Sametime
Version 9

Sametime 9 Software Development Kit

Browser Client Toolkit Guide

Edition Notice

Note: Before using this information and the product it supports, read the information in "Notices."

This edition applies to version 9 of IBM Sametime (program number 5725-M36) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Introduction

The IBM® Sametime® 9 Browser client was the release of new collaboration technology built on the IBM WebSphere® Application Server platform. This product provides a set of features that extend the traditional Sametime solution to allow it to be used in Web applications, so that it can be exploited by external applications which access the underlying data, using it in ways that best suit the organization's needs. This document, along with the samples and other documentation in the Sametime Browser client Software Development Kit (SDK), provides you with the information you need to build applications that integrate Sametime functionality into your applications.

This document presents information in the following order:

- An overview of Sametime 9 Browser client
- A description of the low-level JavaScript APIs
- A description of the JavaScript widgets
- A discussion of how to use the REST APIs

For more information about the contents of the Sametime 9 Browser client SDK, see the *readme.txt* file in the *stbrowsersdk* folder of the directory in which you installed the SDK.

Terms to know

The following terms are used throughout this guide:

Table of terms used in this guide with a definition of each term.

Term	Definition
API	Application Programming Interface
Dojo	The Dojo Toolkit, is an open-source JavaScript framework which facilitates the creation of rich internet applications. See http://www.dojotoolkit.org.
IDE	Integrated Development Environment, The IBM Rational® Application Developer and Eclipse IDEs are examples of IDEs.
ISV	Independent Software Vendor
RTC	Realtime Collaboration, which describes synchronous technologies such as instant messaging, presence awareness, Web conferences, telephony, and so on.
SIP	Session Initiation Protocol, a standard protocol for managing interactive sessions between users. SIP is used for instant messaging, presence, telephony, and a number of other applications.
TCSPI	The Sametime Telephony Conferencing Service Provider Interface toolkit includes a set of Java classes that enable telephony service providers to integrate audio conference call technology into IBM's real-time collaboration server offerings.

Suggested reading

Applications that leverage the Sametime AJAX Proxy Server are built using DHTML technologies and in order to build such applications, it is necessary to have some understanding of these. Although you do not need to be an expert, the more familiar you are with these technologies, the more success you will have exploiting this product.

The following list suggests readings including tutorials that you can use to better acquaint yourself with different technologies the Sametime AJAX Proxy uses:

Table of suggested documents to read with a URL for each document.

What	Where
Dojo Toolkit	http://dojotoolkit.org/
The Official Dojo Documentation	http://docs.dojocampus.org/
Firebug Wiki	http://getfirebug.com/wiki/index.php/Main_Page
OpenAjax Hub	http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification

Chapter 2. What is the Sametime browser client?

Sametime 9 Browser client is a product in the Sametime family, the leading enterprise Instant Messaging and Web Conferencing product and the platform for IBM's Unified Collaboration (UC^2) strategy. The Sametime 9 Browser client enables advanced users to be more productive and assists forward-looking organizations to leverage their internal knowledge to act in real time to ever-changing market and business situations.

The Sametime browser client gives you market-leading instant messaging capabilities that help you collaborate and keep pace with your real-time work environment.

In reality, the Sametime 9 browser client is a manifestation of the toolkit. While the pre-made client provides Sametime features, the real power of the product is in allowing the creation of communications-enabled business processes. In other words, web applications can be given a new degree of interaction, where the users of a web site can directly interact with each other, or with others who are connected to Sametime.

Architecture

The browser client API is divided into three distinct parts:

1. At the lowest level is a REST API. This is actually quite a complex layer, using a mechanism known as a long poll. An initial REST request is made to the server, and the data associated with this are retrieved by means of a separate asynchronous request which waits for the data from the first request. This second separate request constantly reads from the server, sending a new request each time it times out or receives data. In other words, the browser always has a request active to the server, which is responded to when data become available. Because of its complexity, it is not recommended that the REST API be used directly, but rather one of the other layers instead: using REST is discussed in chapter 6 and the REST API is documented in Appendix A.
2. The next layer up is the API referred to as the *Base Components*. These are a set of JavaScript classes that simplify the interface to Sametime, providing all of the features of the REST API in an much more easily used package. This layer is deliberately kept as lightweight as possible. However, while all of the communications functionality is made available, the Base Components do not provide any user-interface functions. These are discussed in Chapter 3.
3. The top level API is the *User Interface* API. This provides a set of widgets that can be incorporated into a web application in a very simple manner. These widgets are based on the Dojo Toolkit technology, and each encapsulates a piece of Sametime functionality. Programming the UI is discussed in Chapter 4.

