IceLib

Quick Reference Guide

Prerequisites

- .Net Framework 3.5
- ININ Trace Initialization Service Installed and Running – this service is installed with any PureConnect product. The ININ Trace Viewer install is a lightweight install that can be used to setup the service.
- I3_FEATURE_ICELIB_SDK license on the server

Design Principals

Watches

Watches are used to query the values of properties and to receive notifications when those properties change. A StartWatch method must be called before object properties are available and before any events are raised.

All event handlers should be added before calling StartWatching to avoid missing an event that is raised as part of the initial watch.

Watches do not carry over from disconnects. After a session disconnects while there are active watches, StopWatching will need to be called to clean up IceLib's cache. After a new Session is connected, new watches can then be started by calling StartWatching.

StartWatching() StartWatchingAsync()	Starts watching the given class instance
StartWatching(items) StartWatchingAsync(items)	Starts watching the specified items for the given class instance
bool IsWatching()	Determines whether a watch is active for the given class instance
bool IsWatching(item)	Determines whether a watch is active for the specified item for the given class instance
ChangeWatchedItems(items) ChangeWatchedItemsAsync(items)	Changes the items being watched for the given class instance. The exact method name varies for different item types
StopWatching()	Stops watching the given class instance

32bit/64bit

The IceLib .NET assemblies are not built as purely managed (AnyCPU) as they have dependencies on native DLLs. This requires both 32-bit and 64-bit versions so that each can have their respective 32-bit and 64-bit dependencies. When setting up your project in visual studio, your project needs to have a platform target of either x86 or x64.

Asynchronous Operations

IceLib provides synchronous (blocking) and asynchronous (non-blocking) versions of most methods, especially those methods that make a server call.

When an asynchronous method is invoked, the work is scheduled to be performed on a worker thread. Then, the asynchronous method returns back to the calling context so that the initial thread can continue without blocking. These asynchronous method flavors are frequently used when called from a UI thread so that the user experience is not degraded.

Backwards Compatibility

IceLib versions should be behind or the same SU version as the CIC server. So if the IC server is 4.0 SU5, the IceLib version should be 4.0 SU5 or earlier.

Connection

When creating a connection, there are a couple different options for authentication and station types. The example below shows the different authentication and station types, but only one of each can be used to connect.

```
//Generally, nothing needs to be done to this
//object besides just creating it
SessionSettings sessionSettings = new SessionSettings ();

var endpoint = new HostEndpoint("YOUSERVERNAME");
HostSettings hostSettings = new HostSettings(endpoint);

//different authentication methods
WindowsAuthSettings windowsAuthSettings = new WindowsAuthSettings ();
ICAuthSettings icAuthSettings = new ICAuthSettings ("USERNAME", "PASSWORD");

//Supported media is a bitwise OR '|' of the
//types of interactions this user can receive
StationSettings workstationSettings =
    new WorkstationSettings ("WorkstationName", //name of the workstation
                             SupportedMedia.Call |
                             SupportedMedia.Chat); //supported media

StationlessSettings stationlessSettings = new StationlessSettings ();

RemoteStationSettings remoteStationSettings =
    new RemoteStationSettings ("WorkstationName", //name of the workstation
                               SupportedMedia.Call, //supported media
                               "5555555"); //remote number

RemoteNumberSettings remoteNumberSettings =
    new RemoteNumberSettings (SupportedMedia.Call, //supported media
                              "5555555", //remote number
                              false); //is persistent


Session session = new Session ();
session.Connect (sessionSettings, hostSettings,
                 windowsAuthSettings, workstationSettings);
```

Queues/Interactions

Interaction vs Queue Watches

Watches can be done on either a queue or a specific interaction. Queue watches are a more general use case when you want to be aware of all interactions on a queue and their interaction attribute changes. Interaction watches are for more specific use cases where you need to watch a subset of attributes or track the interaction as if flows between different queues.

An Interaction watch cannot be started on the "shared" Interaction class instance that is contained in the InteractionChanged events. A new Interaction instance needs to be created for the same Interaction id as the one contained in the EventArgs for the event. This is to protect different pieces of code both handling an InteractionQueue changed event and trying to start their own separate StartWatching watches with conflicting sets of attributes.

