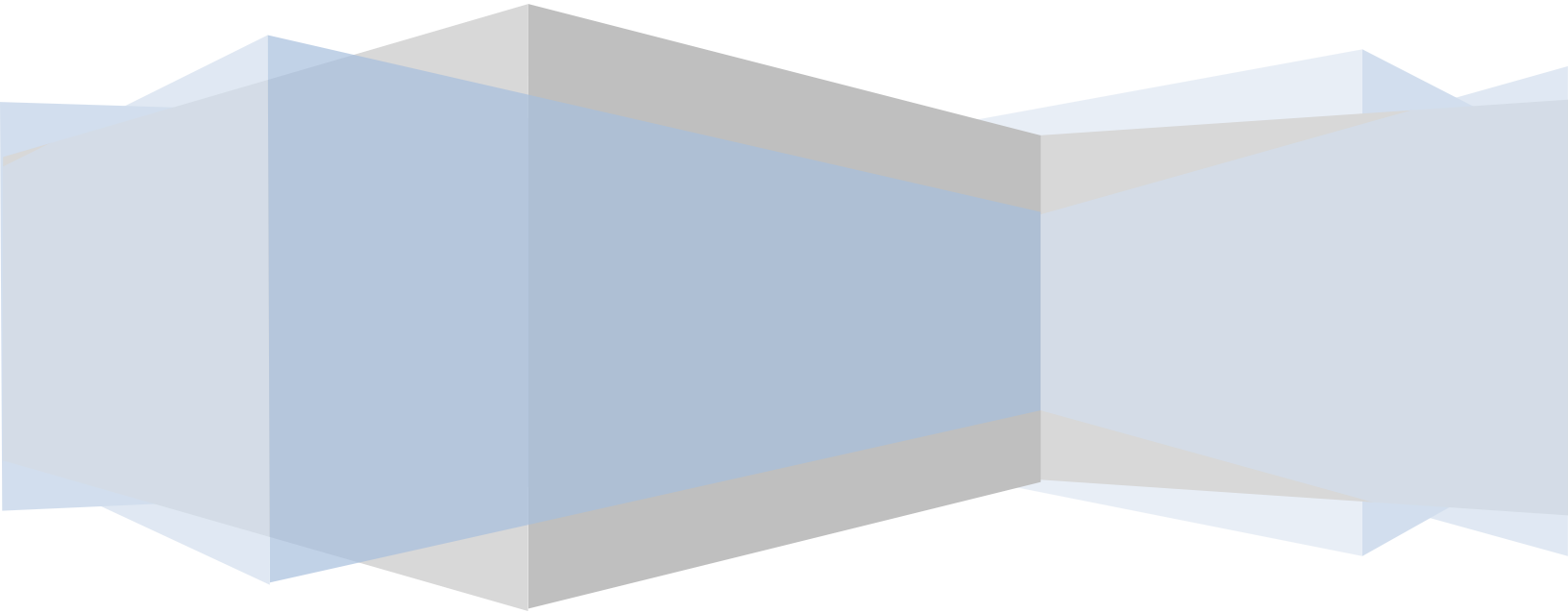
Cisco Systems

Cisco Unity Connection Java REST SDK

For END USER APIs

Utkarsh Katiyar



Contents

Overview

This library is to be used for the development of the Unity Connection REST API based applications. It liberates the application developer from the basic operations like making request, hitting the API URL, parsing the response etc. It has all that functionality wrapped-up inside easy-to-use java objects which enables the developer to perform some of critical API operations with minimum complexity.

It includes all the operations supported by the Connection REST interface related to the END USER (subscriber), like editing basic fields, alternate names, private lists, private list members, html devices etc. For Admin level APIs, there is a separate library.

Integration

The library covers the functionality provided in different Unity Connection releases starting from 8.6. Although the library takes care of the backward compatibility but some of the functionalities might throw error if used with a non-supported release. The library has been tested with all the releases past 8.6 and some of the functionalities don't work or behave differently in different releases. Example: Message count functionality has been introduced post 9.1 so it will not work in the earlier releases.

The library is a jar file which can be included in the java application and the functionalities can be used. The library comes with complete java-docs which help developers and provide enough information while using code-assist features of different IDEs.

This is an open source library so you can fiddle up with the base functionalities as per the requirement, if you want to.

A sample eclipse project is included in the subversion which makes use of the library and has some example classes demonstrating some basic operations. It also includes all the used libraries (http-client, logging, etc.). For downloading the project, you might want to install SVN client (subeclipse plug-in in case of eclipse). The read-only credentials are provided on the web-page.

NOTE: This library uses 3.x version of apache-http-client.

The library

The library intends to wrap the END User API functions. It is divided into different classes related to different END User related objects. Almost all the functions are available in both static and member functions to the object so it can be called upon both ways. Means, you have a method getGreetings() which can be called on a user object and also there is a static method Greeting.getGreetings() which also serves the same purpose.

