. And they'll never play. So, when updating greetings in other languages, be sure to review the helpful InstalledLanguage class section for details on how to make sure you're not wasting your time.

So, let's update the busy greeting for a call handler and apply an English and Japanese language greeting to it to show how that's done. I'll assume the check that Connection has the languages installed has already taken place here.

```
//update the recording for the busy greeting and set it to play that custom greeting
//(instead of the system generated greeting or blank for the greeting) in both US
//English and Japanese.
res = oGreeting.SetGreetingWavFile((int)LanguageCodes.EnglishUnitedStates,
        @"WAVFiles\ENUBusyGreeting.wav", true);

if (res.Success == false)
{
    Console.WriteLine("Error setting ENU Busy greeting recording: " +
        res.ToString());
}

res = oGreeting.SetGreetingWavFile((int)LanguageCodes.Japanese,
        @"WAVFiles\JPNBusyGreeting.wav", true);

if (res.Success == false)
{
    Console.WriteLine("Error setting JPN Busy greeting recording: " +
        res.ToString());
}

oGreeting.PlayWhat = PlayWhatTypes.RecordedGreeting;
res = oGreeting.Update();
```

Don't forget that last step - by default the greeting will play the SystemGreeting which is the canned construction using system prompts - you need to instruct the greeting to play your custom recorded message instead by setting the PlayWhat to the RecordedGreeting type.

You can also use the phone to record a greeting, similar to what was shown for recording a user's voice name earlier. For greetings it would look like this:

```
//use telephone as media device - establish a connection to extension 1003
PhoneRecording oPhone;
try
{
    oPhone = new PhoneRecording(connectionServer, "1003");
}
catch (Exception ex)
{
    Console.WriteLine("Failed establishing phone call:"+ex);
    return;
}
//record a new stream
res = oPhone.RecordStreamFile();

//play the stream we just recorded for confirmation
```

```
res = oPhone.PlayStreamFile();

//set it's US English recording to the recording we just made
res = oGreeting.SetGreetingRecordingToStreamFile(oPhone.RecordingResourceId,
        oGreeting.CallHandlerObjectId,oGreeting.GreetingType,
        (int)LanguageCodes.EnglishUnitedStates);
```

The next thing you want to update for greetings is if they're active or not. Remember that you can't make the standard or error greetings inactive, so don't bother trying (the server will return an invalid operation error if you do). But if you wish to, say, activate an alternate greeting, how does one go about that? It's not a simple on/off switch because greetings are built to offer an expire-time. So, for instance, you can set the alternate greeting to be active only till next Friday at 5pm and then go inactive. As such the active/inactive state is done through a time/date construction which can be a little twitchy to deal with.

This example shows how to do the simple approach of just enabling a greeting forever and how to make it expire at a particular date instead:

```
//use the helper function to enable the greeting.  This call sets it to be enabled
forever.
res = oGreeting.UpdateGreetingEnabledStatus(true);

if (res.Success == false)
{
    Console.WriteLine("failed enabling busy greeting:" + res);
}

//to instead make it enabled only for a set time, use this construction
//Set the greeting active till 2/23/2013 at 10:30 pm.
//Important - times are stored in UTC, be sure to convert to UTC instead of passing
your
//system's local time!
DateTime oDateToExpire = new DateTime(2013, 2, 23, 22, 30, 00).ToUniversalTime();

res = oGreeting.UpdateGreetingEnabledStatus(true,oDateToExpire);

if (res.Success == false)
{
    Console.WriteLine("failed enabling busy greeting:" + res);
}
```

The only other thing to do with greetings is to decide what you do with the call after the greeting plays. This involves setting the greeting's AfterGreetingAction, AfterGreetingConversation and AfterGreetingTargetHandlerObjectId values. That's a little more advanced and it's a pattern that's used in many instances on numerous objects (exit action for a user, menu entry keys, after greeting actions etc...). A fuller explanation of how to manage these three values to make the greeting do or go where you want when it finishes playing is covered in the Setting Actions section.

Transfer Options

Transfer options are a good deal less involved than greetings given they don't involve recorded media. There are only 3 of them to deal with:

Alternate - if active it always determines transfer action.

Off Hours - if active and the schedule the handler is associated with is off hours, it determines what to do with the call.

Standard - if neither other option kicks in, the standard always processes the call. It cannot be disabled.

Transfer options can only really do two things - either try and ring a phone (either doing a release or supervised transfer) or sends the call on to the active greeting rule. As with greetings you cannot create or delete them - only fetch and update. First, let's look at just iterating over the 3 transfer rules for a call handler:

```
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, connectionServer, "",
        "Jeffs Call Handler");
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed fetching handler:"+res);
    return;
}

//get all the user's transfer option
foreach (TransferOption oTempOption in oHandler.GetTransferOptions())
{
    //outputs the transfer type, if its enabled and what it's action is
    Console.WriteLine(oTempOption.ToString());
}
```

and similar to greetings, you can fetch just an individual transfer option by its type name (not case sensitive):

```
TransferOption oTransfer;
res = oHandler.GetTransferOption(TransferOptionTypes.Alternate, out oTransfer);
if (res.Success == false)
{
    Console.WriteLine("Failed fetching transfer option:"+res);
}
```

Also similar to greetings, the alternate and off hours transfer options can be set to expire by a date or by setting the date to null be active forever. The TransferOption class offers a method to handle the details of that for you. In this example we'll first enable the alternate transfer rule forever, then set it to expire on 2/27/2013 at 1:20pm. I'll leave the checks for successful WebCallResult returns out here for readability:

```
//first, enable it forever
res = oTransfer.UpdateTransferOptionEnabledStatus(true);

//now set it to active till 2/27/2013 and 1:20pm - be sure to convert to universal
//time instead of sending the local time from your box.
DateTime oDateToExpire = new DateTime(2013, 2, 27, 13, 20, 00).ToUniversalTime();

res = oTransfer.UpdateTransferOptionEnabledStatus(true, oDateToExpire);

//now disable it - this will fail if you try it on the standard rule.
res = oTransfer.UpdateTransferOptionEnabledStatus(false);
```

and, of course, you'll want to be able to set if it rings a phone, how it rings the phone and what number it dials. It's important to note that the phone number you enter is not checked against the user's restriction tables associated with their Class of Service - you're authenticating as an administrator and making these changes, that's by design. The following example sets the transfer rule to ring a couple different phones in different ways and then turns off transfers for the rule - again, I'm leaving out error handling in here for brevity.

```
//set the transfer rule to ring the phone for 1234 up to 5 times via supervised
transfer
oTransfer.Action = TransferActionTypes.Transfer;
oTransfer.TransferType = TransferTypes.Supervised;
oTransfer.TransferRings = 5;
oTransfer.Extension = "1234";
res = oTransfer.Update();

//set the transfer rule to ring the handler's primary extension release
//(unsupervised)
//CAUTION if the handler has no extension the extension will be blank - good to
//check for that before doing this - for a user it's not a risk since extensions
//are required for them.
oTransfer.Action = TransferActionTypes.Transfer;
oTransfer.TransferType = TransferTypes.Unsupervised;
oTransfer.UsePrimaryExtension = true;
res = oTransfer.Update();
```

```
//set the transfer rule to not ring any phone but go straight to the greeting rule
oTransfer.Action = TransferActionTypes.PlayGreeting;
res = oTransfer.Update();
```

Menu Entries

Every call handler has a set of 12 menu entries, one each for 0-9, @ and # keys. These entries determine what actions are taken when a caller pressing a key during any greeting that is playing for a call handler (or conversely a user tied to a primary call handler, of course). It's important to note that there is only one set of menu entry keys per call handler. More specifically you cannot have different menu entry key mappings that are used when different greetings for the same call handler are played. Some folks assume menu entries are somehow tied to the greetings and that's not the case.

Similar to contact options and greeting rules, you cannot just fetch all menu entries in the system via REST. The menu entries URI includes the ObjectId of the call handler that owns them – so you can just iterate over all the "0" menu entry keys defined for all handlers in the system and check for some mapping you care about. That'd be handy for some scenarios but it's not in the cards for the REST API as it stands.

Also similar to greetings and contacts there is no mechanism to create or delete menu entries as each handler gets a hard coded set of 12 for every call handler created. You can only fetch, examine and update menu entry data. As usual we'll start with the easy one: listing all the menu entries for a call handler we've already fetched:

```
//List all 12 keys for a call handler
foreach (var oKey in oHandler.GetMenuEntries())
{
    //lists the key name, it's action and if it's locked or not
    Console.WriteLine(oKey.ToString());
}
```

You can also fetch an individual menu entry by its key name and then update a property on the key and save it like this:

```
//fetch just a single key by name
MenuEntry oMenuEntry;
res = oHandler.GetMenuEntry("2", out oMenuEntry);
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch menu entry:"+res);
}

//update the locked property to be false
oMenuEntry.Locked = false;
res=oMenuEntry.Update();

if (res.Success == false)
{
    Console.WriteLine("Menu entry update failed:"+res);
}
```

Pretty straight forward and follows the same use patterns we've seen for other types of objects so far. At this point I should probably pause and explain more fully what the behavior is for menu entries since I end up answering this question quite a bit. You'll notice in the above example we "unlocked" the key. What, exactly, does that mean? By default the keys are not locked which means during the greeting if a user hits, say, the "1" key it will stop the greeting and wait for the inter-digit timeout period to see if the user is going to hit more digits (i.e. enter an extension). By default this is 1500 milliseconds, adjustable on a per handler basis. If no further keys are entered, it takes whatever action the key is mapped to (we'll cover that in a bit). If, instead, the key is locked, the action associated with the key is executed right away with no waiting around.

So why would you want to do this? Couple scenarios: First, you can lock all keys and set them to ignore (i.e. do nothing). This means the entire greeting will play and the caller is helpless to interrupt no matter which or how many keys they mash on. If you have a really important message or just really want to annoy your callers, this is a handy technique for that. A trickier scenario involves you wanting to avoid folks "cold dialing" certain users. Say for instance all your valuable engineers (and honestly – who is more valuable than your engineers?) are on extensions starting with 7. In your opening greeting you can lock the 7 key which effectively means no one can just dial any extension starting with 7. You get the idea – locking keys should be reasonably rare but when you need it, it's handy to have it.

So what can you do with a menu entry key? You can set the key to do anything from hang up immediately (very rude but efficient), take a message, jump to the end of the greeting, launch a special conversation such as the "Broadcast

message administrator" or send the call to another user, call handler, interview handler or name lookup handler. Lots of options and you control it all by editing three properties on the menu entry:

- Action
- TargetConversation
- TargetHandlerObjectId

This is the same 3 property construction used in many places in Connection to do the same thing – route the call somewhere based on an action. Anything from the user exiting their mailbox conversation to where to go after a greeting finishes playing to how to handle a caller not entering anything in a name lookup handler. Rather than repeat the details of how to use these three properties here and everywhere else I've included a general discussion on how to use the "Action" properties as these are referred to in a consolidated Setting Actions section.

The last thing I want to talk about with respect to menu entries here is transferring to a phone number. There's two ways to approach this. The first is to use the Action trinity to send the caller to a call handler that's configured to ring the phone you want it to dial. This can be useful if you want options for special greetings and supervised transfer options and such. But it's also a little heavy from a system administration standpoint. Sometimes all you want to do is release the call to a phone number and "let it go". There's a short hand way to do that with menu entries directly that does not require the creation of another object in the directory using the alternate contact number. This following code snippet walks through that:

```
//fetch just a single key by name
MenuEntry oMenuEntry;
res = oHandler.GetMenuEntry("2", out oMenuEntry);
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch menu entry:"+res);
    return;
}

//set the key to do a release transfer to x1234
//note that the SDK provides a way to set the supervised transfer options but the
//API currently ignores those options.
oMenuEntry.TransferNumber = "1234";
oMenuEntry.TransferType = TransferTypes.Unsupervised;
oMenuEntry.Action = ActionTypes.TransferToAlternateContactNumber;

res=oMenuEntry.Update();

if (res.Success == false)
{
    Console.WriteLine("Menu entry update failed:"+res);
}
```

Determining if a User is an Owner of a Handler

See the Roles and Policies section for an example of how to check if a user is listed among the owners of a call handler.

Call Handler Templates

[Requires Unity Connection 10.0 or later]

When creating a new call handler you are required to provide a call handler template's objectId to do it. This is a little more annoying than user templates since they have an alias property which can be used instead of the objectId GUID which is a bit easier to remember and use. Nonetheless, the mechanism is the same. Call handlers have quite a few properties and administrators can configure various default behavior/settings for call handlers in templates and use them when creating new handlers.

A call handler template is very much like a call handler but has somewhat fewer properties. Anything that is only filled in for a specific call handler instance such as their extension or their recorded voice name are not present on the call handler template. Outside of that they behave very similar to a call handler including their sub objects such as their transfer and greeting rules, menu entries etc. As such I'm not going to cover all that again for templates; just the basics and you can review the call handler section for more details on how the object model hangs together for call handlers as a whole.

Finding and Fetching Call Handler Templates

You can always count on at least one user template to be in the system since the installer creates one and marks it undeletable. Unfortunately you cannot simply reference it by a known alias as you can with user templates, so you'll always have to fetch at least one template at least once before you start creating call handlers. The SDK provides the usual list fetching methods accompanied by easy paging capabilities if there are very large numbers of templates on your system. This example simply selects the first 20 (default count limit) and dumps the top level information for each out to the console:

```
List<CallHandlerTemplate> oTemplates;
res = CallHandlerTemplate.GetCallHandlerTemplates(_server, out oTemplates);
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch templates:"+res);
    return;
}

foreach (var oTemplate in oTemplates)
{
    Console.WriteLine(oTemplates);
}
```

You can, of course, include filter/sort clauses as optional parameters to the GetCallHandlerTemplates method as you can with most other GetXXX calls in the SDK. Typically for templates such filtering is not necessary; however it's provided for completeness and consistency.

This example shows how to fetch a specific template by name instead of fetching them all as a list:

```
CallHandlerTemplate oTemplate;

res = CallHandlerTemplate.GetCallHandlerTemplate(out oTemplate, _server, "",
    "System Call Handler Template");

if (res.Success == false)
{
    Console.WriteLine("Failed to find template:"+res);
    return;
}

Console.WriteLine("Template found:"+oTemplate);
```

Creating, Updating and Deleting Call Handler Templates

[Creating and updating call handler templates is only available in Connection 10.0 and later]

This example shows how to create a new call handler template, edit a few properties in it and then turn around and delete it. Not a terribly practical example but you get the idea. Notice that we need to provide a MediaSwitchObjectId as well as a message recipient (either a user or a public list) to create a call handler template. Unlike user templates that can simply take another user template in the constructor, you need to pass these values directly for call handler templates. In this example I use the call handler template to fish those three values off of for simplicity.

```
CallHandlerTemplate oTemplate;

res = CallHandlerTemplate.GetCallHandlerTemplate(out oTemplate, _server,
            "", "System Call Handler Template");

if (res.Success == false)
{
    Console.WriteLine("Failed to find template:" + res);
    return;
}
```

```
CallHandlerTemplate oNewTemplate;
res = CallHandlerTemplate.AddCallHandlerTemplate(_server,"New handler template",
    oTemplate.MediaSwitchObjectId, oTemplate.RecipientDistributionListObjectId,
    oTemplate.RecipientSubscriberObjectId,null, out oNewTemplate);

if (res.Success == false)
{
    Console.WriteLine("Failed to create template:" + res);
    return;
}

Console.WriteLine("Template created:" + res);

oNewTemplate.DispatchDelivery = false;
oNewTemplate.DisplayName = "New Display Name";
res = oNewTemplate.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating template:" + res);
    return;
}

Console.WriteLine("Template updated");

res = oNewTemplate.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed to delete template:" + res);
}
else
{
    Console.WriteLine("Template deleted.");
}
```

Post Greeting Recordings

Post greeting recordings are designed to allow sites to configure a recording to always play after a user or call handler's greeting finishes but before any message is recorded (assuming that's how the greeting is configured). The usual reason for this is a legal disclaimer or warning to the caller not to leave personal or account information in the message or the like. You know, lawyer stuff.

The post greeting recordings are stored at the system level and you can include many recorded streams in each post greeting recording instance: one for each language you want to support. This follows the same model for all greetings in Unity Connection. The SDK provides mechanism for getting and fetching recordings using local WAV files or using stream files via the phone interface off the server itself (CUTI). The pattern should look fairly familiar.

Each call handler can be configured to reference a post greeting recording by setting its PostGreetingRecordingObjectId to the ObjectId of the PostGreetingRecording object along with setting the Boolean PlayPostGreetingRecording flag to a value indicating it'll play for all callers (1) or only to external callers (2).