Queue Watches

Queue watches can be performed in bulk using the QueueContentsChanged event or individually using the InteractionAdded, InteractionChanged, InteractionRemoved, ConferenceInteractionAdded, ConferenceInteractionChanged and ConferenceInteractionRemoved events. When registering for the individual events, you can register for all or only the specific ones your application cares about.

When StartWatching is called on a queue, the QueueContentsChanged or InteractionAdded event is raised to notify interactions on the queue at the time StartWatching was called.

Using QueueContentsChanged

The QueueContentsChanged event bundles up multiple interaction and conference interaction adds, removes, and changes. This event is preferred for workgroup and line queues for performance. It can be used for all queues for convenience. (If there is a QueueContentsChanged event handler, then the individual added/removed/changed events will not be fired.)

```
// For this example, we are interested in the "MyInteractions"
// queue of a user with ID "userid".
QueueId queueId = new QueueId(QueueType.MyInteractions, "userid");
_interactionQueue = new InteractionQueue(_interactionsManager, queueId);

// Any attribute that will later be retrieved from the interaction must
// be included in an Interaction or InteractionQueue watch that the
// Interaction instance is part of.
string[] watchedAttributes = new[] { InteractionAttributeName.StateDescription,
                                     InteractionAttributeName.RemoteId,
                                     "EIC_State"};

_interactionQueue.QueueContentsChanged += QueueContentsChanged;
_interactionQueue.LostRights += QueueLostRights;

// Subscribe to events before starting the watch.
_interactionQueue.StartWatching(watchedAttributes);
```

Using Individual Events

Individual interaction and conference interaction adds/removes/changes can be received, as an alternative to the "batching" of QueueContentsChanged. (If there is a QueueContentsChanged event handler, then the individual added/removed/changed events will not be fired.)

```
QueueId queueId = new QueueId(QueueType.MyInteractions, "userid");
_interactionQueue = new InteractionQueue(_interactionsManager, queueId);

// Any attribute that will later be retrieved from the Interaction must
// be included in an Interaction or InteractionQueue watch that the
// Interaction instance is part of.
string[] watchedAttributes = new[] { InteractionAttributeName.StateDescription,
                                     InteractionAttributeName.RemoteId,
                                     "EIC_State"};

_interactionQueue.InteractionAdded += QueueInteractionAdded;
_interactionQueue.InteractionChanged += QueueInteractionChanged;
_interactionQueue.InteractionChanged += QueueInteractionRemoved;
_interactionQueue.ConferenceInteractionAdded += QueueConferenceInteractionAdded;
_interactionQueue.ConferenceInteractionChanged += QueueConferenceInteractionChanged;
_interactionQueue.ConferenceInteractionRemoved += QueueConferenceInteractionRemoved;
_interactionQueue.LostRights += QueueLostRightsEventHandler;

// Subscribe to events before starting the watch.
_interactionQueue.StartWatching(watchedAttributes);
```

Placing a Call

When placing a call, the only required piece of information is the target queue or number which are passed into the constructor of the CallInteractionParameters object.

```
CallInteractionParameters callInteractionParameters =
    new CallInteractionParameters("5555555");

// The call can be placed on behalf of a workgroup.
callInteractionParameters.OnBehalfOfWorkgroup = "marketing";

// An account code can easily be set on the call.
callInteractionParameters.AccountCodeId = "account code ID";

// Or some notes.
callInteractionParameters.Notes = "custom notes";

// Attributes can be initially set on the new call.
callInteractionParameters.AdditionalAttributes["custom attribute name"] =
    "custom attribute value";

Interaction interaction =
    _interactionsManager.MakeCall(callInteractionParameters);
```

Creating a Generic Object

Generic objects can be used as representations of data in external systems. A generic interaction is similar to a call in that it can be queued, answered, held, transferred and disconnected like a normal call, but the key difference is that there isn't a party on the other end of the object.

```
QueueId queueId = new QueueId(QueueType.Workgroup, "custom workgroup queue");

GenericInteractionParameters genericInteractionParameters =
    new GenericInteractionParameters(queueId, InteractionState.Alerting);

// Attributes can be initially set on the new call.
genericInteractionParameters.AdditionalAttributes["custom attribute name"] =
    "custom attribute value";

// The asynchronous version is preferable if invoking the API from the UI thread.
Interaction interaction =
    _interactionsManager.MakeGenericInteraction(genericInteractionParameters);
```