Some useful features:

1. Every object has a method named *clearPendingChanges*() which can be used to clear up a dirty object.
   Example: If a user update was desirable and the update operation didn't succeed, then to maintain the local state of the object this function can be called.

```
//change the user attribute in the local object.
user.setAltLastName("TestLast");
//send it for update
WebCallResult res = user.update();
if(res.isSuccess()){
      ///Update is successful
}
else{
      //update failed so maintain the original state.
      user.clearPendingChanges();
}
```

2. Each object has its *toString()* method overridden to provide a string which holds some key attributes of that.

3. There is a method named *dumpAllProperties,* available with all the objects which print the properties of an object.

```
user.dumpAllProperties("->");
```

4. The library supports both local and phone recordings which can be assigned to the voice names and greetings of different objects.

5. There is complete messaging support (CUMI) for the user in the library. Various messaging operations like, message counts in different folders (inbox, deleted, sent), sending message, changing message properties etc. are supported.

6. There are 2 types of objects for most user elements. One of those is for an individual element and the other is for a group of those elements.

Example: There are 2 types of alternate device objects. One is *"AlternateDevices"* and the other is *"AlternateDevice"*. AlternateDevice is the object which holds the properties of an individual alternate device whereas AlternateDevices is the object which holds a group of alternate devices.

Alternate devices for the user can be fetched like this:

```
//Fetch all alternate devices for the user using member function in
user construct
AlternateDevices devices = user.getAlternateDevices();
log.info(devices);
```

There is a static construct also available for the same operation

```
//Fetch all alternate devices for the user using static functions
AlternateDevices devices = AlternateDevice.getAlternateDevices(server);
log.info(devices);
```

AlternateDevices object contains a list which has details for the various alternate devices for the user. It can be traversed to get the alternate devices.

```
//Traverse through the fetched devices
List<AlternateDevice> deviceList = devices.getAlternateDevice();
for(AlternateDevice device : deviceList){
      log.info(device);
}
```

Both types of objects exist for most of the user related objects and each of it follows a similar design pattern for CRUD operations.

Now let's go through the different key objects in the library. This document tries to capture as much information and description of the functionalities for those. All the functions and other key information are present in the java-docs.

ConnectionServer

This object contains the attributes related to the unity connection server on which the REST requests need to be sent. All the operations (static) require the server object to be passed as the parameter. A ConnectionServer object can be created like this:

```
//Create connection server object.
            ConnectionServer server = null;
            try{
                  server = new ConnectionServer("test.cisco.com", "user",
"UserPassword");
            }
            catch(Exception ex){
                  //error handling code
            }
```

Creating a connection server instance internally sends a request to fetch the connection version which ensures that the details provided for the server are valid.

There are various functions available with this object which can be helpful. One of such function is `isConnectSuccessful` which returns a boolean value indicating whether the connection to the server was successful or not.

Mostly one wants to check which connection version, the library is being attached to. The connection server object provides a function for this.

```
//Validate connection version
            if (!server.isVersionAtLeast(9,1,0))
            {
                log.info("Version of 9.1.0 or later is required, current
version is " + server.getVersion());
                return;
            }
            else{
                  log.info("Version is fine::" + server.getVersion());
            }
```

The method is overloaded in a way that it can perform checks on various release attributes like major version, minor version, revision, build number.

```
// Example:
server.isVersionAtLeast(10) -  validate major version
server.isVersionAtLeast(10,0) -  validate major and minor version
server.isVersionAtLeast(10,0,0) -  validate major, minor version and the
revision
server.isVersionAtLeast(10,0,0,23) -  validate major, minor version, revision
and build number
```

WebCallResult

This is the class which holds various elements involved in the request to the server and the response, like: response code, response text, error code, error message etc. All the objects have this class as a member which gets set when any operation is performed on that object.

```
//Fetch user information
User user = User.getUser(server);

if(user.getWebCallResult().isSuccess()){
      user.dumpAllProperties("-->");
}
else{
      log.info("Failed in fetching user information::" +
user.getWebCallResult());
      return;
}
```

Most of the functions in the library (almost all updates and deletes) return this object which holds the result of the operation and some other information.

```
user.setAltFirstName("TestFirstName");
WebCallResult res = user.update();
if(res.isSuccess()){
      log.info("User updated successfully");
}
else{
      log.info(res);
}
```

WebCallResult can provide enough information for debugging perspective and mostly it will be used as a holding object for the HTTP request/response attributes. As described earlier, the *toString()* method of this class is overridden, so on printing the object it will show you the result(success/failure), error code and error message. The object also has *dumpAllProperties* method, which can be very helpful in debugging if something goes wrong.

A sample output of *dumpAllProperties* on a failed operation would be something like this:

```
-->Success=false

-->ResponseCode=400

-->ErrorCode=DATA_EXCEPTION

-->ErrorMessage=Duplicate extension in partition: ObjectName =
[EmbeddedKey=CustomType.ObjectType.21], DtmfAccessId = 1048, OwnerObjectId =
856be091-30a9-45dd-9bcc-7f8e4444000f
```

PagingData

WebCallResult construct also contain an attribute which is called paging data. With subscriber APIs, it's only applicable in case of message listing. This construct provides all the paging elements, which will be helpful in designing applications with different paging information. The user need to provide records per page and page number attributes to the API call and it will calculate all the other attributes based on the result. It contains elements such as, records per page, current page number, total pages, total records, next page number, previous page number, start index on this page, end index on this page etc. The example of this is provided in Messaging section.