Let's just walk through a couple typical tasks using post greeting recordings here. In the first example we'll simply walk through all defined post greeting recordings (if any) found on the server and pull off each of their recorded streams for all languages and save them off to local WAV files on the local hard drive.

```
//pull off all the streams for each recording and save them off as WAV files
//on the local drive
foreach (PostGreetingRecording oRecording in oRecordings)
{

    var oStreams = oRecording.GetGreetingStreamFiles();
    Console.WriteLine("{0} has {1} streams recorded",oRecording,oStreams.Count);
```

```
    foreach (PostGreetingRecordingStreamFile oStream in oStreams)
    {
        string strFileName = string.Format("c:\\{0}_{1}.wav", oRecording.ObjectId,
            oStream.LanguageCode);

        res = oStream.GetGreetingWavFile(strFileName);

        if (res.Success == false)
        {
            Console.WriteLine("    Error saving stream:"+res);
        }
        else
        {
            Console.WriteLine("    Saved as:"+strFileName);
        }
    }
}

Console.WriteLine("All streams saved off");
```

In the next example lets fetch a specific post greeting recording by its display name and set the US English version of the stream file for that greeting to a local WAV file on the hard drive. Note that display names are not required to be unique (although they should be) – this will return the first instance of the display name encountered. If your system is not enforcing unique display names you'll want to use a different mechanism for choosing greetings (and probably review your system policies).

```
//fetch a post greeting recording and set its US English recording to
//a wav file on the local hard drive
PostGreetingRecording oRecording;
res = PostGreetingRecording.GetPostGreetingRecording(out oRecording,
    _server, "", "Post Greeting 2");

if (res.Success == false)
{
    Console.WriteLine(res);
}

res = oRecording.SetRecordingToWavFile("c:\\LegalWarning.wav",
    (int) LanguageCodes.EnglishUnitedStates, true);

if (res.Success == false)
{
    Console.WriteLine("Failed to set greeting:"+res);
    return;
}

Console.WriteLine("Recording set.");
```

Finally let's create a post greeting recording and delete it. Pretty standard flow here by now I hope.

```
PostGreetingRecording oRecording;

res = PostGreetingRecording.AddPostGreetingRecording(_server, "New Greeting",
  out oRecording);

if (res.Success == false)
{
    Console.WriteLine("Failed to create greeting:"+res);
    return;
}
```

```
res = oRecording.SetRecordingToWavFile("c:\\Greeting.wav", 1033, true);

if (res.Success == false)
{
    Console.WriteLine("Failed to set stream:"+res);
}
else
{
    Console.WriteLine("Recording created and stream set.");
}
```

Directory Handlers

[Directory handler adds/delete and greeting stream updates require Unity Connection 10.0 or later]

Directory handlers (aka Name Lookup Handlers) are used to help callers find a user in your directory by name. They come in two flavors: a TUI version (touch tone spelling) and a VUI version (voice driven). Since I'm often asked, no, you cannot create a name lookup handler that does both at the same time. You can offer an exit destination from VUI handler to get you to a TUI handler and vice version (which is a common scenario) but a VUI handler only accepts mapped keys for exiting (0, * typically) and voice commands, it cannot also accept touchtone spelling.

Name lookup handlers have several exit "actions" defined off of them: no input, no selection (timeout), exit by * key and exit by 0 key (operator). See the Setting Actions section in this document for more details on configuring the action settings and how they work.

Creating, Updating and Deleting Directory Handlers

The creation, update and delete functions should look fairly familiar - they follow the same static methods and class instance method patterns that most other object classes in the SDK do. This code chunk creates a TUI based name lookup handler, updates a few of its properties and then creates a VUI name lookup handler which it then deletes.

```
//create a new TUI (touch tone) directory handler
DirectoryHandler oDirHandler;
res = DirectoryHandler.AddDirectoryHandler(oServer, "Test TUI Handler", false,
    null,out oDirHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed to create TUI directory handler");
    return;
}

//update directory handler properties
oDirHandler.AutoRoute = false;
oDirHandler.DtmfAccessId = "774123";
res = oDirHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update TUI dir handler:"+res);
    return;
}

//create a new VUI (voice driven) directory handler
res = DirectoryHandler.AddDirectoryHandler(oServer, "Test TUI Handler",true,
    null, out oDirHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed to create TUI directory handler:"+res);
    return;
}

//now delete the handler
```

```
res = oDirHandler.Delete();
if (res.Success==false)
{
    Console.WriteLine("Failed to delete name lookup handler:"+res);
}
```

Note that Connection does not prevent you from making multiple directory handlers with the same display name. It's a very bad practice to have multiple directory handlers (or other types of handlers) with the same name so it's a good idea to check for this prior to creating a new directory handler.

Custom Recorded Greetings

Starting with the release of Unity Connection 10.0, the API supports updating the custom recorded greetings for directory handlers. This lets you apply a custom recording that callers hear before spelling a name using DTMF (for touch tone directory handlers) or speaking the name of a user (for speech driven directory handlers). Similar to greetings for call handlers and users, you can upload separate greetings for different languages that play depending on the language associated with the call. The SDK provides mechanisms to upload WAV files from the local file system as a greeting or using the phone as a media device (CUTI) to record via the phone on the server instead.

This example shows how to create a new directory handler and upload a US English custom greeting from a WAV file on the local hard drive and configure the handler to play the custom greeting instead of the default system greeting to callers.

```
if (res.Success == false)
{
    Console.WriteLine("Failed to create handler:"+res);
    return;
}

//1033 is for US English, you can use the built in LanguageCodes
//enum to make the code more readable like this:
//(int)LanguageCodes.EnglishUnitedStates
res = oDirHandler.SetGreetingWavFile(@"c:\test.wav", 1033,true);
if (res.Success == false)
{
    Console.WriteLine("Failed to upload greeting:"+res);
    return;
}

oDirHandler.UseCustomGreeting = true;
res = oDirHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update dir handler:"+res);
    return;
}

Console.WriteLine("Created new directory handler with custom greeting");
```

Note that Connection will let you upload a greeting for a language even if that language is not installed or available on Connection. Make sure you check which languages are available before uploading a version of it to the directory handler – see the Languages section for more details on that.

Interview Handlers

[Editing/adding interview handlers requires Unity Connection 10.0 or later]

Interview handlers are a specialty item in the Unity Connection directory that are designed to help "prod" callers into leaving specific information one item at a time. Humans being what they are will tend to forget their callback number, address, account number etc… when leaving a message for, say, a dispatch service or the like. The interview handler asks them for each piece of information one at a time, recording their response to each one before moving on to the next. The responses are all concatenated into a single WAV file with each response separated by beeps and left as a message

for the interview handler recipient. While not used heavily for specific types of audio text applications these can be invaluable.

An interview handler is a pretty simple structure. Each handler has 20 questions allocated to it and you can record a question for each and dictate how long of a response (in seconds) you'll allow the caller to use before it cuts them off and moves on. The interview handler's not about being polite, it's about extracting information and moving ahead. Clearly you don't really want to use all 20 questions in the vast majority of applications – no human being is going to stay on the phone for that kind of interrogation, but you can try if you like.

All 20 questions are always pre allocated for the interview handler when it's created – all are active but none have recordings so only those questions that have recordings are actually presented to users. It's fine to run through and disable all the questions you aren't using but it's not necessary. Note that interview handlers have no concept of language – there's only one greeting stream for each question, unlike call handlers which can have several for each language installed on the server. If you wish to have interview handlers presented in separate languages you will need to create separate interview handlers for each language.

Creating, Updating and Deleting Interview Handlers

To create a new interview handler all you need is a message recipient to pass in. The recipient can be either a user with a mailbox (subscriber) or a public distribution list. There is no concept of templates for interview handlers as there is for call handlers and users for instance – the recipient and the name are the only required items for creating a new interview handler.

```
//first, fetch a public list to act as the message recipient, for this example we'll
//just use the All Subscribers list since we know it's there.
DistributionList oList;
res = DistributionList.GetDistributionList(out oList, connectionServer, "",
  "allvoicemailusers");

if (res.Success == false)
{
    Console.WriteLine("Failed finding distribution list:"+res);
    return;
}

//now create the new interview handler
InterviewHandler oIntHandler;
res = InterviewHandler.AddInterviewHandler(connectionServer, "new Handler", "",
    oList.ObjectId, null, out oIntHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed to create new interviewer:"+res);
    return;
}

//Update some properties on the interviewer and save them
oIntHandler.DisplayName = "New Display Name";
oIntHandler.DispatchDelivery = true;
res = oIntHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update interviewer:"+res);
    return;
}

//now delete it
res = oIntHandler.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed to delete interviewer:"+res);
    return;
}
```

A Note About Dispatch Delivery

For all call handlers and interview handlers if the message recipient is a distribution list you can flag it for dispatch delivery. In the above example if you attempted to set the dispatch delivery flag to true and the recipient was a user, it would not have any effect, it's left as a regular message for that user as normal. It's only meaningful if the message is targeted at a list of users.

This means all users in the list will get a copy of the message in their inbox (turning on their lamp, triggering notification devices etc…). However the first user to hear the message and "accept it" then has that message removed from all other user's mailboxes. This is designed to allow for groups of users to get messages but ensure only one responds. Users have the option to "accept" or "reject" a message (i.e. the message is asking a question they can't answer for instance). There's one very large caveated to using this feature, however: all users in the list that gets messages flagged for dispatch must be homed on the same Connection server. The process for removing messages from inboxes automatically does not span Unity Connection clusters.

That said, this is considerably more robust and useful than the old "shared mailbox" approach for handling scenarios where multiple users need to have access to messages for a shared purpose. The huge advantage here is the large number of users that can be supported and that each user can be notified using their own notification devices. Further, Connection is not designed for a "shared mailbox" approach and does not lock users out of a mailbox when someone else logs in. Changes to a mailbox and the message stack are done on a "last write wins" basis which can produce some deeply unexpected results when piling numerous callers into a single mailbox all saving/deleting/forwarding messages.

Short version: Don't do the shared mailbox thing. Dispatch messages are designed for that purpose and are considerably more sophisticated and robust.

Finding and Fetching Interview Handlers

Interview handlers follow the same design pattern for fetching individual and lists of handlers in the directory.

```
//Fetch a single interviewer by name
InterviewHandler oIntHandler;
res = InterviewHandler.GetInterviewHandler(out oIntHandler, connectionServer, "",
    "test interviewer");

if (res.Success == false || oIntHandler == null)
{
    Console.WriteLine("Interviewer not found:"+res);
    return;
}

//fetch the first 10 interviewers from the directory
List<InterviewHandler> oInterviewers;
res = InterviewHandler.GetInterviewHandlers(connectionServer, out oInterviewers,
  1, 10);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching interviewers:"+res);
    return;
}

foreach (var oInt in oInterviewers)
{
    Console.WriteLine(oInt.ToString());
}
```

Updating Questions for Interview Handlers

Of course the primary editing on an interview handler you'll be wanting to do is editing the interview questions and uploading media for them. The media upload methods off the InterviewQuestion class follow the same design pattern as the other classes in the SDK. There's a WAV file upload option which includes a flag to rip into raw PCM first and a stream file version designed to be used with CUTI when recording using the phone off the server instead of using local media resources.

The questions list can be returned off the interview handler class instance and will always return a list of 20 regardless of which are active and which have recorded media associated with them. One difference with the InterviewQuestion class over many others is that the update method takes 3 parameters instead of having you update properties directly on the class instance and using the "dirty flag" approach. There are only three items you can update for a question: Is it active,

how long of a response will it take and the text of the question. The question text is strictly optional and is just there to help administrators keep the question content straight.

This example shows fetching an interview handler, getting it's questions and then editing the first one. Remember, 20 questions are returned and we're dealing with a 0 based array so its questions 0 through 19 here.

```
//Fetch a single interviewer by name
InterviewHandler oIntHandler;
res = InterviewHandler.GetInterviewHandler(out oIntHandler, connectionServer, "",
    "test interviewer");

if (res.Success == false || oIntHandler == null)
{
    Console.WriteLine("Interviewer not found:"+res);
    return;
}

List<InterviewQuestion> oQuestions = oIntHandler.GetInterviewQuestions();
if (oQuestions == null || oQuestions.Count != 20)
{
    Console.WriteLine("Invalid question fetch");
    return;
}

//edit the first question and then upload a wav file for it
//Note the Update method takes params instead of the "dirty list"
//flag approach.
res = oQuestions[0].Update(true, 10, "Say your phone number, including area code");
if (res.Success == false)
{
    Console.WriteLine("Failed updating question:"+res);
    return;
}

res = oQuestions[0].SetQuestionRecording("c:\\sayPhoneNumber.wav", true);

if (res.Success == false)
{
    Console.WriteLine("Failed uploading question recording:"+res);
    return;
}

Console.WriteLine("Interview question updated");
```

Public Distribution Lists

Public distribution lists in Connection are used mostly for message addressing targets – unlike in Unity they do not act as containers for "ownership" assignment for call handlers or the like. They can be tied as a message recipient for a call handler or interviewer or be addressed by a subscriber in after they log in and address a message by name or ID. Distribution lists can also act as a "scope limiter" for name lookup handlers, but that's their only "non message" related function in the system at present.

To be clear, outside callers cannot "dial" a distribution list (I get this question fairly frequently actually). The only way for outside callers to gain access to leaving a message for a list is by assigning the list as the message recipient for a handler and sending the caller to that handler.

Lists can contain users, system contacts and other lists as members. There is no limit to the "depth" of a list tree – however managing list membership when you're dealing with trees dozens deep will be daunting at best.

Creating and Deleting Distribution Lists

For distribution lists there are no templates and all you need to provide is a unique display name and a unique alias. Both the alias and display name need to be unique among all distribution lists or a failure will be returned. An extension is

optional and the “AddDistributionList” method does also provide a way to pass in additional parameters similar to other “Add” object constructors in the SDK – here’s it’s passed as null.

```
//Create a new DL and return a DistributionList object for it
DistributionList oList;
res = DistributionList.AddDistributionList(connectionServer, "New DL", "newdl",
  "1234", null,out oList);

if (res.Success == false)
{
    Console.WriteLine("Error creating list:"+res);
    return;
}

Console.WriteLine("New list created:"+oList.ToString());

//Then turn around and delete it
res = oList.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed deleting list:"+res);
}
```

Note that the extension (“1234” in this case) is entirely optional – this allows users to address to distribution lists by extension if they prefer but it can be passed as blank if addressing by name only is desirable or if the list you’re creating will not be used for message addressing by subscribers.

As with other class “Add” methods, you can also pass in a name/value pair list that will be applied at creation time and opt not to fill in a class object with the newly created list’s details. If you are creating many distribution lists “in batch” this is how you’ll want to go about it. There’s only a couple properties on a list you’d want to fiddle with, however. Notably if you wanted the list in another partition or if you wanted to allow system contacts to be included as members you could do that with the following chunk of code:

```
//Create a distribution list passing in some additional properties
//and not returning a DistributionList object.  Creation is done in
//a single POST operation and no follow-on GET is done.
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("AllowContacts", true);
oProps.Add("PartitionObjectId", strPartitionObjectId);

res = DistributionList.AddDistributionList(connectionServer, "New DL2", "newdl2",
  "", oProps);

if (res.Success == false)
{
    Console.WriteLine("Failed creating DL:"+res);
    return;
}

Console.WriteLine("New list created, objectID="+res.ReturnedObjectId);
```

Finding and Fetching Distribution Lists

Fetching a single distribution list using the alias string follows the same convention other object classes in the SDK do – here’s a code chunk of finding a distribution list and updating its display name and extension:

```
//Fetch a single list by alias and update it
DistributionList oList;

res =DistributionList.GetDistributionList(out oList, connectionServer, "", "newdl");

if (res.Success == false)
```

```
{
    Console.WriteLine("Failed finding list:"+res);
    return;
}

oList.DisplayName = "New Display Name";
oList.DtmfAccessId = "81248";

res = oList.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating list:"+res);
}
```

Getting a list of distribution lists should also be familiar. Again, as with other list fetching routines it's good to put into place paging practices – although distribution lists are considerably smaller to fetch top level details for than handlers or users, it's still good practice to limit fetches to 100 objects at a time. This code chunk goes through all public lists found in the directory, 25 lists at a time:

```
//Fetch and display all lists in sets of 25
List<DistributionList> oLists;
int iPageCount = 1;

do
{
    res = DistributionList.GetDistributionLists(connectionServer, out oLists,
                    iPageCount, 25);

    if (res.Success == false)
    {
        Console.WriteLine("Failed fetching lists:"+res);
        break;
    }

    foreach (var oList in oLists)
    {
        Console.WriteLine(oList.ToString());
    }

    iPageCount++;

} while (oLists != null && oLists.Count > 0);
```

Updating Distribution List Membership

Simple updates of lists were show above for the handful of properties you can edit at the top level such as display name and extension. Of more interest, of course, is managing the membership information of a list. We'll show here how to add users and list as members and then remove them. First, let's add a public list as a member to another public list:

```
//Fetch a distribution list and add the "Undeliverable Messages" DL
//as a member of it.
DistributionList oList;

res = DistributionList.GetDistributionList(out oList, connectionServer, "", "newdl")
;

if (res.Success == false)
{
    Console.WriteLine("Failed finding list:" + res);
    return;
```

```
}

//now fetch the "undeliverable messages" list to add as a member
DistributionList oMemberList;
res = DistributionList.GetDistributionList(out oMemberList, connectionServer, "",
    "undeliverablemessages");

if (res.Success == false)
{
    Console.WriteLine("Failed finding member list:"+res);
    return;
}

//add the Undeliverable Messages DL as a member to our new dl
res = oList.AddMemberList(oMemberList.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed adding list as member:"+res);
}
```

Now let's add a user to that same distribution list (won't show fetching the first list again for brevity):

```
//fetch the "operator" user to add as a new member
UserBase oOperator;
res = UserBase.GetUser(out oOperator, connectionServer, "", "operator");

if (res.Success == false)
{
    Console.WriteLine("Failed finding operator:"+res);
    return;
}

//add the operator as a member to our new dl
res = oList.AddMemberUser(oOperator.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed adding operator as member:"+res);
}
```

In both these cases if the list you are adding the list or user as a member already contains that list/user as a member, the call will return an error from the server – be sure to check the error reasons as this will be common. Checking the membership up front is expensive, it's easier to add a member and check the result.