User Status

Getting a List of User Selectable Statuses

The FilteredStatusMessageList class will provide the list of statuses that the user has the ability to select. To get a list of all statuses configured in the system use the StatusMessageList class.

```
var peopleManager = PeopleManager.GetInstance(session);
FilteredStatusMessageList filteredStatuses =
    new FilteredStatusMessageList(peopleManager);

filteredStatuses.StartWatching(new[] { session.UserId });
ReadOnlyCollection<StatusMessageDetails> mySelectableStatuses =
    filteredStatuses.GetList()[session.UserId];

foreach (StatusMessageDetails status in mySelectableStatuses)
{
    //do something with the status
}
filteredStatuses.StopWatching();
```

Getting a User's Current Status

```
UserStatusList statusList = new UserStatusList(peopleManager);
statusList.StartWatching(new[] { session.UserId });
UserStatus myStatus = statusList.GetUserStatus(session.UserId);
```

Updating a User's Status

When updating a user's status, a StatusMessageDetails object needs to be set on the UserStatusUpdate class. The StatusMessageDetails object can be obtained from either the FilteredStatusMessageList or StatusMessageList classes.

```
var peopleManager = PeopleManager.GetInstance(session);
FilteredStatusMessageList filteredStatuses =
    new FilteredStatusMessageList(peopleManager);

filteredStatuses.StartWatching(new[] { session.UserId });
ReadOnlyCollection<StatusMessageDetails> mySelectableStatuses =
    filteredStatuses.GetList()[session.UserId];

UserStatusUpdate updateStatus = new UserStatusUpdate(peopleManager);
// set the required parameters
updateStatus.UserId = session.UserId;

// set the status to update to, we'll just
// select the first one in our list.
updateStatus.StatusMessageDetails = mySelectableStatuses[0] ;

//Set optional parameters
updateStatus.UntilDate = new DateTime(2014, 5, 14);
updateStatus.UntilTime = new DateTime(2014, 5, 14, 8, 0, 0);
updateStatus.Notes = "Call my cell if needed";
//perform the update
updateStatus.UpdateRequest();

filteredStatuses.StopWatching();
```

Configuration

General Concepts

- If no specific properties are requested then only Id/DisplayName will be returned.
- If no rights are specified then the results will have the associated view right applied.
- If no result limit is specified an unlimited number of results will be returned if less than 5 properties are retrieved. If more than 5 properties are requested, the result set will be limited to 300
- Configuration lists have a StartWatching and a StartCaching method on them. StartWatching is used to get configuration objects and to receive events when they are changed. StartCaching is used to only get the configuration objects, not events will be raised when items are cached. It is not necessary to start caching and watching at the same time.
- The changed event will not trigger for “rights” changes
- The added event doesn’t fire for newly added objects because the watch is on the results of the filter and not the filter itself

Creating a new workgroup

To create a new configuration object, get the appropriate configuration list and use the factory method CreateObject() to get an instance of the new object. Then set the properties you want to configure and call Commit() to save the object.

```
var configurationList = new WorkgroupConfigurationList(configurationManager);
var configurationObject = configurationList.CreateObject();

//Set properties on the workgroup
configurationObject.SetConfigurationId("NewWorkgroupName");
configurationObject.Extension.Value = "1234";
configurationObject.HasQueue.Value = true;
configurationObject.QueueType.Value = WorkgroupQueueType.Acd;

//Save the object
configurationObject.Commit();
```

Retrieve Properties for a User

This example gets the first name, last name and extension for all users that the logged in user has admin rights over. To get users that the logged in user has view rights over, SetFilterRightsToView() could have been used instead of SetFilterRightsToAdmin(). If all properties were needed, SetPropertiesToRetrieveToAll() could have been used instead of listing the individual properties.

```
var configurationList = new UserConfigurationList(configurationManager);
var querySettings = configurationList.CreateQuerySettings();

querySettings.SetPropertiesToRetrieve(UserConfiguration.Property.Extension,
                                      UserConfiguration.Property.PersonalInformation_GivenName,
                                      UserConfiguration.Property.PersonalInformation_Surname);
querySettings.SetRightsFilterToAdmin();

configurationList.StartCaching(querySettings);

foreach (var configurationObject in configurationList.GetConfigurationList())
{
    // Get the ID and DisplayName
    var id = configurationObject.ConfigurationId.Id;
    var firstName = configurationObject.PersonalInformation.GivenName;
    var lastName = configurationObject.PersonalInformation.Surname;
    var extension = configurationObject.Extension;

    // Do something else...
}
configurationList.StopCaching();
```