PhoneRecording

A user can create recordings on a phone and assign that recording to its voice name or greetings. Using REST APIs, this can be a multi-step process which is simplified using this construct.

The constructor takes a server object and the extension on which the call needs to be placed.

```
//Create a phone recording object
      PhoneRecording pc = new PhoneRecording(server, "1040");
```

Create a recording

PhoneRecording object can be used now for recording a stream file. It will place the call on the provided extension and then recording can be done.

```
WebCallResult res = pc.recordStreamFile();
if(res.isSuccess()){
      System.out.println(pc);
}
else{
      System.out.println(res);
      return ;
}
```

After the completion of this step, the elements of the PhoneRecording object are updated with the actual values like callId, resourceId (which is the name of actual wav file on the server) etc.

Now this object can be passed in the functions for updating voice name or greetings of the user, which we'll see in the later sections.

Play recording

To play this recording, there is a function available in this construct.

```
CallControl ctl = pc.playStreamFile();
	WebCallResult res = ctl.getWebCallResult();
	if(res.isSuccess()){
		System.out.println(pc);
	}
	else{
		System.out.println(res);
		return ;
	}
```

A CallControl object is returned which contains various attributes like speed, volume, start position etc.

User

As the library is for the END USER APIs, this is the base class for the library. All the other operations are dependent on a user object. Although those operations can be performed individually using the static operations on their base classes but mostly you might find the need to initiate those from a user object (The concept of static and member functions).

A user object can be created using the getUser() method and passing the connection server object to it.

```
//Fetch user information
User user = User.getUser(server);


if(user.getWebCallResult().isSuccess()){
	user.dumpAllProperties("-->");
}
else{
	log.info("Failed::" + user.getWebCallResult());
}
```

This will fetch you a user object which holds all the basic attributes for it. All the attributes and related objects for the user can be worked directly from this user object.

Example: If the user's alternate devices need to be fetched, can be done in this fashion:

```
//Fetch all alternate devices for the user using member function
in user construct
AlternateDevices devices = user.getAlternateDevices();
log.info(devices);
```

Voice name

If there is a voice name present for a user, it can be fetched and stored in a local wav file.

```
WebCallResult res = user.getVoiceName("C:/test/server_file.wav");
```

Voice name for the user can be updated using both local and phone recordings.

Using local recording:

```
WebCallResult res = user.updateVoiceName("C:/test/local_file.wav");
```

Using phone recording: a phone recording object can be obtained as described in section PhoneRecording.

```
WebCallResult res = user.updateVoiceName(pc);
```

PIN and Password

Every user can have a PIN (Personal Identification Number) and password (used to login to web interfaces like web-inbox). To reset the PIN and password there are methods provided in the User construct.

Reset PIN:

```
WebCallResult res = user.resetPin(“123456”);
```

Reset Password:

```
WebCallResult res = user.resetPassword(“oldpassword”, “newpassword”);
```

NOTE: The PIN and Password are validated as per the authentication rules defined by the administrators.

Greeting

Every user is associated with a primary call handler and every call handler has 7 greetings with it: Alternate, Holiday, Internal, Busy, Off Hours, Standard and Error.

By default, Standard greeting is enabled. Any of those greetings cannot be deleted and no new greeting can be created.

Fetch user greetings

Individual greeting can be fetched by using member function of user object or the static function in the Greeting construct. The greeting name to be passed is case insensitive. To avoid error scenarios use GreetingType enum for passing greeting type into different functions.

```
Greeting g = user.getGreeting(GreetingType.ALTERNATE.value());
if(g.getWebCallResult().isSuccess()){
      g.dumpAllProperties("-->");
}
else{
      g.getWebCallResult().dumpAllProperties("-->");
```

```
}
```

Disable greeting

Standard and Error greetings are always enabled and cannot be disabled. Other greetings can be disabled by using member or static function of Greeting construct.

```
//Disable greeting
WebCallResult res = g.disable();
```

Enable greeting

The greetings can be enabled using member or static function of Greeting construct.

A greeting can be enabled either till a particular date or forever. The method of the Greeting construct is overloaded to support both.

```
//To enable a greeting forever
WebCallResult res = g.enable();

//To enable a greeting till a particular date
//The date which is passed in the function should be
//a UTC date. So whatever date is there as per your system
//convert it to UTC and then pass it to the function
Date d = new Date(); ///The date as per your system timezone
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
df.setTimeZone(TimeZone.getTimeZone("GMT"));
String s = df.format(d);
WebCallResult res = g.enable(df.parse(s));
```

Customized greeting recording

All the greetings can have recordings which are played to the caller when that particular greeting is at work. The recording can be done in multiple languages which provide the functionality to play the recording to the user, in the language for which the call is set to. The library supports both local and phone recordings to be assigned as a greeting recording.

"A greeting can be recorded in multiple languages even when the language is not installed on the connection server but that greeting recording will never play, unless the language is installed."