If a web application already uses a JavaScript framework, it may be more expedient to use the Base Components on the web page, launching a separate window for specific Sametime functions such as chat. This separate window would then use the Dojo-based UI components without incurring its overhead in the main page.

Chapter 3. Base Components

The base components are really a thin layer encapsulating the REST API, to simplify its use. They effectively hide the complexity of selecting whether to connect via the proxy server or a local Sametime Connect client, and manage the connection in a way that allows the developer make simple API calls rather than having to manage the long poll connection.

Program structure

To include functionality from the Sametime Proxy into your web application requires a number of basic steps:

- Include the JavaScript file to provide the function calls
- Log into Sametime
- Include the calls you require in your application
- Optionally log off the Sametime system

Including the JavaScript file

The base JavaScript functions are held in the base Components JavaScript, which is loaded from the *include.js* file. This is included by means of a <*script*> tag as usual. To ensure that the correct version of the file is always loaded in your application, prefix the script name with the path element *latest*:

```
http://www.yourcompany.com/stwebclient/latest/include.js"
```

Previous versions of this document referred to *baseComps.js*: while this is now deprecated in favor of *script.js*, it is still supported for backwards compatibility.

The global stproxyConfig object is used to set system-wide values. These include:

- *server* – This specifies the location of the Sametime Proxy server, typically its URL.
- *tunnelURI* – The URL where the OpenAjax Hub tunnel code can be retrieved. This is usually located on the application server, not the Sametime Proxy server. Note that the use of the OpenAjax Hub replaces the use of domain-lowering and the use of a separate AJAX Proxy, which were required in releases prior to 8.5.2. See under same-origins later in this chapter for more details.
- *isConnectClient* – If set to *true*, the code will automatically attempt to connect to Sametime via the UIM client, rather than the Sametime Proxy Server
- *tokenLogin* - If you are embedding the web client on your page, set this to *true t*o prevent display of the login page. Note that this does not mean that the login is via SSO, but rather that the UI expects that login will be by SSO so it should not default to displaying the username/password login page.
- *Chat* *
    - *bringWindowToFront* – if this is set, it can be used to force any opened chat window to the front when it receives a message. This can be quite intrusive so it may be a good idea to test this feature with your users before configuring it. If this functionality is required for the pre-

made client, add this setting to the *stproxyConfig* object in the *stproxyweb.war/popup.jsp* file. This may also be restricted due to browser security constraints.

- *partnerLeftMessage* – if this is set to true, displays a message when a chat partner leaves an active 1-1 chat session.
- *isSound* – if this is set to true, a short tone will be played as a notification of a new chat message, a meeting invitation or a received announcement.
- *AllowImages* – (Excluding Emoticons) if this is set to true, all images (including external images) will be processed for rendering. If this is set to false images will be replaced with the URL link.

** Should any of these settings be added via the Sametime System Console, the local values will be replaced.*

Here is code for a typical configuration object used in a Web page:

```
<script type="text/javascript">
  var stproxyConfig = {
      server: "http://proxy.company.com:9080",
      tunnelURI: "http://www.company.com/tunnel.html”,
      tokenlogin: true,        // If it's SSO, don't display login page
      isConnectClient: false,  // Prevent use of the rich client
      chat:   {
                  bringWindowToFront: true, // Force chat windows to front
                  partnerLeftMessage: true // Display when user leaves chat
              }
  }
</script>

<script type="text/javascript"
        src="http://www.ibm.com/stwebclient/latest/include.js"></script>
```

Previous versions of this document referred to *baseComps.js*: while this is now deprecated, it is still supported for backwards compatibility.