Removing members is a bit more work – there's no easy way to fetch a member by name or alias with a query, you need to iterate over the list's membership to find the specific user or list you wish to remove and removed them by their membership identifier. To be clear, you do not remove a member by the object's identifier (i.e. the ObjectId of the user themselves for instance) but by their membership objectId – which means you have to find the member in the list before you can remove them.

This code chunk iterates through the members looking for a user with an alias of "operator" and removes it if found:

```
//find the operator user and remove them as a member if found.
//first, fetch the members of the list
List<DistributionListMember> oMembers;
res = oList.GetMembersList(out oMembers);
if (res.Success == false)
{
    Console.WriteLine("Failed fetching members:"+res);
    return;
}
```

```
//iterate over members looking for the operator user
foreach (var oMember in oMembers)
{
    if (oMember.MemberType == DistributionListMemberType.LocalUser &&
        oMember.Alias.Equals("operator"))
    {
        res = oList.RemoveMember(oMember.ObjectId);
        if (res.Success)
        {
            Console.WriteLine("Member removed");
        }
        else
        {
            Console.WriteLine("Failed removing member:"+res);
        }
        return;
    }
}

Console.WriteLine("Member not found in list");
```

Voice Names

Voice names for distribution lists get used in the message addressing conversation only – they are played as confirmation when including a list in the address for a message. This code chunk shows how to update and then turn around and fetch the voice name WAV file for a distribution list:

```
//Fetch a distribution list set and then get the voice name
DistributionList oList;

res = DistributionList.GetDistributionList(out oList, connectionServer, "", "newdl")
;

if (res.Success == false)
{
    Console.WriteLine("Failed finding list:" + res);
    return;
}

res = oList.SetVoiceName(@"c:\VoiceNameIn.wav", true);

if (res.Success == false)
{
    Console.WriteLine("Failed setting voice name:"+res);
}

res = oList.GetVoiceName(@"c:\VoiceNameOut.wav");

if (res.Success == false)
{
    Console.WriteLine("Failed getting voice name:" + res);
}
```

There is also the “SetVoiceNameToStreamFile” method exposed which is used when recording the voice name via the CUTI interface (using the phone as a media device). Set the example for setting the voice name for a User for details on how that works.

Locations

All objects created in the Unity Connection database get assigned to a location when they are created. The vast majority of them are assigned to the “primary location” that is created when you install Connection to begin with. This is the location that uniquely identifies a Connection server on a network. Since it’s something that needs to be used when creating new objects many times as well as for filters and such, the ConnectrionServerRest class has the

“PrimaryLocationObjectId” property built into it for each retrieval – this is implemented as a “lazy fetch” such that it only issues a query to the server once and then stores the value locally after that – this value can never change for a server once installed so there’s no worry of it changing out from under you while your application runs.

So, other than it being required as a parameter when creating some types of objects, why do you care? If you’re building an application that needs to communicate with a network of Connection servers you care a lot. There are two very interesting things you can do with locations that we’ll work through some real world samples here. The first is discovering all the Connection servers in the network armed only with the server name for one of them (assuming the admin credentials are identical on all servers in the network which is a fairly common model). The second, and really the primary driver here, is to find a “global user” and then hop to that user’s home Connection server to access or edit their data.

We’ll build a simple application that creates ConnectrionServerRest instances for every Connection server on a network: remember, you can communicate with multiple Connection servers in your application safely as the ConnectrionServerRest class is designed to work in a multiply threaded application and “bottleneck” all HTTPS communications through a single static instance – in other words your HTTP data is not going to be stepping on itself sending/receiving to multiple servers even if you’re doing it on different threads – they will simply have to politely wait in line for their turn. Once we have all these server instances created we’ll move on to using them when “hopping” to home servers for global users you find in the directory.

To review: a small amount of data about users replicates around to all Connection servers in a network. Their name, extension, recorded voice name and some other properties are included in this. It’s enough so they can be found in the directory and have messages addressed to them. However to gain access to all the user’s data, edit the user, get to their messages etc… you must attach to that user’s “home server” – in other words you need to jump to the server that owns the primary location object that user points to. Make sense? OK, let’s build a simple example for how to handle finding users in the global directory and quickly doing something with that user on their home server.

Finding All Connection Servers in a Network

Before we can associate global users with their home servers we need to gather up all the Connection servers on the network – this sample goes through the entire process from scratch:

```
//our "anchor" Connection server we'll be attaching to first.
ConnectrionServerRest connectionServer;
try
{
    connectionServer = new ConnectionServer("CUC10a", "CCMAdministrator",
        "ecsbulab");
}

//basic error handling - bark and bail.
catch (Exception ex)
{
    Console.WriteLine("Could not attach to Connection server: " + ex);
    return;
}

//.NET has a wide range of very useful container classes that can be leveraged here
//but the dictionary is simple and fits the bill.  We'll use the primary
//locationObjectId value of the servers as their key for fetching the
Dictionary<string, ConnectionServer> oNetworkServers = new Dictionary<string,
        ConnectionServer>();

//add the first server to the dictionary.
oNetworkServers.Add(connectionServer.PrimaryLocationObjectId,connectionServer);

//now, get all the locations that the server we just attached to knows about - if
//the replication on the network is up to date this should include every server on
//the network.

List<Location> oLocations;
WebCallResult res = Location.GetLocations(connectionServer, out oLocations);

if (res.Success == false)
{
    Console.WriteLine("Failure fetching locations:"+res);
    return;
}
```

```
    foreach (var oLocation in oLocations)
    {
        //don't try to add the "anchor" Connection server to the dictionary again
        if (oLocation.ObjectId == connectionServer.PrimaryLocationObjectId)
        {
            continue;
        }

        //locations can point to different destination types - we're only interested in
        //other Unity Connection servers in the network.
if (oLocation.DestinationType == DestinationType.Connection)
        {
            //create a new Connection server instance for this target - this assumes the
            //same admin credentials are being used for all servers of course.  The
            //HostAddress contains the IP address of the server in question.
            try
            {
                ConnectrionServerRest oNewServer = new
                   ConnectionServer(oLocation.HostAddress, connectionServer.LoginName,
                   connectionServer.LoginPw);

                oNetworkServers.Add(oLocation.ObjectId,oNewServer);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error attaching to server:"+ex);
                return;
            }
        }
    }

    //at this point we have all the locations on the network tucked into the
    //dictionary - for fun list all the servers we found.
    Console.WriteLine("Connection servers found:");
    foreach (var oServer in oNetworkServers.Values)
    {
        Console.WriteLine(oServer.ToString());
    }
```

Finding the Home Server for a Global User

Now that we have a list of all the servers, we can easily "jump around" to different servers to get data for any user in the directory we find. A typical scenario would be using the alias or extension to find a user in the global directory and then fetch their details. What you want to do with the user is not really important; this just covers the fetch mechanism. This builds on the example above, I'll just pick up where it left off here:

```
    //If the directory replication is working properly all users should be
    //represented as a global user on every Connection server in the network.
    //So looking for a user by their alias here can be done against our "anchor"
    //Connection server no problem.
    GlobalUser oGlobalUser;
    res =GlobalUser.GetUser(out oGlobalUser, connectionServer, "", "jlindborg");

    if (res.Success == false)
    {
        Console.WriteLine("Failed fetching user by alias:"+res);
        return;
    }

    //fetch the home server using the locationObjectId off the global user object
    //and leveraging the simple dictionary of servers we constructed in the above
    //example.
    ConnectrionServerRest oTempServer;
```

```
if (!oNetworkServers.TryGetValue(oGlobalUser.LocationObjectId,out oTempServer))
{
    Console.WriteLine("Failed fetching Connection server for location="+
        oGlobalUser.LocationObjectId);
    return;
}

//now fetch the details for the user off their home server.  The ObjectId of the
//global user will match the objectId of the user in their home server.
UserFull oUserFull;
res = UserBase.GetUser(out oUserFull, oTempServer, oGlobalUser.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching user details:"+res);
    return;
}

//now we have the UserFull isntance for this user - we can get their transfer
//rules, edit their name, get at their messages etc... Easy.

Console.WriteLine("Fetched user:"+oUserFull);
```

See? That's not so bad – In a little more than a page of code you've created an easy way to get at all your users corporate wide with a minimum of fuss and muss. Be sure to tell your boss it was super hard and pad some goof-around time into your schedule.

Roles and Policies

Users in Connection can be assigned to a set of roles that determine access to the administrative interface and/or the ability to do things such as change the greetings for call handlers they are listed as the owner for. There are a static set of roles with fixed names (i.e. you cannot add or remove roles in Connection currently) and users are assigned to roles by creating a policy that maps the user to the role – and in the case of some roles further maps it to an object (such as a call handler) that the role applies to. Yes, this is a little bit ugly to deal with.

Don't fret; the SDK is here to make your life a bit easier. The Policy class has a few static methods that will save you a bunch of time for a couple common tasks. One item you may want to do when authenticating users for access to certain sites or the like is to check which role(s) they are assigned to quickly. This sample shows validating a user's alias and password and then checking to see if that user has the "Audio Text Administrator" role assigned to them.

```
UserBase oTestUser;

//validate the user's alias and password are valid and get the user details
//off the server in one shot
if (!connectionServer.ValidateUser("jlindborg", "ecsbulab", out oTestUser))
{
    Console.WriteLine("Validation failed, exiting");
    return;
}

//get the list of role names (if any) the user is assigned to
List<string> oRoleNames;

res = Policy.GetRoleNamesForUser(connectionServer, oTestUser.ObjectId,
    out oRoleNames);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching roles for user:"+res);
    return;
}

//check to see if the user has the "Audio Text Administrator" role assigned.  This
is case
```

```
//sensitive.
if (oRoleNames.Contains("Audio Text Administrator"))
{
    Console.WriteLine("User has the audio text administrator role!");
}
else
{
    Console.WriteLine("User does not have audio text administrator role.");

}
```

Another task you may want to do here is to check if the user is the owner of a call handler – among other things this dictates if they have the right to update that call handler's greeting from over the phone – this is the "Greeting Administrator" role, however you need to do a little more work to see that they have the role against a particular call handler you're interested in. When you assign a user as an owner of a call handler (a handler can have many owners) then the user is given the Greeting Administrator role for that handler. This code chunk validates a user and checks to see if they're listed as an owner of the opening greeting call handler:

```
UserBase oTestUser;

//validate the user's alias and password are valid and get the user details
//off the server in one shot
if (!connectionServer.ValidateUser("jlindborg", "ecsbulab", out oTestUser))
{
    Console.WriteLine("Validation failed, exiting");
    return;
}

//fetch the opening greeting call handler
CallHandler oCallHandler;
CallHandler.GetCallHandler(out oCallHandler, connectionServer, "",
        "Opening Greeting");

//we need the ObjectId of the Greeting Administrator role to compare with.
string strRoleObjectId= Role.GetObjectIdFromName(connectionServer,
        "Greeting Administrator");

//get all the policies for our user
List<Policy> oPolicies;
res = Policy.GetPoliciesForUser(connectionServer, oTestUser.ObjectId,
        out oPolicies);

//get a little tricky with LINQ - this determines if the user is listed as
//an owner of the opening greeting call handler
if (oPolicies.Any(oPolicy =>
        oPolicy.RoleObjectId == strRoleObjectId &&
        oPolicy.TargetHandlerObjectId == oCallHandler.ObjectId))
{
    Console.WriteLine("User is an owner for the call handler");
}
else
{
    Console.WriteLine("User is not an owner for the call handler");

}
```

Setting Actions

Actions are a set of values on many objects in various places in the database that define what to do with a user on the phone in the Connection conversation. For instance after leaving a message there is an "action" that lets you decide where the caller goes after the message is send – hang up on them (rude!), send them to the opening greeting, send them to the "say goodbye" call handler etc…

Since this comes up in so many places in the Connection database scheme, I'm going to discuss the meaning and use of the three properties related to "action" options here instead of repeating it all over the place. You'll have to apply the information here to the specific area in question which means the property names may be a little different. For instance UserFull has 3 properties for where to go when users exit their mailbox conversation called ExitAction,

ExitTargetConversation and ExitTargetHandlerObjectId. The DirectoryHandler class actually has 4 separate sets of action properties for “Exit”, “NoSelection”, “NoInput” and “Zero” used as prefixes. They all work identically regardless of naming convention. I believe your grasp of transitive relationships is up to the challenge this bit of documentation simplification requires.

One of the common design patterns in the Connection database schema is the use of the “Action trinity” which allows for the definition of what to do with a call by defining three properties:

- **Action**. Integer that enumerates 9 types of actions you can do with a call.
- **TargetConversation**. If the action is “2” which means “goto”, you have to indicate the conversation name that will “talk to” the caller. I’ll explain this in a bit, it’s not as confusing as it sounds.
- **TargetHandlerObjectId**. Again, if the action is “2” for “goto” in addition to the conversation you need to load a call handler, interview handler or name lookup handler. These are the only objects in Connection that can handle processing calls. You can’t for instance, route a call to a schedule or a public distribution list.

As noted above the naming conention for these three properties will vary slightly depending on which object and action you’re working with but they all work identically.

The leadoff hitter in this trinity is the Action value. It can be one of 9 different values:

0. **Ignore**. Only applies to menu entries.
1. **Hang up**. Pretty self-explanatory I hope.
2. **Goto**. Send the call to another destination including a special conversation.
3. **Play the Error Greeting**. Can only be used with menu entries and greeting rules. Usual option to map.
4. **Take a message**. Can be used on menu entries and greeting rules.
5. **Skip greeting**. Only applies to menu entries.
6. **Restart greeting**. Only applies to menu entries.
7. **Transfer to alternate contact number**. Only applies to menu entries.
8. **Route from next routing rule**. Special option that sends the call back to the inbound call routing rule and it “picks up where it left off” in processing the rules. This can be used for processing inbound calls that play a special message for everyone and then go back to processing the call rules as normal for scenarios like “snow day” messages and other system alert type setups.

You’ll of course notice that several of the actions are only used in the menu entry key object. “Skip greeting” doesn’t make much sense when exiting the subscriber mailbox conversation for instance. Note, however, that typically you won’t get an error back from the REST API call if you set an action that doesn’t make sense so long as the value is legal (i.e. an integer). So if you set the exit action for a user to “3” it returns “OK” and does, in fact, set the action value to 3 – however the behavior of the conversation becomes “undefined” at that point so let’s just agree not to do that, ok?

Easily the most used and most powerful option is “2” which is goto. This gets used to launch a special conversation or go to another handler object. We’ll cover the first one now – the list of conversation names you can send a call too that do NOT require a handler object destination are:

- **SubSystemTransfer**. Allows a subscriber to dial a number to transfer to – the user must log in first.
- **SystemTransfer**. Allows the caller to “free dial” a number to transfer to (it must pass a system restriction table first, of course).
- **BroadcastMessageAdministrator**. Allows a subscriber to manage broadcast messages for the server if their settings allow for it. The user must log in first.
- **GreetingsAdministrator**. Allows a subscriber to manage greetings for call handlers they are listed as owners for. The user must log in first.
- **SubSignIn**. Goes to the subscriber sign in conversation (user is asked to enter ID and PIN).

The SubSystemTransfer, BroadcastMessageAdministrator and GreetingsAdministrator conversations don’t make much sense as options outside the subscriber conversation any longer since the custom key map options allow you to grant those options right off the top level subscriber menu which is much nicer. Since the user has to authenticate anyway it’s easier for them to just dial into their mailbox like they always do and choose one of those options off their main menu.