To update the greeting recording the member function on the user or greeting construct can be used. Also there is a static method in the greeting construct for the same.

```
//Update greeting recording using a local file
```

```
WebCallResult res =
greeting.updateRecording(LanguageCodes.EnglishUnitedStates.value(),
“C:/greeting_recording_server.wav”);


//Update greeting recording using a phone recording
WebCallResult res =
greeting.updateRecording(LanguageCodes.EnglishUnitedStates.value(),
pc);
//Here pc is a PhoneRecording object which can be obtained as described
in section PhoneRecording
```

The language code can be passed as an int, but it’s always better to use LanguageCodes Enum to avoid errors.

The recording for a particular greeting on the server can be fetched on the local machine.

```
//Fetch greeting recording
WebCallResult res =
greeting.getRecording(LanguageCodes.EnglishUnitedStates.value(),
“C:/greeting_recording_local.wav”);
```

After updating the recording the PlayWhat property of the greeting need to be changed to *RecordedGreeting*, to make the custom recording in effect.

Update greeting

A subscriber can update the *PlayWhat* property of the greeting in addition to the enable/disable status. This property can be updated using the member or static function of Greeting construct. The function accepts the value of PlayWhat property as an int.

```
greeting.setPlayWhat(PlayWhatType.RecordedGreeting.value());
WebCallResult res = greeting.update();
```

Although the function accepts an int, but it’s always better to use PlayWhatType Enum to avoid errors.

Messaging

The library provides a thorough messaging support for the subscriber using Message construct. Various messaging related operations can be performed using this construct.

Fetch message list

Fetching message list functionality has several variants in the library. There are member and static functions available in User and static functions in Message constructs. We will see the member functions in the user construct here.

```
	//Fetch messages for the user
	Messages msgList = user.getMessages();
```

This call will fetch the messages from “inbox” folder with criteria as 20 messages per page and page number 1. There are different calls available for this functionality where different folder types, rows per page and page number parameters can be passed. The most useful variant of this function is the one with all of those and message search options which include search and sort criteria.

The call will be something like this:

```
	//Fetch unread-urgent voice messages for the user in descending order
	//of arrival time, means starting with the newest.
	MessageOptions opt = new MessageOptions();
	opt.setPriority(PriorityType.URGENT);
	opt.setRead(false);
	opt.setType(MessageType.VOICE);
	opt.setSortType(MessageSortType.SORT_NEW_FIRST);
	int maxMessagesPerPage = 5;
	int pageNumber = 1;
	Messages msgList = user.getMessages(FolderType.INBOX.value(),
maxMessagesPerPage, pageNumber, opt);
```

This call will produce a list which has unread-urgent voice messages with newest first and so on and also will apply the paging criteria to it. The webcallresult object contained by the msgList object here will have the pagingData object having all the paging information which can be helpful in displaying paging information on an application.
An example is here which makes use of the paging data and traverses through all the pages and fetches message lists on those.

```
	boolean isNext = false;
	int pageNumber = 1;
	int messagesPerPage = 2;
	do{
		Messages list = user.getMessages(FolderType.INBOX.value(),
messagesPerPage, pageNumber);
		WebCallResult res = list.getWebCallResult();
		PagingData data = res.getPagingdata();
		if(data.getNextPage() != -1){
			isNext = true;
			pageNumber = data.getNextPage();
		}
```

```
            else{
                  isNext = false;
            }

            System.out.println("Displaying messages::(" +
data.getStartIndex() + " - " + data.getEndIndex() + ") of " +
data.getTotalRecords());
            for(Message msg : list.getMessage()){
                  System.out.println(msg);
            }
            System.out.println();
      }while(isNext);
```

A sample output of this example would be like this:

```
Displaying messages::(1 - 2) of 6
[Subject=Test Mail] [From=test] [Duration=15.34s] [Type=VOICE]
[Subject=Test Mail_1] [From=test] [Duration=15.34s] [Type=VOICE]

Displaying messages::(3 - 4) of 6
[Subject=Test Mail_2] [From=test] [Duration=15.34s] [Type=VOICE]
[Subject=Test Mail_3] [From=test] [Duration=15.34s] [Type=VOICE]

Displaying messages::(5 - 6) of 6
[Subject=Test Mail_4] [From=test] [Duration=15.34s] [Type=VOICE]
[Subject=Test Mail_5] [From=test] [Duration=15.34s] [Type=VOICE]
```

Fetch message count

Although message listing provides enough information about the counts and all but if someone is interested only in getting count, then there is no point in putting the complete list on the wire. There are separate functions available for fetching the message counts from different folders.

```
      //Fetch message count for inbox folder
      BigInteger count = user.getMessageCount(FolderType.INBOX.value());
      System.out.println("Message count=" + count);
```

Send message

Sending a message involves a couple of things to be done before actually sending a message. We will have a look at those one by one.