Initializing your application

Each call to the Proxy JavaScript code has success and error return functions as the last two functions passed to the call. In this document, we use a common error method for the sake of simplicity:

```
      // Set up the error display
      function generalErrorHandler(reason, error) {
            alert("Error " + error + ": " + reason);
      }
```

When an error occurs, the error callback specified in the API call is executed; otherwise the success callback is executed. Note that all responses from the API are in JSON format and most are asynchronous, i.e. the response is not directly to the API call, but rather delivered to the callback function..

By default your application will connect through the Sametime Connect client if it's available. To force the application to connect to the Proxy server even when the Connect client is active, you should set the *isConnectClient* parameter to false in the *stproxyConfig* object:

```
  var stproxyConfig = {
      server: "http://www.company.com:9080",
      tunnelURI: "http://www.company.com/tunnel.html”,
      isConnectClient : false
  }
```

Alternatively, you can pass the parameter *isConnectClient=false* on the URL to achieve the same effect.

Exiting the application

Note that in Sametime 8.5.1, closing the window or navigating to a new window, disconnected the user from the system. Since Sametime 8.5.2 this has changed so that the system waits for 90s to see if the user reconnects. Each page most reestablish connection to Sametime by calling *login()*: if the user reconnects within 90s, the session is re-established transparently and any queued messages are displayed. However, if the user does not reconnect within this period, any queued messages cannot be delivered and the sender is informed of this.

A general comment on the use of UserIDs

In many API calls, a *userID* parameter is used. It is essential that this is a value that can be resolved to a specific user, i.e. it's a field from the LDAP (or whatever authentication solution is used) that is unique and is known to the Sametime Proxy server as a mechanism to identify a user. Be careful that you don't assume that you can use the display name or E-mail address or similar, since this may not be unique. If in doubt, use the API call *stproxy.resolveUser* which will provide you with the correct ID value.

JavaScript same-origin

The default mechanism to handle JavaScript Same Origin issues, allowing your application to access servers on multiple domains, uses the OpenAjax Hub to pass messages between the various parts of the system. You need to specify the URI of the tunnel file to the *stproxyConfig* object, and include this tunnel file in your application:

```
  var stproxyConfig = {
      server: "http://ajaxproxy.company.com/stproxy",
      tunnelURI: "http://hosting.server.com/tunnel.html”
  }
```

The contents of the tunnel file should be similar to the following:

```
<html>
   <head>
   <title>Hub Tunnel</title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <script type="text/javascript"
              src="http://www.company.com:9080/stbaseapi/OpenAjaxHub/release/all/Op
enAjaxManagedHub-all.js">
      </script>
      <script type="text/javascript">
          function init() {
             OpenAjax.hub.IframeContainer._tunnelLoaded();
          }
       </script>
```

```
    </head>
    <body onload="init();"></body>
</html>
```

Alternatively, some applications require the use of a traditional Proxy server, e.g. the Websphere Portal AJAX proxy, to manage access across multiple domains. If you use this solution. you must allow the Sametime cookies and header values through for it to function correctly. The cookies are:

- LtpaToken
- LtpaToken2
- loginName
- sid
- selectorMap
- headUpdaterWindowName
- UnicaNIODID
- x-Q-App-Context
- flushedId
- JSESSIONID

and the headers are:

- Host
- Accept
- Keep-Alive
- Connection
- X-Requested-With
- Referer
- Cookie
- rtc4web-Nonce

Even if you use such a proxy server, you still must specify the location of your tunnel file for the communications to work correctly.

Debugging your application

To enable debugging for your application, you need to set the Dojo debug flag:

```
var djconfig = { isDebug: true }
```

If you have no browser debugging features installed, the system will automatically load Firebug Lite. You can add debugging statements to trace your code, such as:

```
console.debug("Print this on the console”);
```

Please see the Firebug Wiki for more information.

Login

Logging into Sametime

The code to log into the Sametime server requires that you provide the ID of the user's Sametime account, along with the account's password:

```
<script type ="text/javascript">
      // Set up the success return
      function loggedIn(person) {
            console.log("Logged in as : " + JSON.stringify(person));
      }

      function login() {
            var user =  document.getElementById("username").value;
            var pass = document.getElementById("password").value;
            stproxy.login.loginByPassword(user, pass, stproxy.status.AVAILABLE,
                  'I am online', loggedIn, generalErrorHandler);
      }
</script>
...
<p>
   Username: <input id="username" type="text" name="username" size="20" />
</p>
<p>
   Password: <input id="password" type="password" name="password" size="8"/>
</p>
<p>
   <input type="submit" value="Login" onclick="login();" />
</p>
```

Here, the *username* and *password* are strings retrieved from the web page, while the *stproxy* object manages the connections to the server. Then we call the login object's *login()* method.