Finally, there’s the option for sending the caller to another handler. This requires you set the conversation name to one of these:

- **AD**. If you want to send the caller to a name lookup handler. “AD” used to stand for “Alpha Directory” many years ago.
- **PHTransfer**. If you want to send the call to the transfer entry point of a call handler (i.e. if a transfer option is configured to ring a phone it will), use this conversation name. “PH” here used to mean “Phone Handler” long ago before we realized we weren’t handling phones, but calls.
- **PHGreeting**. If you want to send the call to the greetings entry point (i.e. skip any transfer options and go right to the greeting), use this conversation name.
- **PHInterview**. If you want to send the call to an interview handler, use this conversation name.

In addition to setting the conversation name in this case you also must provide the ObjectId of an appropriate handler. By “appropriate” I mean of the correct type. AD goes with a DirectoryHandler ObjectId, PHTransfer and PHGreeting go with a CallHandler ObjectId and PHInterview goes with an InterviewHandler ObjectId. Remember, when transferring a call to a

user's greeting remember, you are sending the call to that **user's primary call handler**, not their user objectId. I see this mistake a lot. Don't do that.

NOTE: These Conversation names ARE case sensitive.

So – that's not really that complex. Let's see a few examples here for a couple different objects. First, let's set a user's exit action to go to the greeting for a call handler named "Jeffs Handler". We'll work this one all the way through fetching the user, then the handler then performing the update.

```
//fetch user with alias of "jlindborg" - the exit options are only on UserFull.
UserFull oUserFull;
res = UserBase.GetUser(out oUserFull, connectionServer, "", "jlindborg");

if (res.Success == false)
{
    Console.WriteLine("Could not find user in database by alias=jlindborg");
    return;
}

//fetch a call handler named "Jeffs Handler"
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, connectionServer, "",
  "Jeffs Handler");

if (res.Success == false)
{
    Console.WriteLine("Failed to find handler"+res);
    return;
}

//set our user's exit destination to go to the greeting for Jeffs Handler –
//this means it will skip any transfer options if they're configured and go right
//to the active greeting.
oUserFull.ExitAction = ActionTypes.GoTo;
oUserFull.ExitTargetConversation = ConversationNames.PHGreeting;
oUserFull.ExitTargetHandlerObjectId = oHandler.ObjectId;

res = oUserFull.Update();

if (res.Success == false)
{
    Console.WriteLine("Updating user failed:"+res);
}
```

Not that complicated. Notice the use of the enums provided in the library for making the code more readable. You could just plop a "2" in the ExitAction and that's legal but someone coming along later may not appreciate your magical coding practices – you'll find enums for more limited field values in the library.

Let's do another example. In this case we'll set the "7" menu entry key for a user to go to the sign in conversation.

```
//fetch user with alias of "jlindborg" - the exit options are only on UserFull.
UserFull oUserFull;
res = UserBase.GetUser(out oUserFull, connectionServer, "", "jlindborg");

if (res.Success == false)
{
    Console.WriteLine("Could not find user in database by alias=jlindborg");
    return;
}

//fetch just a single key by name off the user's primary call handler
MenuEntry oMenuEntry;
res = oUserFull.PrimaryCallHandler().GetMenuEntry("7", out oMenuEntry);
if (res.Success == false)
```

```
{
    Console.WriteLine("Failed to fetch menu entry:"+res);
    return;
}

//update the 7 key so that it goes to sign in.
oMenuEntry.Action = ActionTypes.GoTo;
oMenuEntry.TargetConversation = ConversationNames.SubSignIn;
res = oMenuEntry.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating menu entry:"+res);
}
```

Notice that we didn't have to provide a value for the TargetHandlerObjectId in that one since the SignIn conversation doesn't need it. Ok, finally lets set the after message action for a call handler to hang-up.

```
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, connectionServer, "",
  "Jeffs Handler");

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch handler:"+res);
    return;
}

oHandler.AfterMessageAction = ActionTypes.Hangup;
res = oHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("handler update failed:"+res);
}
```

Again, notice that neither the conversation name or target handler objectId value needed to be provided since the sign in action does not require it. One or both of those only ever needs to be provided if the action is set to "2" for "goto".

Schedules

Schedules in Connection's API are notoriously daunting – both via REST and ODBC. At first blush they can be a little complex but when you break it down they aren't that bad. The SDK makes an effort to try and simplify interacting with them as much as possible.

Before we dig into the API calls, lets break down how schedules work in Connection. There are 3 elements to schedules:

1. Schedule Sets.
2. Schedules.
3. Schedule Details.

The short version is a Schedule Set contains 1 or more Schedules. A schedule contains 0 or more Schedule Details. I know… hang with me, this isn't that bad.

A schedule detail consists of a start time and end time stored as minutes from midnight and a Boolean flag for each day of the week indicating it's active or not. So a single schedule detail can be something like "from 8am to 5pm, Monday through Friday".

A schedule can contain any number of schedule details. A simple schedule will just be one – the "Monday through Friday 8am to 5pm" is an example of that. But what if you wanted something like "8am to 5pm Monday, Wednesday, Friday and 7am to 4pm Tuesday and Thursday"? You'd create two details – one for Monday, Wednesday and Friday and another for Tuesday and Thursday. Combined these two details give you what you want. You'd do the same thing if you wanted Monday through Friday 8am to 12pm and 1pm to 5pm (i.e. closed during lunch). You'd do two different schedule details for that. You can get as complex or crazy as you want to here. Typically 1 or 2 details is all that are needed unless you're getting really funky.

A schedule set can contain 1 or more schedules. Why? Holidays. A schedule can be designated as a "holiday" schedule or not. So typically a Schedule set consists of one regular schedule (with associated details) and one holiday schedule (usually that has details that consist of single, whole day items representing holidays on the company calendar). The holiday schedule isn't required but it's pretty normal.

The one thing to keep in mind is that the item you associate users, call handlers, notification devices etc… that have "schedule" references is actually the ObjectId of the schedule set. You never reference schedules or schedule details externally.

OK? Don't worry – the SDK hides the ugliness of checking to see if a schedule is active or not and creating simple schedules for you.

First, lets start with an easy item – get the schedule set for a user's call handler and see if it's active, inactive or holiday according to the schedule for the current time. These are the only three states a schedule can report itself in which is used for greeting selection. If it's active the standard greeting is played, if inactive the after hours greeting is played (if it's recorded) and if it's a holiday the holiday greeting plays (if it's recorded).

```
ScheduleState oState;
//There's a "lazy fetch" to get the primary handler off the user and another
//lazy fetch off the handler to get the schedule set.
oState = oUser.PrimaryCallHandler().GetScheduleSet().GetScheduleState(DateTime.Now);

//will output "ACTIVE", "INACTIVE" or "HOLIDAY" depending on what the schedule details
//the user is assigned
Console.Writeline("Current schedule state for user="+oState.ToString());

//will output all the schedule detail items for all schedules (both regular and
//holiday) that the user is associated with.
foreach (Schedule oSchedule in oUser.PrimaryCallHandler().
   GetScheduleSet().Schedules())
{
     Console.WriteLine("Schedule Name="+oSchedule.DisplayName);

     foreach (ScheduleDetail oDetail in oSchedule.ScheduleDetails())
     {
         Console.WriteLine("Details in schedule:");
         Console.WriteLine(oDetail.DumpAllProps(" "));
     }
 }
```

So you can see the schedule details being iterated over for a schedule and the schedules being iterated over in a schedule set which, in turn, is associated with the user's primary call handler here. Reasonably straight forward when you get the basic object model down.

So the next task is actually creating a schedule that you can use. There's an example showing how to create a schedule for a notification device assigned to a user in the Notification Device section you can review, but let's see that same simplified one step schedule creation for a system schedule (one that will appear in the schedule section in the CUCA web based administration interface). We'll create a schedule that's active Monday, Wednesday and Friday from 5am to 2pm:

```
//Fetch the primary location for the Connection server we're using
List<Location> oLocations;
res = Location.GetLocations(connectionServer, out oLocations,
  "query=(IsPrimary is 1)");

if (res.Success == false || oLocations.Count != 1)
{
    Console.WriteLine("Failed fetching primary location:"+res);
    return;
}

//Pass in the location objectId of the primary location on the
//Connection server as the schedule owner (instead of a user) and
//the schedule will be visible as a system schedule.
res = ScheduleSet.AddQuickSchedule(connectionServer, "New Schedule",
    oLocations[0].ObjectId, "",
```

```
    Schedule.GetMinutesFromTimeParts(5, 0), Schedule.GetMinutesFromTimeParts(14, 0),
    true, false, true, false, true, false, false);

if (res.Success == false)
{
    Console.WriteLine("Failed creating schedule:"+res);
    return;
}
```

Note that you can also "steal" the primary location object off just about any other object you've already fetched from the Connection directory as most objects contain a "LocationObjectId" reference which is almost always the primary location object of the server they are created on. But in this example I fetch it "plain" off the Location class.

OK, so that's not so bad. But now what if you want to get really tricky and create some crazy detailed schedule for some reason? Lets say Monday, Wednesday and Friday it's active from 8am to 12pm and from 1pm to 5pm and Tuesday and Thursdays it's active from noon to 5pm. Then we need to break it down and do it by parts:

```
//first, create the schedule set
ScheduleSet oScheduleSet;
res = ScheduleSet.AddScheduleSet(connectionServer, "Test Sched", strLocationObjectId
  , "",out oScheduleSet);

if (res.Success == false)
{
    Console.WriteLine("failed creating schedule set:" + res);
    return;
}

//now create a schedule
Schedule oSchedule;
res = Schedule.AddSchedule(connectionServer, "Test Sched", strLocationObjectId, "",
  false, out oSchedule);

if (res.Success == false)
{
    Console.WriteLine("Failed creating schedule:"+ res);
    return;
}

//now add some details to our schedule
//Monday-Wednesday-Friday before lunch.
res=oSchedule.AddScheduleDetail("MWF 8am-12pm",
  Schedule.GetMinutesFromTimeParts(8,0), Schedule.GetMinutesFromTimeParts(12,0),
  true, false, true, false, true, false, false);

if (res.Success == false)
{
    Console.WriteLine("failed adding detail:"+res);
}

// Monday-Wednesday-Friday after lunch
res = oSchedule.AddScheduleDetail("MWF 1pm-5pm",
  Schedule.GetMinutesFromTimeParts(13,0), Schedule.GetMinutesFromTimeParts(17,0),
  true, false, true, false, true, false, false);

if (res.Success == false)
{
    Console.WriteLine("failed adding detail:" + res);
}

//Tuesday, Thurs 1 to 5.
res = oSchedule.AddScheduleDetail("TTh 1pm-5pm",
  Schedule.GetMinutesFromTimeParts(13,0), Schedule.GetMinutesFromTimeParts(17,0),
  false, true, false, true, false, false, false);
```

```
if (res.Success == false)
{
    Console.WriteLine("failed adding detail:" + res);
}

//Finally, add our schedule to the schedule set
res = oScheduleSet.AddScheduleSetMember(oSchedule.ObjectId);
if (res.Success == false)
{
    Console.WriteLine("Failed adding schedule to set:"+res);
}
```

A bit more work to be sure but really not that bad if you break it down into its parts.

Languages

Dealing with language settings can sometimes be troubling since fetching the installed languages from the server for various types is tedious. The SDK provides a simple InstalledLanguage class that takes care of most of this for you. There are four types of language functions in Connection: TUI, VUI, GUI and TTS. To review nomenclature here:

TUI = Touch tone user interface (phone conversation)

GUI = Graphical user interface (web admin)

VUI = Voice driven user interface (speech recognition - currently only US English)

TTS = Text to Speech engine for reading text.

The following code chunk shows the process of fetching the installed language details from a Connection server and using it in your application to review the languages and check if specific languages are installed for specific functions:

```
//get the installed languages on the server
InstalledLanguage oLanguages;
try
{
    oLanguages = new InstalledLanguage(connectionServer);
}
catch (Exception ex)
{
    Console.WriteLine("failed fetching languages:"+ex);
    return;
}

//dump all installed languages out - includes their ID, description and type.
foreach (var oLang in oLanguages.InstalledLanguages)
{
    Console.WriteLine(oLang.ToString());
}

//check to see if languages are installed - defaults to checking for TUI language
type.
Console.WriteLine("1033 TUI=" + oLanguages.IsLanguageInstalled(1033));
Console.WriteLine("1036 TTS=" +
oLanguages.IsLanguageInstalled(1036,LanguageTypes.TTS));

//if you don't know the language code or you don't like using "magic numbers" you
can use the LanguageCodes enum
Console.WriteLine("Japanese TUI=" +
oLanguages.IsLanguageInstalled((int)LanguageCodes.Japanese));
```

Time Zones

Time zones can be annoying to have to deal with given they change really more than they should. If you're like me there should be exactly 24 time zones and each locale in the world gets to pick one of them. I don't care which one, go wild, but

that's it. This business where there are hundreds of different zones with 15 minute offsets and varying compliance with DST is just insane. But I digress. The SDK provides a TimeZones class that pulls all the timezones defined on the Connection server and provides an easy mechanism to get the details of the time zone users are associated with including daylight savings settings, bias, description and such. This code chunk shows how to create an instance of the TimeZone class and use it to display information about the time zone a user is associated with.

```
TimeZones oZones;
try
{
     oZones = new TimeZones(connectionServer);
}
catch (Exception ex)
{
    Console.WriteLine("Failed fetching time zones:"+ex);
    return;
}

ConnectionTimeZone oCxnTimeZone;
res =oZones.GetTimeZone(oUserBase.TimeZone, out oCxnTimeZone);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch user time zone:"+res);
    return;
}

//dumps out the id, display name and bias of the time zone.
Console.WriteLine(oCxnTimeZone.ToString());
```

Partitions

Partitions and search spaces work together to provide a way to group objects and extensions and allow for things such as overlapping numbering plans, tenant services type object segmentation and the like. A lot of folks can get confused over how they work and why they're used so let's cover the basics here.

All directory objects (i.e. anything that can contain an extension number – users, contacts, call handlers, interviewers, distribution lists, name lookup handlers, locations…) are assigned to a partition. This includes alternate extensions which are separate items – in other words a user can have alternate extensions that are homed in OTHER partitions than the user is assigned to. I know, it seems strange but you may want users in other partitions to be able to dial you by specific numbers.

All extension numbers for all objects in a partition must be unique. In a basic install of Unity Connection there is a single partition and a single search space. All directory objects are assigned to the one partition which is contained in the single search space. Everyone can dial everyone else, there's no overlapping numbers, all is well.

In a larger organization it's typical that everyone has a "short dial" number such as a 4 digit extension. These can typically overlap in the global directory – so there can be two people with the extension "3189". Those user's must be in separate partitions then or it's a conflict. Typically then they will have a 2nd globally unique number (i.e. a 10 digit number). These can all live in a single partition.

For instance you might have a setup like this:

"Global Partition"

"Chicago Partition"

"San Jose Partition"

"Seattle Partition"

So everyone has their short extension assigned to either Chicago, San Jose or Seattle. Everyone's long unique extension is in the Global partition. Now, what about the search space? Every server could have a single search space that contained all 4 of the partitions (remember, partitions replicate around the network). So in Seattle search space would contain the Global partition and the Seattle partition followed by the San Jose and then the Chicago partition.

Order in the search space is important. If someone in Seattle called into Connection and dialed "3189" it will search the global partition first, not find a match, then go to Seattle's partition and search – it will find me and take it. If there's other "3189" extensions in Chicago and/or San Jose they won't be found – it's a "first match wins" model.

But I can dial anyone in the directory using their 10 digit unique number which gets picked up in the global partition. See? You can get all kinds of complex including 7 digit numbers and the like but at its root the partition and search space objects work together to create an extension grouping mechanism. Not so complex.

There's very few properties on a partition, basically just its name and description – a partition doesn't have any functionality unto itself, it's really just a placeholder to indicate grouping membership.

Finding and Fetching Partitions

```
//All partitions from all servers can be found – they replicate around the
//network like users and locations do.
List<Partition> oPartitions;
WebCallResult res = Partition.GetPartitions(connectionServer, out oPartitions);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching partitions:"+res);
    return;
}

foreach (var oPartition in oPartitions)
{
    Console.WriteLine(oPartition);
}
```

Creating, Editing and Deleting Partitions

The Partition class follows the same design conventions that most of the other object classes do allowing for creating via static methods, updates off instances and such – this code chunk shows creating a new partition (the name must be unique), editing the description and then deleting the partition.

```
Partition oPartition;
WebCallResult res = Partition.AddPartition(connectionServer, out oPartition,
    "New Partition","New partition");

if (res.Success == false)
{
    Console.WriteLine("Failed creating partition:"+res);
    return;
}

Console.WriteLine("Partition created:"+res);

//update the description of your new partition
oPartition.Description = "My new description";
res = oPartition.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating partition:"+res);
    return;
}

//delete the partition you just added
res = oPartition.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed deleting partition:"+res);
}
```

Search Spaces

Search spaces work with partitions for directory segmentation – see the Partitions section above for a breakdown of how the segmentation mechanism is design.

Like partitions, search spaces don't have much information on them besides names and descriptions – the exception here is that you associate partitions with them. A search space should have at least one partition but can have many. As noted in the partitions section the order of the partition assignment is important since the first match for an extension number is the one selected even if other partitions later in the list also contain that same extension.