Setting message properties: First step is to set message properties like, priority, sensitivity etc. Those properties are set in a Map which can be passed in the send message function.

```
      //To set other attributes of the message use the property map
      Map<String, String> props = new HashMap<String, String>();
      props.put("Priority", PriorityType.URGENT.toString());
```

Creating recipient list: This step results in an "Addresses" object containing different recipients in the form of "Address" objects. This object can be created by creating the objects manually or obtained by making a Directory search as explained in Directory section. Let's see how can it be created manually.

```
//First recipient
Address add1 = new Address();
add.setType(AddressType.SUBSCRIBER);
add.setSmtpAddress("rec1@foo.com");

//Second recipient
Address add2 = new Address();
add.setType(AddressType.SUBSCRIBER);
add.setSmtpAddress("rec2@foo.com");

//Prepare the list of recipients
Addresses list = new Addresses();
list.getAddress().add(add1);
list.getAddress().add(add2);
```

This *list* object can now be set as TO, CC, or BCC for the message.

Creating recordings: This step is optional as its not mandatory to always send a recording with the message but sending a voice mail without any voice wouldn't make any sense. The recordings can be local wav files or phone recordings. You can send as many recordings with one message.

After those steps the message is now ready to be sent. The message call is using the static function in message construct.

```
//Send message using recorded wav files as attachments
String recordings[] = {"C:/recording1.wav"};
WebCallResult res = Message.sendMessage(server, "Test Message",
"sender@foo.com", list, null, null, props, recordings);
```

The method is overloaded to accept phone recordings as input also.

```
//Send message using phone recordings as attachments
PhoneRecording recordings[] = {pc};
WebCallResult res = Message.sendMessage(server, "Test Message",
"sender@foo.com", list, null, null, props, recordings);
```

In case you don't want to send any recording with the message, pass the recording parameter as null.

```
//Send message without any voice attachments
WebCallResult res = Message.sendMessage(server, "Test Message",
"sender@foo.com", list, null, null, props, null);
```

Fetch message details

The message object obtained from the message list contains all the high level attributes of the message but there are some low level details like Recipient Information, Attachments etc are not there in it. To get this information, a function call is available which takes message id as input and fetches the Message object having all the details.

```
//Fetch message details
Message fullmsg = Message.getMessage(server, msg.getMsgId());
```

Update message properties

A message can be updated by marking it read/unread or changing its subject. Those functionalities can be achieved by using either the member or static function in the message construct.

```
//Mark message as Read, Here msg is a Message object
//fetched from a list or using details function
msg.setRead(true);
WebCallResult res = msg.update();
```

Delete message

A message delete/undelete/permanent delete is provided using member or static functions on the message construct. A delete operation on a message is dependent upon the COS setting for deleted messages. Its decided by the server itself based on this setting whether the message is hard deleted or sent to deleted folder, unless a permanent delete is used.

```
//Delete a message, Here msg is a Message object
//fetched from a list or using details
WebCallResult res = msg.delete();
```

A deleted message if it goes to deleted folder, can be undeleted (sent back to the inbox folder).

```
//UnDelete a message, Here msg is a Message object
//fetched from a list or using details
WebCallResult res = msg.undelete();
```

A message can be permanently deleted without sending it to a deleted folder, irrespective of the COS setting.

```
//Hard delete a message, Here msg is a Message object
//fetched from a list or using details
WebCallResult res = msg.deletePermanently();
```

CometNotification

The user can receive Comet notifications for message operations on the Inbox and Deleted Items folders by subscribing to those events. The functionality provided in this library subscribes the user for all the events (listed below), which will be received as and when they arrive.

Subscribe

To subscribe for notifications member function of the user construct or the Static function of the CometNotification construct can be used.

```
	//Subscribe for event notifications
	CometNotification notification = user.subscribe();
	if(notification.getWebCallResult().isSuccess()){
			System.out.println("Subscription ID:" +
notification.getSubscriptionId());
	}
```

The subscription process will also take care of the various steps required to establish connection to Jetty server like handshake, connect.

Receive events

After the subscription, one would need to send re-connects towards the server which will gather events if there are any.

```
	//Repetitively send connect requests towards the server
	//for gathering events
	while(true){
	api.MessageEvent event = CometNotification.receiveEvents(server,
notification);
		if(event.getWebCallResult().isSuccess()){
			if(event.getUsn() != null){
				System.out.println("An event is received");
				event.dumpAllProperties("-->");
			}
			else{
				System.out.println("NO event");
			}
		}
		else{
			System.out.println("Failed to receive events");
			System.out.println(event.getWebCallResult());
		}
	}
```

An event is wrapped inside a MessageEvent construct which can have one or more MessageInfo objects.

MessageEvent defines some of the basic properties of the notification event such as:

Display Name	Description
SubscriptionId	Subscription ID
USN	Message USN
EventTime	Time of Event
EventType	Type of Event
MailboxId	Mailbox ID of the user

EventType can have following values:

EventType	Description
NEW_MESSAGE	This event is sent when a new message arrives.
SAVED_MESSAGE	This message is sent when a message is marked read.
UNREAD_MESSAGE	This event is sent when a message is marked as unread.
DELETED_MESSAGE	This event is sent when a message is deleted.