The variants of *login()* are:

```
loginByPassword(username, password, initialStatus, initialStatusMessage, callBack,
                errorCallBack, clientId);
loginByToken(username, initialStatus, initialStatusMessage, callBack,
                errorCallBack, clientId);
loginAsAnon(username, initialStatus, initialStatusMessage, callBack,
                errorCallBack, clientId);
loginWithToken(username, token, initialStatus, initialStatusMessage, callBack,
                errorCallBack, clientId);
loginAsAnonWithToken(username, token, initialStatus, initialStatusMessage,
                callBack, errorCallBack, clientId);
```

In practice, it is expected that the vast majority of installations will use SSO and so will use *loginByToken*. *LoginByToken* expects an SSO cookie to be available, while *LoginWithToken* allows the token to be passed as an additional parameter. You can identify if a particular token is available by means of the *hasToken()* method:

```
// Callback once the token is tested
function gotToken(gotIt) {
      if (gotIt)
         loginByToken( … );
      else
         loginByPassword( … );
}
...
stproxy.login.hasToken(gotToken, "LtpaToken2");
```

If login is called when transitioning from one web page to another, it will only perform a true login if the user is not already logged in. Otherwise, it simply reconnects to the user's current server session.

The parameters to the login calls are described below. Note that the *key* (password) parameter is only required when logging in using the *loginByPassword* method.

Table listing parameters for login() method with a description of each parameter.

Parameter	**Type**	**Description**
username	String	The user's login name. For *loginByToken* this is only necessary when connecting to a Communty server that is older than version 8.0.
key	String	The user's password for *loginByPassword*
initialStatus	Numeric	The user's initial status, which can be one of: • stproxy.awareness.OFFLINE • stproxy.awareness.AVAILABLE or • stproxy.awareness.AWAY • stproxy.awareness.DND • stproxy.awareness.NOT_USING • stproxy.awareness.IN_MEETING • stproxy.awareness.AVAILABLE_MOBILE or • stproxy.awareness.AWAY_MOBILE • stproxy.awareness.DND_MOBILE • stproxy.awareness.IN_MEETING_MOBILE • stproxy.awareness.UNKNOWN
initialStatusMessage	String	An initial status message
callback	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered
clientId	Numeric	This indicates to the community server which application has made the call. This should be left blank unless there is a specific reason to set it to a particular value.
token	String	An authentication token

This particular call must be made before any other and it returns a JavaScript object in the response similar to:

```
{
 "status":1,
 "statusMessage":"I'm in the office",
 "username":"user.name",
 "id":"user.id",
 "isAnnon": false
}
```

Not all values are returned each time login is called: it depends very much on how your system has been configured. The above set of values are typical, but it may contain fewer or more items than shown here.

In practice, it is likely that the application will try to log in automatically, as illustrated in the code below. Note that this sample does not attempt to process the success/failure of the login:

```
<script type ="text/javascript">
   stproxy.addOnLoad(function () {
      stproxy.login.loginByToken(null, stproxy.status.AVAILABLE,'I am online');
   });
</script>
```

In exceptional circumstances, when the login is complete, starting some functions immediately may fail because the data transfers associated with the login may not be complete. For example, if a chat is automatically launched immediately after login, it may not work correctly. The solution is to wait a second or so for the data to catch up:

```
function startChat() {
   ...
}

// After logging in, ...
function loggedInOK() {
   setTimeout (startChat, 1000); // Wait 1 second ...
}

// Log in
function loginUser() {
   // Replace with the appropriate user information
   stproxy.login.loginByPassword(userID, password,
                       stproxy.awareness.AVAILABLE, "I'm available",
                       loggedInOK, generalErrorHandler);
}
```

Logging out

The code to log off the Sametime server is quite simple:

```
<script type ="text/javascript">
     // global variables
     var username = null;

     // Set up the success return
     function loggedOut() {
          alert("Goodbye!");
     }
</script>
<p>
     <input type="submit" value="Logout"
        onclick="stproxy.login.logout(true, loggedOut, generalErrorHandler);"/>
</p>
```

Table of parameters for logout() with a description of each parameter.