Whenever an extension is searched against for any reason there's a search space to dictate the limits of that search. This includes a user signing into their mailbox and addressing a message by extension, a caller dialing an extension number from a call handler greeting or the like.

Call handlers are assigned to one search space, mapped to its "CallSearchSpaceObjectId" property. Users, however, have two(!?). One is used to dictate the scope of the search when the user is addressing messages using extension numbers and there is a separate one for dictating the scope when addressing by speech recognition. Currently these are always mapped to the same search space in the web admin interface but fun fact: you can set them separately in the API. Use your powers only for good.

Finding and Fetching Search Spaces

Like most of the other classes in the SDK, you can fetch the search spaces using static methods, nothing unusual here, hopefully.

```
List<SearchSpace> oSearchSpaces;
WebCallResult res = SearchSpace.GetSearchSpaces(connectionServer, out
oSearchSpaces);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching search spaces:"+res);
    return;
}

Console.WriteLine("Search spaces found:");
foreach (var oSpace in oSearchSpaces)
{
    Console.WriteLine(oSpace);
}
```

Creating, Editing and Deleting Search Spaces

In this code chunk we'll create two new partitions, create a search space and then add those two partitions to it in order. Finally we'll assign the new search space to a user.

```
//Create two partitions we'll add to our new search space.
Partition oPartition1;
WebCallResult res = Partition.AddPartition(connectionServer, out oPartition1,
    "Partition1","Partition1");

if (res.Success == false)
{
    Console.WriteLine("Failed creating partition:"+res);
    return;
}

Partition oPartition2;
res = Partition.AddPartition(connectionServer, out oPartition2,
"Partition2", "Partition2");

if (res.Success == false)
{
    Console.WriteLine("Failed creating partition:" + res);
    return;
```

```
}

//create the new search space
SearchSpace oSearchSpace;
res = SearchSpace.AddSearchSpace(connectionServer, out oSearchSpace, "New
SearchSpace", "New SearchSpace");

if (res.Success == false)
{
    Console.WriteLine("Failed creating search space:"+res);
    return;
}

//Add the partitions in order
res = oSearchSpace.AddSearchSpaceMember(oPartition1.ObjectId, 1);
if (res.Success == false)
{
    Console.WriteLine("Failed adding partition:"+res);
}

res = oSearchSpace.AddSearchSpaceMember(oPartition2.ObjectId, 2);
if (res.Success == false)
{
    Console.WriteLine("Failed adding partition:" + res);
}

//Now update a user's search space.  Its good style to assign both the
//extension search space and name search space limits to be the same since
//this is what the CUCA web admin interface does.

UserFull oUser;
res = UserBase.GetUser(out oUser, connectionServer, "", "jlindborg");
if (res.Success == false)
{
    Console.WriteLine("Failed fetching user:"+res);
    return;
}

oUser.SearchByExtensionSearchSpaceObjectId = oSearchSpace.ObjectId;
oUser.SearchByNameSearchSpaceObjectId = oSearchSpace.ObjectId;
res = oUser.Update();
if (res.Success == false)
{
    Console.WriteLine("Failed updating user:"+res);

}
```

Class of Service

Finding and Fetching Classes of Service

First, we'll just list all the Class of Service objects on the server – normally there's not that many of these so there's no paging construct built into the class. If someone comes up with a system that has thousands of classes of service, we'll come back to that, but first I want to hear why you have some many classes of service! Note that when fetching a list of Classes of Service, most of the data in the class wont be filled in – just the name and ObjectId are filled in when fetching lists. This is similar to UserBase and UserFull however the SDK does not break out the class of service into a ClassOfServiceBase and ClassOfServiceFull. You just need to be aware that if you need the full set of properties you need to fetch the individual class of service using the display name or objectId (next example). But the vast majority of the time all you need from the COS is its name and ID so many times that's enough.

```
//fetch all the Class of Service name/IDs on the server and list them
List<ClassOfService> oCoses;

res = ClassOfService.GetClassesOfService(connectionServer, out oCoses);
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch Coses:"+res);
    return;
}

foreach (var tmpCos in oCoses)
{
    Console.WriteLine(tmpCos.ToString());
}
```

To fetch a class of service by name, you can just pass in the name to the static class method. Names for classes of service are required to be unique across all classes of service so it's fine to do this. The name is not case sensitive here. Remember, this pulls all the properties of a COS, not just the name and ID as fetching a list of COSes does.

```
ClassOfService oCos;
res = ClassOfService.GetClassOfService(out oCos, connectionServer, "",
  "Voice Mail User COS");

if (res.Success == false)
{
    Console.WriteLine("Failed to find COS:"+res);
}
```

Finally, there are some lazy fetch handles for getting the restriction table definitions for outcalling, transfers and fax numbers you can use. You must have fetched the full Class of Service for this to work – the restriction table details are not includes when fetching lists of Classes of Service. For instance if you wanted to see the restriction table definition for transfer numbers assigned to a particular user, you could do it like this:

```
RestrictionTable oTable = oUser.Cos().TransferRestrictionTable();
Console.WriteLine("Transfer RT="+oTable.ToString());
Console.WriteLine("Restriction patterns:");

foreach (var oPattern in oTable.RestrictionPatterns())
{
    Console.WriteLine("    "+oPattern.ToString());
}
```

Creating, Updating and Deleting Classes of Service

The next code chunk shows how to create a COS, update its properties and then delete it. Note that if you try and delete a class of service that has users associated with it, the delete attempt will fail. You must reassign users associated with a COS first before removing it.

```
ClassOfService oCos;

res = ClassOfService.AddClassOfService(connectionServer, "My Cos", null, out oCos);
if (res.Success == false)
{
    Console.WriteLine("Failed creating COS:"+res);
    return;
}

//update COS
oCos.CanRecordName = true;
oCos.AccessAdvancedUserFeatures = true;
oCos.DisplayName = "New Display Name";
oCos.MovetoDeleteFolder = true;

res = oCos.Update();
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed updating COS:"+res);
    return;
}

res = oCos.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed deleting COS:"+res);
}
```

Similar to user creation shown earlier you can create a COS and pass a series of name property values in that will be applied to that COS at creation time instead of first creating the COS and updating it as I've shown here. I pass it as null in this example and update it via the easier instance method since normally you won't be creating large numbers of COSes in batch that would warrant passing the parameters in up front.

Phone Systems, Port Groups and Ports

Phone system integration configuration can be a bit daunting given the number of objects types that are involved with creating a new integration from scratch. This is particularly true if you want to customize which audio codecs are advertised in a port group and the like. The SDK is designed to try and simplify as much of this as possible, however a grasp of the overall object model is still necessary for this to make much sense. In that respect it's a bit like schedules – once the flow is understood they're not as complex as they seem at first blush.

Here's the short version: At the top of the food chain is a phone system object. Under the phone system object you can have one or more port groups. A port group defines which server we're attaching to for phone services, which codec(s) are advertised and what the phone integration type is (SIP, SCCP, PIMG/TIMG). Under port groups are 1 or more port objects. Ports define a phone end point on the Connection server – think of them as a virtual phone. They can be used to answer calls, send MWIs, dial out for notification etc…

To be fully functional and capable of taking phone calls a Unity Connection server must be configured with a phone system that has at least one port group which in turn has at least one port. Without that the Connection server is not able to process inbound phone calls or initiate outbound calls. It's perfectly legal to create users, handlers, lists etc… without a phone system defined, but your system is going to be reasonably useless until you configure the phone system.

An important note: users are associated with the phone system object(s) defined in Connection. They have no attachment to a port group or a port. As such you can delete ports and port groups without issue, but if you attempt to delete a phone system and one or more users are still associated with that phone system, the delete will fail. You must first reassign all those users to another switch before you're allowed to delete it. See the Phone System References section below for more details.

The easiest way to explain how this all hangs together is to just walk through an example that creates all three components needed for Unity Connection to be able to process calls. The next section walks through this process.

Creating a Phone System Integration from Scratch

This example creates a new phone system and then an SCCP port group for that phone system and finally adds 4 ports to that port group. The process for SIP and TIMG/PIMG is nearly identical; you can just pass in the corresponding phone system integration type enumerator value for the integration type you want. We'll cover some other more advanced items dealing with codecs, updating and deleting items separately.

```
ConnectrionServerRest oServer;
try
{
    oServer = new ConnectionServer("192.168.0.194", "CCMAdministrator", "ecsbulab");
}
catch (Exception ex)
{
    Console.WriteLine("Failed to attach to server:"+ex);
    return;
}

Console.WriteLine("Connected to:"+oServer);

WebCallResult res;
```

```
//Create a new phone system
PhoneSystem oPhoneSystem;
res = PhoneSystem.AddPhoneSystem(oServer, "New Phone System", out oPhoneSystem);

if (res.Success == false)
{
    Console.WriteLine("Failed creating phone system:"+res);
    return;
}

//create port group for the server we just created
//In the case of an SCCP port group you must provide the SCCP
//prefix string, SIP and PIMG do not need this value.
PortGroup oPortGroup;
res = PortGroup.AddPortGroup(oServer, "SCCP port group", oPhoneSystem.ObjectId,
    "Cxn91.cisco.com",TelephonyIntegrationMethodEnum.SCCP, "CiscoUM1-VI",
    out oPortGroup);

if (res.Success == false)
{
    Console.WriteLine("Failed creating port group:"+res);
    return;
}


//Create new ports for the port group
//You can update port objects after creating but since typically you're making
//several at a time it's much more efficient to setup your properties in a
//property list and include it in the create call right up front.  In this case
//we're enabling all the capablities on all ports and indicating we want non
//secure SCCP ports.  New ports are enabled by default.
ConnectionPropertyList oProps = new ConnectionPropertyList();
oProps.Add("CapAnswer", true);
oProps.Add("CapNotification", true);
oProps.Add("CapMWI", true);
oProps.Add("CapTrapConnection", true);
oProps.Add("SkinnySecurityModeEnum", (int)SkinnySecurityModes.Insecure);

//add ports - if this is for a PIMG port group be sure to pass the last optional
//parameter as "true" instead.
res = Port.AddPort(oServer, oPortGroup.ObjectId, 4, oProps);

if (res.Success == false)
{
    Console.WriteLine("Failed adding ports:"+res);
    return;
}

Console.WriteLine("Phone system configured, ready to take calls.");
```

Assuming your Call Manager is configured properly for the “CiscoUM1-VI” prefix provided and the DNS is working correctly to resolve “Cxn91.cisco.com” for you, your system will be ready to take calls once that above is executed.

Fetching, Updating and Deleting Phone Systems

Phone systems are at the top level in the Unity Connection directory so the fetch provided by the SDK simply gets them all. Typically there will only be a small handful of separate phone systems, however technically you can create many of them (one for each port up to 256 on paper), so the GetPhoneSystems method used in this example does support paging parameters – it defaults to fetching the first 20.

This example, while a bit contrived, shows all the parts of the fetch, update and subsequent delete of the phone system we created in the above example. Note that when a phone system is deleted, Unity Connection automatically deletes all port groups associated with that phone system which, in turn deletes all ports associated with any of those port groups.

Note, however, that the deletion of the phone system will fail if there are any users referencing it. To handle this problem, see the Phone System References section below.

```
//Fetch all phone systems defined, update the display name for the one
//named "New Phone System" and then delete it.
List<PhoneSystem> oPhoneSystems;
res = PhoneSystem.GetPhoneSystems(oServer, out oPhoneSystems);

foreach (var oPhoneSystem in oPhoneSystems)
{
    if (oPhoneSystem.DisplayName.Equals("New Phone System"))
    {
        oPhoneSystem.DisplayName = "Updated Display Name";
        oPhoneSystem.MwiPortMemory = true;
        oPhoneSystem.CallLoopGuardTimeMs = 1000;
        oPhoneSystem.RestrictDialUnconditional = true;

        res = oPhoneSystem.Update();

        if (res.Success == false)
        {
            Console.WriteLine("Failed updating phone system:"+res);
            return;
        }

        res = oPhoneSystem.Delete();

        if (res.Success == false)
        {
            Console.WriteLine("Failed deleting phone system:"+res);
            return;
        }

        Console.WriteLine("Phone system removed");
        return;
    }
}

Console.WriteLine("Did not find phone system");
```

Fetching, Updating and Deleting Port Groups

Typically when finding port groups you want to find all port groups associated with a particular phone system. You can certainly get all defined port groups but the SDK also provides for an override on the GetPortGroups method so you can pass in the ObjectId of a phone system to get only those port groups associated with a specific phone system you're working with. Note that it's perfectly legal to have a phone system with no port groups defined for it so be sure to check for that.

Again, this example is a bit contrived but shows fetching all ports, editing and then deleting the first one.

```
//Get phone system by name
PhoneSystem oPhoneSystem;
res = PhoneSystem.GetPhoneSystem(out oPhoneSystem, oServer, "","Test Phone System");

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch phone system:"+res);
    return;
}

//get all port groups associated with phone system.
List<PortGroup> oPortGroups;
res = PortGroup.GetPortGroups(oServer, out oPortGroups, oPhoneSystem.ObjectId);
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed to fetch port groups:"+res);
    return;
}

if (oPortGroups == null || oPortGroups.Count < 1)
{
    Console.WriteLine("No port groups found for phone system");
    return;
}

//update the details for the first group
PortGroup oPortGroup = oPortGroups[0];
oPortGroup.DisplayName = "Updated display name";
oPortGroup.DelayBeforeOpeningMs = 123;
oPortGroup.EnableMWI = false;

res = oPortGroup.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed to update port group:"+res);
    return;
}

//delete port group
res = oPortGroup.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed to delete port group:"+res);
    return;
}

Console.WriteLine("Port group deleted");
```

Fetching, Updating and Deleting Ports

Similar to fetching port groups from a phone system object, often you'll want to fetch ports from a port group object. The SDK allows you to fetch all ports at the system level of course but provides an overload that takes the ObjectId of a PortGroup to restrict the list of ports returned to those owned by that port group. Note that it's perfectly legal to have no ports defined for a port group, so be sure to check for that.

Once again the example is a little contrived but this shows fetching, updating and deleting a port from a port group.

```
//Get port group by name
PortGroup oPortGroup;
res = PortGroup.GetPortGroup(out oPortGroup, oServer, "", "SCCP port group");

if (res.Success == false)
{
    Console.WriteLine("Failed fetching port group:"+res);
    return;
}

//get all ports for that port group
List<Port> oPorts;
res = Port.GetPorts(oServer, out oPorts, oPortGroup.ObjectId);

if (res.Success == false)
{
```

```
    Console.WriteLine("Failed fetching ports:"+res);
    return;
}

if (oPorts == null || oPorts.Count == 0)
{
    Console.WriteLine("No ports found for port group");
    return;
}

Port oPort = oPorts[0];

oPort.DisplayName = "Updated display name";
oPort.CapMWI = false;

res = oPort.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating port:"+res);
    return;
}

res = oPort.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed deleting port:"+res);
    return;
}

Console.WriteLine("Port removed");
```

Phone System References

When attempting to delete a phone system you need to be sure there are no users associated with that phone system or the delete operation will fail. The PhoneSystem class has a method designed to help with this process that will return a list of all users associated with the phone system in question. This allows you to loop through all those users and update their phone system reference (media switch ObjectId) to a different phone system prior to doing the delete. The following example shows how this can be done.

```
//get phone system to delete
PhoneSystem oPhoneSystemToDelete;
res = PhoneSystem.GetPhoneSystem(out oPhoneSystemToDelete, oServer, "","Red Shirt");

if (res.Success == false)
{
    Console.WriteLine("Failed to find phone system to delete:" + res);
    return;
}

//get phone system to replace it with
PhoneSystem oPhoneSystemNew;
res = PhoneSystem.GetPhoneSystem(out oPhoneSystemNew, oServer, "", "PhoneSystem");

if (res.Success == false)
{
    Console.WriteLine("Failed to find replacement phone system:" + res);
    return;
}

List<PhoneSystemAssociation> oAssociations;
res = oPhoneSystemToDelete.GetPhoneSystemAssociations(out oAssociations);
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed to find associations:" + res);
    return;
}

//assign all users associated with the old switch to the new one.
foreach (var oAssociation in oAssociations)
{
    UserBase oUser;
    res = UserBase.GetUser(out oUser, oServer, "", oAssociation.Alias);
    if (res.Success == false)
    {
        Console.WriteLine("Failed to fetch user by alias:" + res);
        return;
    }

    oUser.MediaSwitchObjectId = oPhoneSystemNew.ObjectId;

    res = oUser.Update();
    if (res.Success == false)
    {
        Console.WriteLine("Failed to update user:" + res);
        return;
    }
}

//at this point there should be no more references to the switch we want to delete
res = oPhoneSystemToDelete.Delete();
if (res.Success == false)
{
    Console.WriteLine("Failed to delete phone system:"+res);
    return;
}

Console.WriteLine("Phone system removed.");
```

Customizing Codecs

One of the trickier tasks to accomplish related to the phone system integration is changing which audio codecs are advertised by a port group. Currently Unity Connection supports 5 audio codecs that can be selected:

- G.711 mu-law
- G.729
- G.711 a-law
- G.722
- iLBC

The RtpCodecDef static class in the SDK is provided to produce a list of the supported codec and their corresponding ObjectId references.