MessageInfo has attributes related to the message for which the notification event was generated. It can have following values of the message:

- MessageId
- CallerAni
- MsgType
- Priority
- ReceiveTime
- Sender

Un-subscribe

To un-subscribe from the notification event requests, the member function of the user construct or static function of the ComentNotification construct can be used.

```
//Unsubscribe
WebCallResult res = user.unsubscribe(notification.getSubscriptionId());
System.out.println(res);
```

TransferOption

The primary call handler associated with the user gives the feature of transfer options which can either place a call on the phone (ring the phone) or send the call to active greeting rule. There are 3 transfer options available: Standard, Alternate and Off Hours. Those can't be deleted or new ones can't be created.

Fetch transfer option

Individual transfer option can be fetched by using member function of user object or the static function in the TransferOption construct. The transfer option type to be passed is case insensitive. To avoid error scenarios use TransferOptionType Enum for passing into different functions.

```
	TransferOption opt = user.get TransferOption
(TransferOptionType.ALTERNATE.value());
	if(opt.getWebCallResult().isSuccess()){
		opt.dumpAllProperties("-->");
	}
	else{
		opt.getWebCallResult().dumpAllProperties("-->");
	}
```

Disable transfer option

Standard transfer option is always enabled and cannot be disabled. Other transfer options can be disabled by using member or static function of TransferOption construct or the member function of user object.

```
	//Disable transfer option
	WebCallResult res = opt.disable();
```

Enable transfer option

Transfer options can be enabled using member or static function of TransferOption construct or the member function of the user object.

A transfer option can be enabled either till a particular date or forever. The method of the TransferOption construct is overloaded to support both.

```
//To enable a transfer option forever
WebCallResult res = opt.enable();

//To enable a transfer option till a particular date
//The date which is passed in the function should be
//a UTC date. So whatever date is there as per your system
//convert it to UTC and then pass it to the function
Date d = new Date(); ///The date as per your system timezone
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
df.setTimeZone(TimeZone.getTimeZone("GMT"));
String s = df.format(d);
```

```
WebCallResult res = opt.enable(df.parse(s));
```

Update transfer option

A subscriber can update various properties of a transfer option. Here in the example the transfer option is updated to make it perform a supervised transfer to extension 123456, when in use.

```
//To update a transfer option
opt.setExtension("123456");
opt.setAction(TransferActionType.Transfer.value());
opt.setTransferType(TransferType.Supervised.value());
WebCallResult res = opt.update();
```

Directory

Library also provides functionality to search the addressable objects that can be used to send messages, to be added to private list etc. The objects can be searched by name, extension or both and matching criteria can be exact match or starts with.

A search returns an *Addresses* object which contains a list of all the addressable objects in the form of *Address* objects.

If all the objects having names starting with “User” need to be searched, can be done in this fashion:

```
Addresses add = Directory.getAddressableObjects(server,
Directory.SearchType.STARTSWITH, Directory.SearchBy.NAME, "User");
            if(add.getWebCallResult().isSuccess()){
                  List<Address> list = add.getAddress();
                  for(Address a: list){
                        a.dumpAllProperties("-->");
                  }
            }
            else{
                  add.getWebCallResult().dumpAllProperties("-->");
            }
```

If all those objects need to be searched which are having name or extension starting with 1, it can be done in this fashion.

```
Addresses add = Directory.getAddressableObjects(server,
Directory.SearchType.STARTSWITH, null, "1");
```

AlternateName

There is provision for every user to have a first and last name in the system. A user can have alternate first and last names. Those are the classes which deal with the alternate names of the user.

Fetch the list of alternate names

To fetch alternate names for the user 2 constructs are available:

Get the list from the user object:

```
//Fetch all alternate names for the user using member function in user
construct
AlternateNames names = user.getAlternateNames();
log.info(names);
```

There is a static construct also available for the same operation

```
//Fetch all alternate names for the user using static function
AlternateNames names = AlternateName.getAlternateNames(server);
log.info(names);
```

Now this list can be traversed for getting individual names.

```
//Traverse through the fetched names
List<AlternateName> nameList = names.getAlternateName();
for(AlternateName name : nameList){
      log.info(name);
}
```

Create an alternate name

To create an alternate name for a user following constructs can be used:

Call a function on the user object.

```
//Create an alternate name
      String objectId = "";
      WebCallResult res = user.createAlternateName("Test", “Guy”);
      if(res.isSuccess()){
            objectId = res.getObjectId();
            log.info("Created object id::" + objectId);
      }
      else{
            log.info(res);
      }
```

Use static function to create a new alternate name.

```
//Create an alternate name
      String objectId = "";
```

```
	WebCallResult res = AlternateName.createAlternateName(server, "Test",
“Guy”);
	if(res.isSuccess()){
		objectId = res.getObjectId();
		log.info("Created object id::" + objectId);
	}
	else{
		log.info(res);
	}
```

Fetch details of an individual alternate name using the object id.

```
//Fetch details of an alternate name using its object id
AlternateName d = AlternateName.fetchDetailsForAlternateName(server,
objectId);
d.dumpAllProperties("-->>");
```