Parameter	Type	Description
isRealLogout	Boolean	If this is set to *true*, the user is logged out and all open windows are closed. If set to *false* the user is logged out from the server only if the current window is the only Sametime window open.
callBack	Function	The function executed on successful completion, but it is only called if *isRealLogout* is *true*.
errorCallBack	Function	The function called if an error is encountered

When logging out, there area some important issues to be taken into consideration. When the user logs out, the client loses all connection to the server. However, since logging in again in a new page is a not unusual pattern, if the user reconnects to the server within 90s, the connection is reestablished.

If the user wishes to log in again on the same page, it is necessary to tell Sametime about any livenames that might be on the page, as well as reconnect from any other open windows that may require Sametime awareness. To associate the existing livenames with the Sametime functions is quite straightforward:

```
<script type ="text/javascript">
stproxy.addOnLoad(function() {
  dojo.connect(stproxy.login, "onLogin", function() {
      var users=new Array();
      dojo.query(".IMAwarenessDisplayedUser .dn")
           .forEach(function(node){users.push(node.innerHTML);});
      if (users.length > 0) {
         stproxy.watchlist.add(users);
       }
  });
});
</script>
```

Reconnecting each open page is more complicated. It requires that you track each window that has been opened by your application, and when logging in, cycling through the list of open windows calling the login function on each.

Custom Attributes

Custom attributes are updated on the Sametime server by custom plugins. The attributes can be retrieved at the web client by indicating that they should be returned to the client as they change. To indicate to the system that custom attributes should be returned, you can use code similar to this below. This code also illustrates how to unsubscribe from the attribute updates:

```
     // Successful registration of the attributes
     function attrOK(attrs) {
          alert("Registered attributes successfully - " + JSON.stringify(attrs));
     }

     // Indicate that status of attributes should be returned
     function listenForAttibutes() {
        var attr = new Array("12345", "67890");
        stproxy.attributes.add(attr, attrOK, generalErrorHandler);
     }

     // Stop listening for the attributes
     function stopeListening() {
        var attr = new Array("12345", "67890");
        stproxy.attributes.remove(attr, attrOK, generalErrorHandler);
     }
```

Table of parameters for managing custom attributes with a description of each parameter.

Parameter	Type	Description
attributes	String[]	A list of the monitored attributes
CallBack	Function	The function executed on successful completion.
errorCallBack	Function	The function called if an error is encountered

This returns an array to the callback function, listing the attributes that were added.

Processing updates from custom attributes requires that the user captures the update events by overriding the *attributes.onUpdate* method.

```
// Override the dummy method
stproxy.attributes.onUpdate = function(key, value, userid) {
   // We have to use a dynamic array because of the use of variables
   var jsonArr = {};
   jsonArr["id"] = userid;
   jsonArr[key] = value;

   // Force the update into the LiveName model
   stproxy.watchlist.onUpdate(jsonArr);
}
```

Any listeners on the livenameModel can respond accordingly, e.g. by adding extra icons to a livename (see below under *Adding additional icons to a LiveName*).

However, this is an overly simplistic example in that it illustrates an override of the update method. Instead it is more correct to chain methods on an event. In the case of Dojo, this would use the *Connect* method:

```
dojo.Connect(stproxy.attributes, "onUpdate",
             dojo.hitch(this, function( key, value, userid) {
                                 // Your code
                               }
             )
);
```

See later for a list of other event functions, in the section marked *Events.*

Throughout this document, wherever event methods are set to a specific function, these should really be connected in a chain to any methods that might already be listening for the event.

Contacts

There is a set of API calls to allow manipulation of the current user's contact list.