By default when you create a new port group G.711 mu-law is registered as the preferred codec and G.729 is registered as a lower order preferred codec. If the call being presented to Unity Connection lists both of those as possible codecs to use in other words, the G.711 mu-law codec will be used. This only applies if the phone system integration is SIP since the SIP protocol is the only one that supports the “handshake” protocol to determine which codec to use here.

The easiest way to handle this is to delete all codecs for a port group and add back the one(s) you want to advertise setting the preference value as you do. The following code chunk shows that process and it adds back just G.729 as an example.

```
PortGroup oPortGroup;
```

```
res = PortGroup.GetPortGroup(out oPortGroup, oServer, "", "PhoneSystem-1");

if (res.Success == false)
{
    Console.WriteLine("Failed to get port group:"+res);
    return;
}

//delete all codecs
List<PortGroupCodec> oCodecs;
res = PortGroupCodec.GetPortGroupCodecs(oServer, oPortGroup.ObjectId, out oCodecs);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching codecs:"+res);
    return;
}

foreach (var oCodec in oCodecs)
{
    res = oCodec.Delete();
    if (res.Success == false)
    {
        Console.WriteLine("Failed deleting codec:" + res);
        return;
    }
}

//add back in the G.722 codec
//first, fetch all the codecs on the server
List<RtpCodecDef> oCodecDefs;
res = RtpCodecDef.GetRtpCodecDefs(oServer, out oCodecDefs);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching codec defs:" + res);
    return;
}

//get the ID for the G.722
string strCodecId = oCodecDefs.First(x => x.DisplayName.Equals("G.722")).ObjectId;

//add the codec to the port group
res = PortGroupCodec.AddPortGroupCodec(oServer,oPortGroup.ObjectId,strCodecId,20,1);

if (res.Success == false)
{
    Console.WriteLine("Failed adding codec:"+res);
    return;
}

Console.WriteLine("Codec added");
```

Routing Rules

Ok, so now that we've talked about most of the major objects in Unity Connection used for handling and routing calls we need to talk about the first level handling of calls when they are first presented to the system. Routing rules are the very first object a new inbound call encounters and determines where that call is initially routed. In most cases once the call is dropped to a conversation or an object then routing rules are out of the picture – however there is a "route from next" scenario we'll discuss that allows a call handler, for instance, to send a call back to the routing rules to evaluate a new destination. This can be used for advanced "snow day" type inbound call routing where a greeting is played to all inbound calls and then the call is sent along its way where it would normally go. We'll discuss that special case later.

The process for evaluating routing rules is very simple: all rules are evaluated one at a time starting at the top (index 0) and proceeding across each rule until the conditions for that rule are all met. If they are the action for that rule is taken and you're done. The important bit there is that as soon as a match is found it stops. Easily the most common error I see

with routing rules is accidentally having a very broad rule (i.e. one that fires on ANY forwarded call) firing at the top and none of the more specific rules (i.e. associated with specific ports or calling numbers) never get evaluated. The rule of thumb is to start specific and get broad at the bottom. The last routing rule in the table should always be the generic "opening greeting" rule which fires on ANY call and dumps you to the opening greeting call handler. This is referred to as the "backstop" rule since we don't want to fall off the end of the table ever, you always want that call going somewhere. If not you end up getting the dreaded "failsafe" prompt ("I'm sorry I can't talk to you now") and the caller gets hung up on. Really bad style, don't do that.

Finding and Fetching Rules

The RoutingRule class follows the same convention as most other object types in the SDK: you use static methods for fetching lists or individual rules via out parameters. An important issue here is rule order. By default when fetching lists of rules they are presented sorted from HIGHEST index number to lowest (i.e. from the last rule processed to the first). If you know this and account for it, that's fine but it's an easy way to trip up. To ensure you are getting the rules in the order in which they are processed you can pass in a sort command as in this example:

```
//fetch first 20 rules and print them out
List<RoutingRule> oRules;
res = RoutingRule.GetRoutingRules(_server, out oRules,1,20, "sort=(RuleIndex asc)");
if (res.Success == false)
{
    Console.WriteLine("Failed fetching rules:"+res);
    return;
}

foreach (var oRule in oRules)
{
    //includes the index, name and ID of each rule
    Console.WriteLine(oRule.ToString());
}
```

That said, there's a large limitation to doing this. Given the constraints on the query and sorting options in the API you cannot both sort by the RuleIndex AND filter by display name for instance. So if you wish to get all rules that start with "GroupA_" for instance you cannot also sort by RuleIndex. So if you are going to be pulling a set of rules out based on name or search space reference or the like, be aware that they are presented in reverse order and this can be critical in your understanding of how those rules get processed. This is especially true if you are going to be reordering rules as discussed in a couple sections – a very easy way to burn a system's ability to process calls.

Routing rule names must be unique among all routing rules so searching by name as a unique identifier is safe. The RoutingRule class provides for this in addition to the Unique identifier search.

```
RoutingRule oRule;
res = RoutingRule.GetRoutingRule(out oRule, _server, "", "GroupA Forwarding Calls");
if (res.Success == false)
{
    Console.WriteLine("Failed to find rule:"+res);
    return;
}

Console.WriteLine(oRule.ToString());
```

Creating New Routing Rules

So the process here is to create a rule, decide where it goes in the list and where it will route calls when all its conditions are met and then to add conditions you want to constrain the rule against.

So let's create a new routing rule for our internal group of super-secret developers we'll call "GroupA". This rule should only fire during the GroupA's scheduled work hours and when forwarding from a couple extension numbers we've setup for this purpose. The rule should send the caller to the special opening greeting call handler for GroupA.

```
//first, grab the call handler we want to send callers to
CallHandler oHandler;
res = CallHandler.GetCallHandler(out oHandler, _server,"", "GroupA Greeting");
if (res.Success == false)
```

```
{
    Console.WriteLine("Failed to fetch call handler:"+res);
    return;
}

ScheduleSet oScheduleSet;
res = ScheduleSet.GetScheduleSet(out oScheduleSet, _server, "", "GroupA Schedule");
if (res.Success == false)
{
    Console.WriteLine("Failed fetching schedule:"+res);
    return;
}

//create our new rule
RoutingRule oRule;
res = RoutingRule.AddRoutingRule(_server, "Forwarding GroupB", null,out oRule);
if (res.Success == false)
{
    Console.WriteLine("Failed creating rule:"+res);
    return;
}

//default behavior is to use the caller language (which falls back on the
//system default).  If you wish to "tag" a call with a specific language
//at routing time you need to set the language code and turn the option
//to useCallLanguage off.
oRule.LanguageCode = (int) LanguageCodes.EnglishUnitedStates;
oRule.UseCallLanguage = false;

//send the call to the greeting for the call handler selected
oRule.RouteAction = RoutintRuleActionType.Goto;
oRule.RouteTargetConversation = ConversationNames.PHGreeting;
oRule.RouteTargetHandlerObjectId = oHandler.ObjectId;

//this rule only fires on forwarded calls
oRule.Type = RoutingRuleType.Forwarded;

res = oRule.Update();
if (res.Success == false)
{
    Console.WriteLine("Failed updating rule:"+res);
    return;
}

//add a condition that fires only if the forwarding station is 1234 or 1235
res = oRule.AddRoutingRuleCondition(RoutingRuleConditionOperator.In,
                                    RoutingRuleConditionParameter.ForwardingStation,
                                    "1234,1235");
if (res.Success == false)
{
    Console.WriteLine("Failed adding condition:" + res);
    return;
}

//make sure the rule only fires when our schedule is active
res=oRule.AddRoutingRuleCondition(RoutingRuleConditionOperator.Equals,
                                    RoutingRuleConditionParameter.Schedule,
                                    oScheduleSet.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed adding condition:"+res);
    return;
}
```

```
Console.WriteLine("New rule added!");
```

HINT: If you're creating rules in a setup that's taking calls it's always a good idea to setup the rule as inactive (meaning it's skipped during call processing). This gives you a chance to review the rule and its conditions in the CUCA web interface without potentially interfering with active calls.

There are a lot of steps involved there but the process itself is reasonably straight forward. Of note here, the newly added rule is inserted at index 0 (the top of the routing rule list). All rules are in one table regardless of if they're for forwarded or direct calls (or both).

In most cases if you're adding batches of rules for internal group routing like this if you plan your adds correctly you can (and should) be able to simply add them in at the top in order and you're done. So long as your rule conditions are creating specific enough routing conditions such that they don't fire and "cover up" routing rules later in the list this is the preferred method for adding rules.

Ordering Routing Rules

I'm just going to state this right up front: reordering the routing rules is generally a bad idea and is an operation that can easily harm a Connection server's ability to process inbound calls if not done correctly. Adding specific routing rules to the top as we did in the previous example is reasonably safe and if the rule introduces a problem you can simply disable it or remove it, no harm done. If you have a system with a large number of rules, however, and you reshuffle their order in a large operation you mess up, the entire system is down for the count and cleaning it up manually is a tedious and painful process. If you really need to go this route (hint: you are probably doing it wrong if you do) please be sure to test heavily in lab settings before letting it fly in a production setup.

In this example we'll move a particular routing rule to the topmost slot (index 0) which gets processed first. The exercise here is to produce a list of ALL the routing rules (you can't reorder just a few of them, that will not be allowed), take the one you want to move out of the list and then insert it at the top of the list before calling the SDK's method to update the ordering of the rules.

```
List<string> oRuleObjectIds = new List<string>();
int iPage = 1;
List<RoutingRule> oRules;

do
{
    //Fetch all routing rules in the system, 20 at a time.
    res = RoutingRule.GetRoutingRules(_server, out oRules, iPage, 20,
            "sort=(RuleIndex asc)");

    if (res.Success == false)
    {
        Console.WriteLine("Failed fetching rules:"+res);
        return;
    }

    //We've sorted ascending so adding the objectIds to the list like this tacks
    //them onto the end which is correct
    foreach (var oRule in oRules)
    {
        oRuleObjectIds.Add(oRule.ObjectId);
    }

    iPage++;
} while (oRules.Count>0);

//Make sure the rule we want to move is actually in the list before
//removing it and adding it in again at the top!
if (oRuleObjectIds.Exists(s => s == strObjectIdToMoveToTop))
{
    oRuleObjectIds.Remove(strObjectIdToMoveToTop);
    oRuleObjectIds.Insert(0,strObjectIdToMoveToTop);
}

//Reorder all the rules - this call can take a while with a lot of rules.
res = RoutingRule.UpdateOrderOfAllRoutingRules(_server, oRuleObjectIds.ToArray());
```

```
if (res.Success == false)
{
    Console.WriteLine(res);
}

Console.WriteLine("Rules reordered");
```

So to review, adding something to a generic list sticks it at the end of the list; you need to use .Insert with an index of 0 to stick something at the top. Generic lists are guaranteed to hold their order so if you're careful you can use them. Another technique is to use sortable containers of objects like Dictionaries and such to collect all your rules, order them using a custom sorting method etc… but that's a little heavy for what we need here.

NOTE: moving rules to the END of the list is generally not a good idea s this will stick it past the end of the "backstop rule" we mentioned earlier and will likely never get processed.

Tenants

[Tenants requires Unity Connection 10.0 or later]

Starting with Connection 10.0, the API supports the concept of creating a "tenant" object which includes a collection of objects such as schedules, phone systems, partitions, search spaces routing rules etc… that work together to create a subset of objects and users that can be easily partitioned off from other users/objects in the system. The idea being that you can create several tenants on a single Connection server that host users that should not interact with one another. So one tenant cannot directly dial or address messages to someone in another tenant for instance.

The curious reader perhaps ran off to look at the CUCA web administration interface for Unity Connection and did not find a "tenant" interface anywhere on there and is scratching their heads right now. Yes, there is no administration interface for these as they're intended for special markets that have provided their own administration interfaces for this. But since you're that extra special kind of person that's stepping up to making incredible applications using the SDK, you can quite easily create and manage tenants as well.

What is a Tenant?

In short a tenant is a collection of objects (partition, search space, schedule, phone system, routing rules, call handlers, interview handlers, name lookup handler, users, distribution lists, and a class of service) that allows you to create a mini "sub directory" within a Connection installation. Users assigned to a tenant can only address messages to other users in their same tenant – they cannot address, dial or know about other users or objects outside of their tenant definition.

Unity Connection has long had the ability to partition objects into groups like these using the partition and search space objects that have been there since Connection was released in version 2.0. This would allow the very ambitious administrator to actually host, say, multiple companies onto a single Unity Connection installation such that each had their own little auto attendant pond and their users could only address messages to other members of their company etc… The rather significant down side is managing this is enormously complex and the possibilities for mistakes by administrators creating new tenants and managing them manually is very high. Cleaning up objects associated with a tenant group is a nightmare given the foreign key dependencies that require you remove items in a very particular order or deletions would not be allowed. Really ugly for an administrator which explains why this was never a model that folks pursued.

Unity Connection 10.0 introduces a new Tenant "meta object" that helps enormously in creating, managing and, very importantly, cleaning up when removing collections of objects that make up a tenant definition. Partnered with your friend the .NET SDK you have the tiger by the tail when it comes to offering tenant type functionality in Unity Connection deployments.

What Happens When You Create a New Tenant?

Magic! Well, close to magic. A number of items are created for you and tied into the Tenant object allowing you to easily keep your segmented directory in order, add objects to tenants, find objects associated with tenants and, of course, neatly tidy up after yourself when you decide to remove a tenant. Using the SDK we can create a tenant in just a few lines of code and then look at what gets created for you on the back end:

```
Tenant oTenant;
res = Tenant.AddTenant(_server, "LindborgLabs", "LindborgLabs.com",
                       "Lindborg Labs Software",out oTenant);

if (res.Success == false)
{
    Console.WriteLine("Tenant creation failed:"+res);
```

```
    return;
}

Console.WriteLine("New tenant created:"+oTenant);
```

So, what happened on the back end there? Unity Connection made the following objects for you using the tenant alias you passed in ("LindborgLabs" in this case) as the basis for how they are named in the directory. Here's a breakdown of the objects created:

- **Partition** named "LindborgLabs_Partition_1".
- **Search Space** named "LindborgLabs_SearchSpace_1"
- **Phone System** named "LindborgLabs_PhoneSystem_1". Note that it does not create port groups or ports for you – that's left to you to handle.
- **3 Routing Rules** named "LindborgLabs_FRAttemptForward_1", "LindborgLabs_DRAttemptSignIn_1" and "LindborgLabs_RROpeningGreeting_1"
- **Class of Service** named "LindborgLabs_COS_1"
- **Schedule Set** named "LindborgLabs_Weekdays_1" along with associated schedules "LindborgLabs_Weekdays_1" and "LindborgLabs_Holiday_1".
- **User** named "LindborgLabs_Operator_1"
- **User Template** named "LindborgLabs_UserTemplate_1"
- **3 Call Handlers** named "LindborgLabs_OpeningGreetingCH_1", "LindborgLabs_GoodbyeCH_1" and "LindborgLabs_OperatorCH_1"
- **Call Handler Template** named "LindborgLabs_SystemCallhandlerTemplate_1"
- **Interview Handler** named "LindborgLabs_Interviewer_1"
- **Directory Handler** named "LindborgLabs_DirectoryHandler_1"
- **Public Distribution List** named "LindborgLabs_AllVoicemailUsers_1"

Wow, what value! That's a lot of action for a single create call – yes, this can take a few cycles so be sure your application is being friendly about not hogging the UI thread while it's busy.

Most of the items in the list are "tied" to the tenant by virtual of having their PartitionObjectId be the same as the Partition created by the tenant creation. This is, in fact, all that's necessary to create new users, lists and handlers in the tenant – simply create a new user and either pass in the PartitionObjectId off the tenant into it or change it after creation if you prefer. Easy.

Be aware, however, that when you delete a tenant, ALL items associated with the tenant are deleted. Let me repeat that. **ALL ITEMS tied to the partition (that entire list above) are removed in one shot**. Yes, this is very convenient and helps keep thing tidy. But it can burn you badly if you didn't mean to delete users (and all their messages and greetings etc…) because they were tied to the partition for a tenant. Be VERY careful when managing your tenants!

The keen observer will note that a few of the items included with a new tenant do not have partitions associated with them. Notably the Class of Service, Phone System (Media Switch) and Schedule are not tied to a partition. These are tied to the tenant only via a mapping table associated with the tenant object itself. Don't worry, the SDK wraps the methods for fetching these from the tenant and, when necessary, adding new ones to the tenant definition.