Sometimes it may seem unwanted to create an object and the fetch it in different steps. This construct has a function from which an alternate name can be created and fetched in a single step. Internally it will also send two different requests to the server but it definitely saves some code lines.

```
//Create and fetch details of an alternate name simultaneously
AlternateName name = AlternateName.createAndFetchAlternateName(server,
"Test", “Guy”);
	if(name.getWebCallResult().isSuccess()){
		log.info(name);
	}
	else{
		log.info(name.getWebCallResult());
	}
```

This can be done from the user object as well.

```
//Create and fetch details of an alternate name simultaneously
AlternateName name = user.createAndFetchAlternateName("Test", “Guy”);
```

Modify alternate name

There are two ways to modify an alternate name.
One is to directly set the properties of an alternate name and call the update method on it.

```
	//Update the alternate name
		d.setFirstName("Test");
		d.setLastName("Guy");
		WebCallResult r = name.update();
		if(r.isSuccess()){
			log.info("Alternate name successfully updated");
		}
		else{
			log.info(r);
			name.clearPendingChanges();
		}
```

In the second approach, the alternate name can be updated using the static method present in the AlternateName class by passing the properties to be updated in the form of a Map where the key is property name and the value is the property value to be updated.

```
//Update the name
      Map<String, String> props = new HashMap<String, String>();
      props.put("FirstName", "Testing");
            WebCallResult result =
AlternateName.updateAlternateName(server, objectId, props);
      if(result.isSuccess()){
            log.info("name successfully updated");
      }
```

Delete alternate name

To delete an alternate name also, there are two constructs available.
One is to call the delete function on the AlternateName object.

```
//Delete the name
      WebCallResult result = name.delete();
      if(result.isSuccess()){
            log.info("name successfully deleted");
      }
      else{
            log.info(result);
      }
```

Other approach is to use the static method on AlternateName class by passing the objectId of the alternate name.

```
//Delete the name
      WebCallResult result = AlternateName.deleteAlternateName(server,
objectId);
      if(result.isSuccess()){
            log.info("name successfully deleted");
      }
```

AlternateDevice

Every user has a primary extension associated with it in unity connection. There can be other extensions (mobile, home phone etc.) which can be called as alternate devices (alternate extensions). The access to alternate devices depends upon the COS setting for it. The four levels of access are:

- No access
- Read access to administrator-defined alternate devices only
- Read/Write access to user-defined alternate devices only
- Full access (read access to administrator-defined alternate devices and read/write access to user-defined alternate devices)

NOTE: Since the design pattern is same for all the APIs, from here on, only the methods on user object will be mentioned in the document. The static methods can be seen in the java-docs.

Fetch alternate devices for the user:

```
//Fetch all alternate devices for the user using member function in
user construct
AlternateDevices devices = user.getAlternateDevices();
```

Now this list can be traversed for getting individual devices.

```
//Traverse through the fetched devices
List<AlternateDevice> deviceList = devices.getAlternateDevice();
for(AlternateDevice device : deviceList){
      log.info(device);
}
```

Create an alternate device:

```
WebCallResult res = user.createAlternateDevice("12345", null);
String objectId = res.getObjectId();
```

Fetch details of an individual alternate device using the object id.

```
//Fetch details of an alternate device using its object id
AlternateDevice d =
AlternateDevice.fetchDetailsForAlternateDevice(server, objectId);
```

A device can be created and fetched in a single step as well.

```
//Create and fetch details of an alternate device simultaneously
AlternateDevice device = user.createAndFetchAlternateDevice("4358595", null);
```

Modify an alternate device:

```
//Update the device
      device.setDtmfAccessId("7465767");
      device.setDisplayName("abcd");
      WebCallResult r = device.update();
```

Delete an alternate device:

```
//Delete the device
      WebCallResult result =
AlternateDevice.deleteAlternateDevice(server, objectId);
```

HtmlDevice

Every user can have html notification devices using which the HTML notifications (Intelligent notifications as they are called), can be received on an email address.

An END User does not have the privileges to create and delete the HTML device, only view and modify privileges are there. The creation and deletion is allowed to the administrators only.

The Html devices can be fetched as follows:

```
HtmlDevices list = user.getHtmlDevices();
```

This list can be traversed for getting individual devices.

```
//Traverse through the fetched devices
List<HtmlDevice> deviceList = list.getHtmlDevice());
for(HtmlDevice device : deviceList){
      log.info(device);
}
```

Fetch details of an individual html device using the object id.

```
HtmlDevice d = HtmlDevice.fetchDetailsForHtmlDevice(server, objectId);
```

Modify the html device:

```
//Update the html device
d.setSmtpAddress("myaddress@foo.com");
d.setActive(true);
WebCallResult res = d.update();
```

UserExternalServiceAccount

A user may have zero or more unified messaging service accounts (also known as external service accounts). Examples of these services include Cisco Unified MeetingPlace 8.x, Exchange 2003, Exchange 2007, Exchange 2010 or Office 365. A subscriber can do following operations:

- Retrieve a list of its unified messaging service accounts
- Retrieve one of its unified messaging service accounts
- Change the password for one of its unified messaging service accounts

Only an administrator can create or delete an external service account.