Update the Current User

When updating an object, get the object out of the appropriate configuration list, call PrepareForEdit() object, edit the properties you want to change and then call Commit() to save the changes.

```
var configurationList = new UserConfigurationList(configurationManager);
var querySettings = configurationList.CreateQuerySettings();

querySettings.SetFilterDefinition(UserConfiguration.Property.Id,
                                  session.UserId,
                                  FilterMatchType.Exact);
querySettings.SetRightsFilterToAdmin();
querySettings.SetPropertiesToRetrieve(
    new[] {
        UserConfiguration.Property.PersonalInformation_PhoneNumberOfHome1
    });

configurationList.StartCaching(querySettings);

if (configurationList.GetConfigurationList().Count > 0)
{
    var configurationObject = configurationList.GetConfigurationList()[0];
    configurationObject.PrepareForEdit();
    configurationObject.PersonalInformation.PhoneNumberOfHome1.Value =
        new BasicPhoneNumber("5551234", String.Empty, false);
    configurationObject.Commit();
}

configurationList.StopCaching();
```

Delete a User

Deleting an object requires getting the object out of the appropriate configuration list then calling Delete() on that object.

```
var configurationList = new UserConfigurationList(configurationManager);
var querySettings = configurationList.CreateQuerySettings();

querySettings.SetFilterDefinition(UserConfiguration.Property.Id,
                                  "USER ID",
                                  FilterMatchType.Exact);

configurationList.StartCaching(querySettings);

if (configurationList.GetConfigurationList().Count > 0)
{
    var configurationObject = configurationList.GetConfigurationList()[0];
    configurationObject.Delete();
}

configurationList.StopCaching();
```

Custom Notifications

Custom notifications can be used to send messages to other IceLib application or to fire off handlers. Custom notifications between applications cannot be targeted at a specific user, but you can add user information in the notification data and then act on a received message based on the values of the data.

Send Message to Handler

```
CustomNotification customNotification = new CustomNotification(session);
// Load a string array with the message
String[] purchaseInfo = new string[]
                {
                    "Data Value 1",
                    "Data Value 2",
                };

// Create a custom request, the targeted custom handler should be watching
// for object ID "MyCustomObjectID" and notification event (event ID)
// "MyCustomEventID". Both of these strings would be entered into the
// custom notification initiator of the custom handler. NOTE: both strings
// would be entered without quotes.
CustomRequest creditCardSwipeNotification = new CustomRequest(
    new CustomMessageHeader(CustomMessageType.ServerNotification,
                            "MyCustomObjectID",
                            "MyCustomEventID"));
//Load the request with data
creditCardSwipeNotification.Write(purchaseInfo);
//Send the notification to the custom handler
customNotification.SendServerRequestNoResponse(creditCardSwipeNotification);
```

Receive Message from Handler

```
class MyCustomNotificationWatcher : CustomNotification
{
    public MyCustomNotificationWatcher(Session session)
        : base(session)
    {
        CustomNotificationReceived += HandleCustomNotification;

        //The custom handler would send a custom notification with
        // a Custom Object Identifier of "MyNotificationObjectID" and
        // a Custom Event Identifier of "MyNotificationEventID".
        CustomMessageHeader customHandlerMessageHeader = new CustomMessageHeader(
            CustomMessageType.ServerNotification,
            "MyNotificationObjectID",
            "MyNotificationEventID");

        StartWatching(new[] { customHandlerMessageHeader });
    }

    private void HandleCustomNotification(object sender,
                                          CustomNotificationReceivedEventArgs e)
    {
        var customData = e.Message.ReadStrings();

        if (customData != null)
        {
          //do something with the data
        }
    }
}
```