Finding and Fetching Tenants

To find a single tenant using its alias use this construction:

```
Tenant oTenant;
res = Tenant.GetTenant(out oTenant, _server, "", "LindborgLabs");

if (res.Success == false)
{
    Console.WriteLine("Failed to find tenant:"+res);
    return;
}

Console.WriteLine("New tenant found:"+oTenant);
```

For fetching a list of tenants and iterating over them, it would look like this if you wanted to fetch the first 10 (by default the first 20 are returned):

```
List<Tenant> oTenants;
res = Tenant.GetTenants(_server, out oTenants, 1, 10);
if (res.Success == false)
```

```
{
    Console.WriteLine("Failed fetching tenants:"+res);
    return;
}

foreach (var oTenant in oTenants)
{
    Console.WriteLine(oTenant.ToString());
}
```

Creating and Deleting Tenants

Note that there is no “update” of a basic tenant object. You cannot change the alias or STMP domain name on a standing tenant object. The only top level property you can change is the description string which hardly warrants an update path for the class. Adding and removing objects associated with the tenant is most of what you need to do with it and that’s covered in a bit here.

The following example simply creates a new tenant and then turns around and deletes it.

```
Tenant oTenant;
res = Tenant.AddTenant(_server, "SmithCo", "SmithCo.com","Smith Corp",out oTenant);

if (res.Success == false)
{
    Console.WriteLine("Failed to create tenant:" + res);
    return;
}

Console.WriteLine("New tenant created:" + oTenant);

res = oTenant.Delete();

if (res.Success == false)
{
    Console.WriteLine("Failed to delete tenant:"+res);
    return;
}

Console.WriteLine("Tenant deleted");
```

Note, again, that the single delete there removes around 18 objects in total (at least, assuming you’ve assigned no new users or handlers to the tenant). Remember that ALL objects associated with the tenant are removed.

Adding and Finding User Templates

By default there’s a single user template created that has the tenant’s COS, schedule, opening greeting references, operator call handler references etc… all buttoned up for you. When creating new users it’s a simple matter of using this template and the user is nicely tied in with your tenant, no questions asked. If you need to create additional templates for whatever reason, you can. However be sure to base those new templates off the one created by the tenant creation call so that all the default references are correct. If you do this manually on your own you will screw it up, I promise you. There are simply too many references to keep straight cleanly.

The Tenant object in the SDK has a simple helper method to return the list of user templates associated with the tenant for you without making you deal with the partition assignment filter query directly.

```
Tenant oTenant;
res = Tenant.GetTenant(out oTenant, _server, "", "LindborgLabs");
if (res.Success == false)
{
    Console.WriteLine("Failed to find tenant:"+res);
    return;
}

List<UserTemplate> oTemplates;
```

```
res = oTenant.GetUserTemplates(out oTemplates);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching templates:"+res);
    return;
}

foreach (var oTemplate in oTemplates)
{
    Console.WriteLine(oTemplate);
}
```

To add a new template, simply use the standard user template class construction to create a new one based on an existing one already associated with the tenant. So picking up where we left off above that would look like this:

```
//just use the first template in the list for the tenant to
//create a new user template
UserTemplate oNewTemplate;
res = UserTemplate.AddUserTemplate(_server, oTemplates[0].Alias, "NewTemplate",
    "New Templates", null, out oNewTemplate);

if (res.Success == false)
{
    Console.WriteLine("Failed to create new template:"+res);
    return;
}

Console.WriteLine("New template created");
```

Adding and Finding Call Handler Templates

Nearly identical to user templates, the call handler templates have a similar helper method for fetching all those associated with the tenant (i.e. those assigned to the same partition). I don't show the add process here, just the fetch since it's nearly a duplicate of the above user template example.

```
List<CallHandlerTemplate> oTemplates;
res = oTenant.GetCallHandlerTemplates(out oTemplates);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching templates:"+res);
    return;
}

foreach (var oTemplate in oTemplates)
{
    Console.WriteLine(oTemplate);
}
```

Adding and Finding Class of Services

Classes of service are a little trickier to do directly with the API since it requires a couple REST fetches before you can find the identifier for the Class of Service's that are tied to the tenant since Classes of Service do not have a simple partition assignment. But because the builders of the .NET SDK love you so much, the mechanism is nearly identical for fetching the list of Classes of Service as for other object types. *fist bump*

```
List<ClassOfService> oCoses;
res = oTenant.GetClassesOfService(out oCoses);
```

```
if (res.Success == false)
{
    Console.WriteLine("Failed fetching Coses:"+res);
    return;
}

foreach (var oCos in oCoses)
{
    Console.WriteLine(oCos);
}
```

Since Classes of Service don't have a partition assignment, creating new one and association it with the tenant requires you use the special method provided for that purpose off the Tenant class. The process of creating a new COS and adding it to the tenant looks like this:

```
ClassOfService oNewCos;
res = ClassOfService.AddClassOfService(_server, "New Cos", null, out oNewCos);
if (res.Success == false)
{
    Console.WriteLine("Failed creating new COS:"+res);
    return;
}

Console.WriteLine("COS Created:"+oNewCos);

res = oTenant.AddClassOfServiceToTenant(oNewCos.ObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed adding COS to tenant:"+res);
    return;
}

Console.WriteLine("COS added to tenant.");
```

Adding and Finding Schedules

Similar to the Class of Service, schedule have no partition assignment to work with so there using the fetching and setting methods provided off the Tenant class are necessary here. Note that you are fetching ScheculeSet objects here, these are tied to schedules and scheduleDetail objects. If you need to review, check out the Schedule section again. To fetch schedules off a tenant looks like this:

```
List<ScheduleSet> oSchedules;
res = oTenant.GetScheduleSets(out oSchedules);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching schedules:"+res);
    return;
}

foreach (var oSchedule in oSchedules)
{
    Console.WriteLine(oSchedule);
}
```

To create a new schedule and then associate it with a tenant you do it in a very similar way as you do with Classes of Service:

```
res = ScheduleSet.AddQuickSchedule(_server, "New Schedule Set",
    _server.PrimaryLocationObjectId, "",480, 1020, true, true,
    true, true, true, false, false);

if (res.Success == false)
{
    Console.WriteLine("Failed creating schedule:"+res);
    return;
}

Console.WriteLine("Schedule created");

res = oTenant.AddScheduleSetToTenant(res.ReturnedObjectId);

if (res.Success == false)
{
    Console.WriteLine("Failed adding schedule to tenant:"+res);
    return;
}

Console.WriteLine("Schedule added to tenant");
```

Finding Phone Systems

Note that there's no "adding" option here. A tenant is tied to a single phone system, it cannot span more than one phone system definitions. To fetch the phone system still presently returns a list since that's how it's stored in the associated mapping table.

```
List<PhoneSystem> oPhoneSystems;
res = oTenant.GetPhoneSystems(out oPhoneSystems);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch phone systems:"+res);
    return;
}

foreach (var oPhone in oPhoneSystems)
{
    Console.WriteLine(oPhoneSystems);
}
```

Adding and Finding Users

Finding users can be done by simply filtering for users assigned to the same partition the Tenant in question is assigned to. You can do this yourself using the "query=(PartitionObjectId is xxx)" construction yourself or you can just call the little helper method off the Tenant class which injects this into the URI for you along with the paging parameters. Not that it's UserBase classes that are returned since we're dealing with lists. If you're scratching your head right now, you can go review the User section again for an explanation of UserBase vs. UserFull classes.

```
List<UserBase> oUsers;
res = oTenant.GetUsers(out oUsers, 1, 30);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching users:"+res);
    return;
}

foreach (var oUser in oUsers)
{
```

```
    Console.WriteLine(oUser);
}
```

To add a new user is best done using the user template(s) provided off the tenant you want to add the user to. Yes, you can manually create a user and manually adjust their schedule, COS, partition, search space etc… to point where they need to for the tenant. Just don't. Use the templates off the tenant, that's what they're there for. The process would look like this:

```
//First, get the user templates off the tenant.
List<UserTemplate> oTemplates;
res = oTenant.GetUserTemplates(out oTemplates);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching user templates:"+res);
    return;
}

//use the first one to create a new user
UserFull oNewUser;
res = UserBase.AddUser(_server, oTemplates[0].Alias, "NewUser",
    "8381", null,out oNewUser);

if (res.Success == false)
{
    Console.WriteLine("Failed to create new user:"+res);
    return;
}

Console.WriteLine("User created:"+oNewUser);
```

Adding and Finding Call Handlers

Not surprisingly the call handler process is very similar to the user process for both fetching existing handlers assigned to the tenant and for adding new handlers for that tenant. One thing to note about call handlers in this case is that ALL call handlers are returned. This means both primary call handlers (tied to users) and system call handlers (i.e. "stand alone") are returned. The reason for this is currently the CUPI API only supports a single filter clause and to get the call handlers we must filter on the partitionObjectId matching the partition of the tenant. This does not allow for also filtering on the isPrimary flag for the call handlers. While LINQ makes it easy to fish only non primary handlers out of the return set, be aware that more data than you may want will be coming across the wire here.

```
//this gets ALL call handlers, including primary
List<CallHandler> oHandlers;
res = oTenant.GetCallHandlers(out oHandlers, 1, 30);
if (res.Success == false)
{
    Console.WriteLine("Failed fetching handlers:"+res);
    return;
}

//list all call handlers
foreach (var oHandler in oHandlers)
{
    Console.WriteLine(oHandler);
}

//filter list for system call handlers with LINQ
var oSystemHandlers = oHandlers.Where(oHandler => !oHandler.IsPrimary);

foreach (var oHandler in oSystemHandlers)
```

```
{
    Console.WriteLine(oHandler);
}
```

To add a new system call handler for a tenant you will again want to use a template off the Tenant object for this. By default one is created for you but as noted above you can add others. Using a template sets the partition, search space, switch assignment etc… always best to stick with this method for adding new handlers.

```
//get list of templates from tenant
List<CallHandlerTemplate> oTemplates;
res = oTenant.GetCallHandlerTemplates(out oTemplates);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching templates:"+res);
    return;
}

//use the first template to create a new call handler
CallHandler oNewHandler;
res = CallHandler.AddCallHandler(_server, oTemplates[0].ObjectId,
    "New call handler", "", null,out oNewHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed creating handler:"+res);
    return;
}

Console.WriteLine("New handler created:"+oNewHandler);
```

Adding and Finding Directory Handlers

By now the design pattern should be reasonably familiar. Here's how to fetch the directory handlers assigned to the same partition as the tenant:

```
List<DirectoryHandler> oHandlers;
res = oTenant.GetDirectoryHandlers(out oHandlers);

if (res.Success == false)
{
    Console.WriteLine("Failed to fetch directory handlers:"+res);
    return;
}

foreach (var oHandler in oHandlers)
{
    Console.WriteLine(oHandler);
}
```

To create a new directory handler there is no concept of a template to use, in this case it's merely a matter of creating a new directory handler and assigning its PartitionObjectId to the same partition as the tenant which requires an additional update step to complete:

```
DirectoryHandler oNewHandler;
res = DirectoryHandler.AddDirectoryHandler(_server, "New Dir Handler",
    false, null, out oNewHandler);

if (res.Success == false)
```

```
{
    Console.WriteLine("Failed creating handler:"+res);
    return;
}

Console.WriteLine("New directory handler created:"+oNewHandler);

oNewHandler.PartitionObjectId = oTenant.PartitionObjectId;
res = oNewHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("Handler update failed:"+res);
    return;
}

Console.WriteLine("New directory handler added to tenant");
```

Adding and Finding Interviewers

Rounding out the handlers management for tenants here, the interview handler follows the same design pattern as call and directory handlers, to no one's surprise. First, let's fetch the interviewers associated with a tenant – again, this is all interviewers assigned to the same partition that the tenant is.

```
List<InterviewHandler> oHandlers;
res = oTenant.GetInterviewHandlers(out oHandlers);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching handlers:"+res);
    return;
}

foreach (var oHandler in oHandlers)
{
    Console.WriteLine(oHandler);
}
```

To create a new interview handler there is no concept of a template to use, in this case it's merely a matter of creating a new interviewer and assigning its PartitionObjectId to the same partition as the tenant which requires an additional update step to complete:

```
InterviewHandler oNewHandler;
res = InterviewHandler.AddInterviewHandler(_server, "New interviewer",
    null, out oNewHandler);

if (res.Success == false)
{
    Console.WriteLine("Failed creating interviewer:"+res);
    return;
}

oNewHandler.PartitionObjectId = oTenant.PartitionObjectId;
res = oNewHandler.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating the interviewer:"+res);
    return;
}
```

```
Console.WriteLine("Interviewer created");
```

Adding and Finding Distribution Lists

Similar to handler and user interaction, public lists for a tenant can be fetched via a provided helper method off the class:

```
List<DistributionList> oLists;
res = oTenant.GetDistributionLists(out oLists, 1, 30);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching lists:"+res);
    return;
}

foreach (var oList in oLists)
{
    Console.WriteLine(oList);
}
```

Also similar to interview and directory handlers, lists are associated with tenants only via their PartitionObjectId assignment which requires a single additional update step to complete when adding new lists for a tenant.

```
DistributionList oNewList;
res = DistributionList.AddDistributionList(_server, "New list", "newList",
    "", null, out oNewList);

if (res.Success == false)
{
    Console.WriteLine("Failed creating new list:"+res);
    return;
}

oNewList.PartitionObjectId = oTenant.PartitionObjectId;
res = oNewList.Update();

if (res.Success == false)
{
    Console.WriteLine("Failed updating list:"+res);
    return;
}

Console.WriteLine("New list added to tenant");
```

Cross Launching CUCA Admin Web Pages

Unity Connection provides a lot of administration options for many, many objects in the database via the CUCA web administration interface. Often when building an application you may want to provide a small slice of custom functionality but you don't want to take on rebuilding the entire administration interface for a particular object (good luck with that if you want to go that route). One trick I leverage often in my customer applications is to provide an option to launch the CUCA interface directly to the object in question and let the user leverage the existing CUCA administration options directly. Since only a single frame from the interface is shown you can even get fancy and embed this page into your own interface if you like.

The CUCA admin interface provides for this using a feature called "cross launch". All this entails is a specially constructed URL that identifies the object type and the specific instance of that object you want to show. For instance if you want to show the admin page for a specific user on a server name "CUC91Test" the URL for that would look like this:

https://CUC91Test:8443/cuadmin/user.do?op=read&objectId=b083a973-c2a4-4373-aae9-34678ab08d32

of course the ObjectId there at the end identifies a specific user you want to show. You can then launch that page using a specific browser of just let Windows launch the URL in whatever browser the user has registered for default handling of web pages.

This can save you a ton of time when developing applications and can provide your users with a much nicer and fuller experience using your tools. Since there are a lot of object types and constructing the URLs for them all is a bit tedious I added a helper method for this in the ConnectrionServerRest class that can be used for this. The following code chunk shows how to use it and launch the page in the default browser.

```
//fetch user with alias of "jlindborg"
UserFull oUser;
WebCallResult res = UserBase.GetUser(out oUser, connectionServer, "","jlindborg");

if (res.Success == false)
{
    Console.WriteLine("Could not find user in database by alias=jlindborg");
    return;
}

//launch the user's admin page in CUCA using "cross launch" feature
string strUrl = connectionServer.GetCucaUrlForObject(ConnectionObjectType.User,
    oUser.ObjectId);

//launch in browser registered on the client
System.Diagnostics.Process.Start(strUrl);
```

How easy was that? Just about every object type you can get to in the CUCA administration pages is handled by the construction method so just about anything from routing rules to call handlers, interviewers, distribution lists and users can be accommodated. Sometimes I wish I could turn off the awesome, it can be a burden at times.

Dealing with Lists and ComboBoxes

If you're like me you have to throw up quick WinForms applications as proof of concept vehicles or distributable tools as the case may be. One of the items that comes up fairly often is the need to populate a list control of one type or another with items the user can select from such as which schedule set to apply to a user or which template to select when creating a new call handler etc. This comes up often enough that we added some sugar to the SDK to help deal with it quickly and efficiently. You'll find the ComboBoxHelper static class that's there to help you fill and fetch such lists easily - you can extend the design pattern to just about any type of list based object.

Here's a quick example of how to use it to present a list of tenants discussed above. It can be used with any object class that implements the IUnityDisplayInterface, which is most of them.

```
List<Tenant> oTenants;
var res = Tenant.GetTenants(GlobalItems.CurrentConnectionServer, out oTenants);
if (res.Success == false)
{
    Logger.Log("(error) fetching tenants:"+res,pDisplayInPopup:true);
    return;
}

ComboBoxHelper.FillDropDownWithObjects(this.comboBoxTenants,oTenants);
```

At this point the combo box is nicely filled in with your tenants shown with their "description" fields presented to the user in ascending sorted order. Various objects will use different fields for presentation which is defined by the IUnityDisplayInterface. Users present display names, alternate extensions display their DTMFaccessId field etc. Anyway, when you're ready to fetch the currently selected item from a list box you can get the fully populated object back without having to do another fetch, easy as falling off a chair:

```
Tenant oTenant;
if (!ComboBoxHelper.GetCurrentComboBoxSelectionId(comboBoxTenants, out oTenant))
{
    MessageBox.Show("Could not fetch tenant from combobox");
    return;
}
```

```
MessageBox.Show("Tenant selected=" + oTenant);
```

There's not a lot of reasons for the fetch to fail other than the combo box is empty, frankly, but safety first.