The accounts can be fetched as follows:

```
	UserExternalServiceAccounts list =
user.getUserExternalServiceAccounts();
```

This list can be traversed for getting individual accounts.

```
	//Traverse through the fetched devices
	List<UserExternalServiceAccount> accountList =
list.getUserExternalServiceAccount());
	for(UserExternalServiceAccount account : accountList){
		log.info(account);
	}
```

Fetch details of an individual account using the object id.

```
	UserExternalServiceAccount d =
UserExternalServiceAccount.fetchDetailsForUserExternalServiceAccount(server,
objectId);
```

Change the password of an account:

```
		//Update the account
		WebCallResult res = d.update("newpassword");
```

PrivateList

A subscriber can have private lists to create own groups of voice message recipients. When a voice message is addressed to one of the private lists, all of the recipients on the list receive the message.

Fetch private lists for the user:

```
	//Fetch all private lists for the user using member function in user
	construct
	PrivateLists lists = user.getPrivateLists();
```

Now this list can be traversed for getting individual private lists.

```
	//Traverse through the fetched private lists
	List<PrivateList> lists = lists.getPrivateList();
	for(PrivateList list : lists){
		log.info(list);
	}
```

Create a private list:

```
WebCallResult res = user.createPrivateList("MyList", null);

String objectId = res.getObjectId();
```

Fetch details of an individual private list using the object id.

```
//Fetch details of a private list using its object id
PrivateList d = PrivateList.fetchDetailsForPrivateList(server,
objectId);
```

A private list can be created and fetched in a single step as well.

```
//Create and fetch details of a private list simultaneously
PrivateList list = user.createAndFetchPrivateList("MyList", null);
```

Modify a private list:

```
//Update the private list
list.setDisplayName("MyNewList");
WebCallResult r = list.update();
```

Update voice name for the private list:

Using a local file:

```
WebCallResult res = list.updateVoiceName("C:/test/local_file.wav");
```

Using a phone recording:

```
WebCallResult res = list.updateVoiceName(pc);

//Here pc is the PhoneRecording object which can be obtained as
described in PhoneRecording section.
```

Fetch voice name of the private list:

```
WebCallResult res = list.getVoiceName("C:/test/server_file.wav");
```

Delete a private list:

```
//Delete the private list
WebCallResult result = PrivateList.deletePrivateList(server, objectId);
```

PrivateListMember

Any user or system distribution list that is included in the directory can be a member of the private list. The maximum number of members that a subscriber can add to a private list is specified by the administrator.

Following are the various operations related to private list members:

Fetch all members of a private list

The members of a private list can be fetched using member function on a PrivateList object or by static functions in PrivateListMember construct.

Using member function:

```
PrivateListMembers members = list.getPrivateListMembers();
```

Using static function:

```
PrivateListMembers members =
PrivateListMember.getPrivateListMembers(“43c86d14-4065-4a29-ae50-
82d4f5148749”);
```

Now this list can be traversed for getting individual members.

```
	//Traverse through the fetched members
	List<PrivateListMember> nameList = names.getPrivateListMember();
	for(PrivateListMember name : nameList){
		log.info(name);
	}
```

Create a private list member

To create a private list member for a private list, the member function on the private list object can be used. The method accepts a map which contains the properties to be assigned to the private list member.

```
//Create a private list member
	Map<String, String> props = new HashMap<String, String>();
	props.put("MemberSubscriberObjectId", “afa59a71-d06b-4d66-87d2-
75ff0597be6f”);

	WebCallResult res = list.createPrivateListMember(props);
```

Fetch details of an individual private list member using the object id.

```
//Fetch details of a private list member using its object id
PrivateListMember member =
PrivateListMember.fetchDetailsForPrivateListMember(server, objectId);
d.dumpAllProperties("-->>");
```

Those two steps can be done with a single function call.

```
//Create and fetch details of a private list member simultaneously
PrivateListMember member =
PrivateListMember.createAndFetchPrivateListMember(server, props);
```

Delete private list member

To delete a private list member, call the delete function on the PrivateListMember object.

```
//Delete the name
      WebCallResult result = member.delete();
      if(result.isSuccess()){
            log.info("name successfully deleted");
      }
      else{
            log.info(result);
      }
```

SmtpProxyAddress

A subscriber can have email addresses which are not its primary smtp address.

Fetch smtp proxy addresses for the user:

```
//Fetch all addresses using member function in user construct
SmtpProxyAddresses addresses = user.getSmtpProxyAddresses();
```

Now this list can be traversed for getting individual proxy address.

```
//Traverse through the fetched smtp proxy addresses
List<SmtpProxyAddress> list = addresses.getSmtpProxyAddress();
for(SmtpProxyAddress add : list){
      log.info(add);
}
```

Fetch details of an individual smtp proxy address using the object id.

```
//Fetch details of an smtp proxy address using its object id
SmtpProxyAddress address =
SmtpProxyAddress.fetchDetailsForSmtpProxyAddress(server, objectId);
address.dumpAllProperties("-->>");
```

References

1. http://docwiki.cisco.com/wiki/Cisco_Unity_Connection_Provisioning_Interface_%28CUPI%29_API_--_For_End_Users
2. http://www.ciscounitytools.com/CodeSamples/Connection/CUPIForUsers/CUPIForUsers.html