Directories

A list of directories can be retrieved from the DirectoryConfiguration class which can also be used to watch changes on the list. Using the DirectoryMetadata, you can get the list of get the ContactDirectory object to view a list of entries in the directory.

```
DirectoriesManager directoryManager = DirectoriesManager.GetInstance(session);
DirectoryConfiguration directoryConfig = new DirectoryConfiguration(directoryManager);

directoryConfig.StartWatching();

//Get a list of all Directories
ReadOnlyCollection<DirectoryMetadata> metaData = directoryConfig.GetList();

foreach (DirectoryMetadata dirMetadata in metaData)
{
    ContactDirectory contactDir = new ContactDirectory(directoryManager, dirMetadata);
    contactDir.StartWatching();

    ReadOnlyCollection<ContactEntry> contactEntries = contactDir.GetList();
    //interate over all the entries in the directory
    foreach (ContactEntry entry in contactEntries)
    {
        //do something with the entry
    }
}
```

Session Watches

Session watches are used to watch for login/logout events for users, all users on a station, or all users on a computer

```
SessionWatch sessionWatch = new SessionWatch(_Session);

var watchList = new[] {new SessionWatchId(SessionWatchType.User, "USER ID"),
                       new SessionWatchId(SessionWatchType.Station, "Station ID")};

var watchSettings = sessionWatch.StartWatching(watchList);

foreach (SessionWatchSettings setting in watchSettings)
{
    var userName = setting.UserName;
    var stationSettings = setting.StationSettings;
}
```

Interaction Attribute Monitor

The InteractionAttributeMonitor solves the problem that you need to find interactions across the entire system that have an interaction attribute with a certain value. Without this class, you would have to setup queue watchers on a number of different queues. Based on your environment that could mean watching all line queues, or maybe most workgroup and user queues. This can be problematic because your application needs to keep up with queues being added an removed and you are going to get a lot of change notifications if your attribute changes a lot, but you only care about one value.

The InteractionAttributeMonitor does have some limitations. The primary one is that for each monitor, you can only have one attribute/value pair. An example of this is if I want to know when a call goes disconnected I'll setup an attribute monitor on EIC_State, but because a disconnect is really two different states (internal disconnect and external disconnect) I would need two attribute monitors, although the event handlers for both monitors can be the same. Depending on your needs, this might make things more difficult if you have complex rules on what you care about.

```
        string attributeValue = "H"; //H is the value of EIC_State when the call is held
        _manager = InteractionsManager.GetInstance(session);

        //We want to be able to monitor when the state is held
        AttributeMonitorId monitorId =
            new AttributeMonitorId(InteractionAttributeName.State, attributeValue);

        _monitor = new InteractionAttributeMonitor(_manager, monitorId);

        //notice that there is event handler for InteractionChanged.  In this case,
        //changed will never be called, only added and removed will be

        //Interaction Added will be called when the value of EIC_State = "H"
        _monitor.InteractionAdded += OnMonitorInteractionAdded;
        //Interaction Removed will be called when the value of EIC_State
        //  used to be "H" and now no longer is.
        _monitor.InteractionRemoved += OnMonitorInteractionRemoved;

        //attributes must be passed into the start watching, these do not
        //have to contain the attribute that is in the AttributeMonitorId.
        _monitor.StartWatching(_attributesToWatch);
```

If you call .GetStringAttribute on the interaction from the event args, you may get stale information. The solution is to create a new interaction using the InteractionManager and then grab the attributes off of that interaction.

Proxy Logins

Proxy logins are done to create sessions on behalf of other users. A common use case for this is when you have a server based integration that needs to create different sessions for different users. In order for this to work, the account doing the proxy login needs to have the "Proxy Logins" Access Control right configured on their IC User via the Interaction Administrator application. This setting is in the "Miscellaneous" section of the Access Control settings. The ProxyAutoSettings class is used to achieve this.

```
SessionSettings sessionSettings = new SessionSettings ();

var endpoint = new HostEndpoint("YOUSERVERNAME");
HostSettings hostSettings = new HostSettings(endpoint);

ProxyAuthSettings proxyAuthSettings =
    new ProxyAuthSettings("My_IC_ProxyUserID",
                          "My_IC_ProxyPassword",
                          "My_IC_TargetUserId");

//the workstation settings are the settings for the target user you
//are logging in
StationSettings workstationSettings =
    new WorkstationSettings ("WorkstationName", //name of the workstation
                          SupportedMedia.Call |
                          SupportedMedia.Chat); //supported media


Session session = new Session ();
session.Connect (sessionSettings, hostSettings,
                 proxyAuthSettings, workstationSettings);
```

www.genesys.com