Sorting Lists of Objects

And while we're talking about IUnityDisplayIntervace lets talk about the generic case where you want to sort objects for output (reports, grid presentation, etc.). Any class that supports the IUnityDisplayInterface (again, most of them) allows for a simple sorting of any list using the UnityDisplayObjectCompare class defined in the interface.

This example fetches the first 20 call handlers in a system, display them unsorted as they come out of the database and then sorts them.

```
List<CallHandler> oHandlers;
res =CallHandler.GetCallHandlers(_server, out oHandlers, 1, 20);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching handlers:"+res);
    return;
}

Console.WriteLine("*** Unsorted ****");
foreach (var oHandler in oHandlers)
{
    Console.WriteLine(oHandler);
}

oHandlers.Sort(new UnityDisplayObjectCompare());

Console.WriteLine("*** Sorted ****");
foreach (var oHandler in oHandlers)
{
    Console.WriteLine(oHandler);
}
```

The sort is ascending only based on the display element for the object (users rely on the alias, most other objects use their display name).

Building Portals

A common need is for an application to be doing operations against Unity Connection using CUPI (Administrator rights needed) but allowing non administrators (i.e. end users) to authenticate/identify themselves to gain access to your application. A classic example is building a PIN/Password reset web service that limits which users they can reset PINs for. Say a company has local area admins and you want to allow them to go to a web page, sign in with their login and password from Connection and then be able to reset PINs for users only in their administrative group.

Let's just build the bones of that application here and walk through the steps – there's a simple ASP project showing how to do some of the basics via IIS if you're just getting started with web services (full disclosure: I don't "do pretty" – I'm nothing like a web jockey, it's a very basic example to get you started, nothing more!).

To be real clear, your application is logging into Connection for CUPI (i.e. full administrator) access, but folks using your portal do not need to be administrators – that's the point of the exercise here.

Step 1 – Authenticating Users for Access to Your Site.

The SDK makes this pretty easy. Collect the user name (alias) and the password (not the PIN, this only works for the GUI password for users in Connection – that's important) and you can validate the user and fetch their details in one step:

```
UserBase oUser;
if (connectionServer.ValidateUser("jlindborg", "ecsbulab",out oUser))
{
    Console.WriteLine("User validated:"+oUser.ObjectId);
}
```

```
else
{
    Console.WriteLine("User not validated");
}
```

If they failed to authenticate give them the bad news and kick them out – if they pass, open your new PIN reset application.

This works if our "jlindborg" user is a full administrator or just a user, doesn't matter – it validates that the alias is found and the password matches that account and if so, the UserBase object is passed back. It's important to note that there's no equivalent (currently) in the API for validating a user's PIN unfortunately – but for our purposes here the password is the appropriate credential to be using anyway. If you don't pass that last parameter in it does not do the secondary "GET" to fill in the user details – if you're validating a bunch of user credentials for some reason and don't want to DO anything with them, you do it more efficiently that way.

If you want to check for a Unity Connection role assignment for the user before letting them continue, you can check out the Roles and Policies section for an example of how to go about that.

Step 2 – Define the "Scope" of Users to Choose From

So, now that the user has accessed your super-cool PIN reset web site, you need to decide how you are going to determine WHICH users they can reset PINs for. You can simply allow them to reset anyone's PIN but that's probably not what you want since you can just give them the help desk Administrator role in Connection and have them do it through CUCA. More likely you want to be able to limit them to users in their specific administration area.

There's a couple different ways to approach this I've seen in the field:

1. Leverage distribution list membership. System administrators create public distribution lists that contain users for a particular administration scope (say a single city in a large distributed corporation). Such lists are typically created and maintained anyway as regional messaging is a common need (i.e. for facilities related notifications and the like). So once the user authenticates grab the list they are associated with and allow them to choose any user that's a member of that list.
2. Leverage partitions. If each administration area assigns their users (and lists and handlers etc…) to a specific partition for that area this can easily be leveraged. This is a popular choice and efficient. Since partition assignment can be added to templates and separate templates would be used for each administration area it's pretty easy to maintain moving forward as well (distribution list maintenance can be a headache by comparison). Once the user authenticates, get their partition and find all users associated with that partition and this defines the list of users they can reset passwords for.
3. Leverage search spaces. Similar to partitions but if you need area admins to reset PINs for users that can span Connection clusters, you'll want to use search spaces instead. Search spaces replicate around and you can have users in multiple clusters assigned to the same search space. Typically admin areas are smaller than that and tend to be local so typically partitions are ok to use. Partitions are easier to work with than search spaces (which can contain many partitions) but either will work fine.
4. Use editable fields like "Building", "City", "Department" and such for determining admin group association. This is certainly easy but risky given how easy it is to change or fat-finger a string entry into these fields.
5. Provide an external mapping table of aliases/extensions to group membership you call on your own. Many sites already have their own provisioning systems configured with such data and they can simply plug their interfaces into their web portal and go to town. If that's you, carry on!

There are, of course, other methods that can be used such as COS assignment, naming conventions or extension ranges or the like, but I consider them all to be poor ideas on the whole that don't scale well and are difficult to maintain. Partition or search space assignment is what I normally recommend to sites starting with a "greenfield" project since partition objects are small, simple and partitioning is what they're designed to do (it's in the name!). As such, that's what we'll show here.

Step 3 – Pick an Object to Edit

Using the partition assignment method of grouping area users this is just about trivial. One you authenticate your area admin the exercise is to generate a list of users that are assigned to that partition:

```
//fetch user with alias of "jlindborg" - he's our area administrator for the west
//wing of the Lindborg manor.
UserBase oLocalAdmin;
if (!connectionServer.ValidateUser("jlindborg", "ecsbulab", out oLocalAdmin))
{
    Console.WriteLine("Not authorized!  Buh, bye!");
    return;
}
```

```
//use the partition of the admin to gather a list of users they are allowed
//to reset PINs for.  All the users in their partition in other words.
List<UserBase> oAreaUsers;
string strQuery = string.Format("query=(PartitionObjectId is {0})",
oLocalAdmin.PartitionObjectId);

res = UserBase.GetUsers(connectionServer, out oAreaUsers, strQuery);

if (res.Success == false)
{
    Console.WriteLine("Failed fetching area users:"+res);
    return;
}
```

At this point you have a full list of users the area admin is allowed to work against – you can present them in a grid or a simple drop down or the like. The above implementation assumes a fairly contained set of users an area admin is responsible for. If we're talking about hundreds of users then this design does not work well – you would instead provide a way for the area admin to enter an alias or extension, find the user that had that alias or extension and make sure they were a member of the same partition. This isn't quite as slick as presenting a grid view or something similar but is considerably faster and does not involve dragging loads of data across the wire from Connection to populate a large list or having to deal with paging scenarios.

Step 4 – Perform Edit.

Once the area administrator has selected a user they can act on it's a simple matter of performing the edit you want to allow in your portal – in this case resetting the PIN. You can let them enter a random PIN but for password and PIN reset portals I always prefer designs that generate the PIN or password, reset it and present it to the admin to communicate back to the user either over the phone or via email. A randomly generated 5 digit number or a password consisting of two dictionary word entries (i.e. "SunnyPancake") are much nicer for admins to deal with than having to enter their own. They also don't leave the account vulnerable for a period of time with a trivial password until the user logs in and changes it.

For this run through we'll simply be resetting the PIN to a random 5 digit number, setting it to require the user change it on next login and moving on. We'll just continue the example from above:

```
//lets just say the area admin picked the first user in the list of those
//in his area to reset here.
UserBase oAreaUser = oAreaUsers[0];

//simple 6 digit random number generation - note that the PIN policy at your
//company may have issues with repeated digits or minimum length you need to
//deal with.
Random random = new Random();
int iNewPin = random.Next(100000, 999999);

//Clear any locked flag, set to require reset on next login
res =oAreaUser.ResetPin(iNewPin.ToString(), false, true);

if (res.Success == false)
{
    Console.WriteLine("PIN Reset failed:"+res);
    return;
}


Console.WriteLine("PIN reset to: "+iNewPin);
```

Obviously the above is a very crude CLI example and you'll be dealing with pretty web or GUI desktop applications or the like, but you can see the process here is pretty straight forward for providing limited/specific task access to users without requiring they be granted administrator access in Unity Connection. Anything you can do in CUPI can be offered up, of course, it's up to you to safeguard your site access and be careful with what you offer up to your "limited" administrators.

Revision History

Version 3.0.25 – 8/7/2013

- Packaged for deployment via NuGet repository.

Version 3.0.24 - 7/12/2013

- Exposed changed property list on classes that support it as a public ChangeList property.
- Added ability to check for specific items in the ChangeList by property name and value.
- Added ability to include contacts as distribution list members in the DistributionList class.
- Added support for compound query option for Unity Connection10.x releases that will support it.
- Reorganization of unit and integration tests, with accompanying fixes.

Version 3.0.22 - 6/8/2013

- Added optional out parameters for all notification device "Add" methods to return instances of newly created devices.
- Added UserLdap class for finding and importing users via LDAP integration.

Version 3.0.20 – 6/1/2013

- Renamed main class to ConnectionServerRest to avoid conflicts with the existing ConnectionServer class used for ODBC SDK. This allows REST and ODBC connectivity in the same application without trouble.
- Updated test project to implement test harness HTTP transport for additional test case coverage.
- Added clause parameter to AlternateExtensions, CallHAndlerTemplate, ClassOfService and Cluster classes for search parameter options.
- Fixed some search clause cases where some special characters were not being escaped out on the URI when searching.

Version 3.0.16 – 5/23/2013

- Reworked event model for debug and error events such that everything is registered off the ConnectrionServerRest class – the HTTPFunctions class is no longer exposed or necessary.
- Plumbed through JSON serialization events through the same error and debug events exposed off the ConnectrionServerRest object so it's no longer necessary to include JSON.NET in the projects that wish to monitor for serialization errors.
- Reworked the REST methods underneath the server class such that all necessary calls are exposed directly through the instance of the ConnectrionServerRest class. It's no longer necessary to make calls into the static HTTPFunctions class, everything is under "one roof" now.

Version 3.0.15 – 5/20/2013

- Finish rework of base classes to provide string based ENUMs for conversation names, subscriber conversation, contact rule types and greeting rule types. Updated numerous classes and test drivers to accommodate.
- Added additional routing rule and routing rule condition and comboBoxHelper tests.

Version 3.0.14 - 5/14/2013

- Added ComboBox helper class.
- Added IUnityDisplayInterface and included its implementation on many of the object classes in the SDK for list presentation options.
- Updated ConnectrionServerRest class to fetch server (cluster) information on login only for 9.0 and later Connection versions.
- Added option to ConnectrionServerRest login to authenticate as a user vs an administrator.
- Major rework of the HTTPFunctions class to simplify handling of session cookies for Connection 10.x and later builds.
- Updated session token logic in HTTPFunctions class to force expiration every 60 seconds regardless of traffic.

Version 3.0.11 – 5/5/2013

- Added UnityConnectionRestException section to help.
- Fixed CallHandlerTemplate creation logic to address missing MediaSwitchObjectId requirement.
- Added VmsServers list property to the ConnectrionServerRest class for easier handling of cluster configuraitons.
- Updated HTTPFunctions class to allow for adjusting timeout – defaults to 15 seconds, allows 1 to 99.
- Updated HTTPFunctions to throw UnityConnectionRestException with an error code of -1 for HTTP timeouts.
- Updated unit tests for all objects to handle UnityConnectionRestException throwing.

Version 3.0.10 – 4/28/2013

- Added interview handler question support for Unity Connection 10.0 and later.
- Added interview question audio upload methods

- Updated developers guide for 10.0 requirement notes and for using InterviewHandler class.
- Updated unit tests for more coverage of base classes
- Fixed error handling code for a few classes not handling empty strings passed into constructors.

Version 3.0.8 – 4/21/2013

- Added PostGreetingRecording and PostGreetingRecordingStreamFile classes for adding, fetching and editing post greeting recordings.
- Added Tenant class for Unity Connection 10.0 release
- Updated MailboxInfo class to include option for fetching inbox, sent items and deleted items counts.

Version 3.0.7 – 4/18/2013

- Added hooks to static HTTPFunctions class to get JSON parsing error events.
- Added hooks for DebugEvent and ErrorEvent off the HTTPFunctions class
- Updated unit tests to output debug information for missing properties by the JSON parser during testing.
- Added new LoadEachObjetTypeTest sub project for loading every class supported in the SDK while monitoring JSON parsing errors.
- Added InterviewQuestion class and wired up a lazy fetch for them in the InterviewHandler class.
- Updated many of the classes with results from new testing
- Updated HTTP functions class to handle new cookie changes coming in Unity Connection 10.0 release.
- Updated directory handler class to allow for creation of new directory handlers.

Version 3.0.4 – 4/11/2013

- Massive update of SDK including naming convention updates – some adjustments from previous projects may be necessary
- Switched from XML payloads to JSON for all REST calls with a few exceptions.
- Added JSON.NET as project dependency
- Reconfigured solution to automatically include NuGet dependencies when rebuilding if they've been removed.
- Added Media Switch Related classes and accompanying phone system documentation in the developers guide.
- Added many more operations to the test suite
- Added VmsServer class
- Added RtpCodecDef class
- Updated ConnectrionServerRest class to do lazy fetches of primary location and VMSServer references directly.

Version 2.0.15 - 3/22/2013

- Updated development guide to add more examples for user messages, filled in the system contacts and name lookup handler sections.
- Moved City, Department and EmployeeId fields from the UserFull to UserBase definition.

Version 2.0.14 – 3/15/2013

- Major rework of UserMessage class – adding reply, reply all, delete, methods to clear deleted items folder, working on forwarding (not complete yet) as well as the ability to update the read status of messages and the subject line from an instance or static call.
- Updating of internal variable naming conventions
- Code cleanup in HTTPFunctions and ConnectrionServerRest classes to remove unused methods and add additional comments.

Version 2.0.13 – 3/12/2013

- Exposed CreateMessage methods for both local WAV file and resourceId for CUTI interfaces off the UserMessage class.

Version 2.0.12 – 3/8/2013

- Added ability to pass in alternate user ObjectId to phone interface's message playback function so administrator logins with mailbox proxy access can use CUTI for playback.
- Added PrimaryLocationObjectId property off the ConnectrionServerRest class.
- Added LocationDestinationTypes enum for code readability.
- Added option to build URL for cross launch of CUCA pages into the ConnectrionServerRest class

Version 2.0.9 – 2/21/2013

- Added "ValidateUser" methods to the ConnectrionServerRest class for easy alias/password validation of users – targeted at "portal" type administrative applications.
- Updated Role class to include a quick option to get the ObjectId of a role by its name.

Version 2.0.8 – 2/17/2013

- Updated most object classes to return “true” on a fetch of a list of objects that returns no results along with an empty list instead of “false” and a null reference for ease of use.
- Updated developers guide for public distribution list functions.
- Changed formatting of PDF output on developers guide to align better for margins.

Version 2.0.7 – 2/14/2013

- Added NotificationTemplate class to fetch HTML device templates.
- Updated NotificationDevice class to allow for creation of new HTML notification devices.
- Added ConversationName enum to ConnectionTypes class.
- More updates to dev guide.

Version 2.0.6 - 2/13/2013

- Updated InstalledLanguage class (and test) to be more consistent with the rest of the library
- Updated CallHandler to not return error when no results on a query fetch were returned.
- Updated Schedule and TimeZone classes to support DumpAllProps and a few code comment fixes.
- More dev guide updates

Version 2.0.5 – 2/12/2013

- Added create, delete and update capabilities to the ScheduleSet class.
- Added create, delete and update capabilities to the Schedule class.

Version 2.0.4 – 2/11/2013

- Fixed some items in the sorting mechanism for user objects such that empty first/last/display name elements are sorted correctly.
- Fixed MWI and NotifcationDevice classes to set the “Active” property correctly on new object creation
- Updated ToString override for RestrictionPattern to show blocked property for each pattern.
- Updated UserMessage class to include instance methods for fetching attachments and counts.
- Added first draft of developers guide into the project

Version 2.0.2 – 2/10/2013

- Added remaining unit tests for new classes brining the library code coverage up to %90 when tested against Connection 9.1
- Fixed various bugs in wrapped classes as part of the unit testing effort.

Version 2.0.1 - 1/16/2012

- First more or less “full” release of SDK functionality covering CUPI for administrators and the most common CUTI functionality. CUMI coverage is minimal at this point.
- This completely supersedes the 1.x version of the CUPI REST library previously available.