Retrieving the contact list

To retrieve the contact list, use the ***get()*** method of the buddylist object:

```
<script type ="text/javascript">
     // Set up the success return
     function gotBuddies(buddies) {
          alert("Got them!");
     }
     function getBuddies(buddylist) {
        stproxy.buddylist.get(true, false, gotBuddies, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Get Buddylist" onclick="getBuddies();" />
```

The parameters are:

Table of parameters for get() with a description of each parameter.

Parameter	Type	Description
isWatchList	Boolean	This is set to *true* if the contact list is to be added to the WatchList and *false* if not.
isWatchLocation	Boolean	This is set to *true* if you want to listen for updates to the users' locations, and *false* otherwise.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

The *buddylist.get* function returns an object which includes a sequence of groups, each of which contains:

```
"type":"private/public",
"id":"GroupID",
"displayName":"Display Name",
"children":{
    {<user-0 info>},
    {<user-1 info>},
    ...,
    {<user-n info>}
}
```

Note that the child objects can also be other nested groups. The child user information is:

```
{
   "communityId":"<communityID",
   "id":"<user.id>::<communityID>",
   "isExternal":true/false,
   "contactId":"<contact.id>",
   "displayName":"Display Name"
}
```

Adding a group to the contact list

To add a group to the contact list, use the ***addGroup()*** method of the buddylist object:

```
<script type ="text/javascript">
     // Success function
     groupAdded = function() {
          alert("Group added");
     }

     function addGroup() {
          stproxy.buddylist.addGroup("MyNewGroup”, true, groupAdded,
                                              generalErrorHandler);
     }
</script>
...
<input type="submit" value="Add Group" onclick="addGroup()" />
```

Table of parameters for addGroup() with a description of each parameter.

Parameter	Type	Description
groupId	String	The ID of the group to be added to the buddylist.
isPrivate	Boolean	If this is a private group or not. If it is private, the group is created.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

On success this returns the name and privacy of the new group:

```
{
  "groupId”: "groupID”,
  "isPrivate”: true
}
```

Removing a group from the contact list

To remove a group from the contact list, use the ***removeGroup()*** method of the buddylist object:

```
<script type ="text/javascript">
   // Set up the success return
   groupGone = function() {
      alert("Group removed");
   }

   function removeGroup() {
      stproxy.buddylist.removeGroup("MyPrivateGroup”, true, groupGone,
                                    generalErrorHandler);
   }
</script>
...
<p>
   <input type="submit" value="Remove Group" onclick="removeGroup()" />
</p>
```

Table of parameters for removeGroup() with a description of each parameter.

Parameter	Type	Description
groupId	String	The ID of the group to be removed from the buddylist.
isPrivate	Boolean	If this is a private group or not
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

On success this returns the name and privacy of the deleted group:

```
{
  "groupId”: "groupID”,
  "isPrivate”: true
}
```

Renaming a group in the contact list

To rename a group, use the *renameGroup()* method of the buddylist object:

```
<script type ="text/javascript">
     // Set up the success return
     groupRenamed = function() {
          alert("Group renamed");
     }

     function renameGroup() {
          var oldGroupId = "MyGroup”;
          var newGroupId = "MyRenamedGroup”;
          stproxy.buddylist.renameGroup(oldGroupId, newGroupId,
                                        groupRenamed, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Rename Group" onclick="renameGroup()" />
```

Table of parameters for renameGroup() with a description of each parameter.

Parameter	Type	Description
oldGroupId	String	The current group ID.
newGroupId	String	The new name for the group
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

On success this returns the old and new group names:

```
{
  "oldGroupId":"testGroup",
  "newGroupId":"testGroup2"
}
```

Retrieving users

To retrieve the list of users, use the *getUsers()* method of the buddylist object:

```
<script type ="text/javascript">
     // Success function
     gotUsers = function(resp) {
        alert("Retrieved users:" + JSON.stringify(resp));
     }

     stproxy.buddylist.getUsers(true, gotUsers, generalErrorHandler);
</script>
...
     <input type="submit" value="Get Users" onclick="getUsers()" />
```

Table of parameters for getUsers() with a description of each parameter.

Parameter	Type	Description
isPrivate	Boolean	If this is set to *true*, return the list of private users; otherwise return the list of public users.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

On success, this returns an array of users to the callback:

```
[
  { "communityId": "",
    "isExternal": false,
    "id": "userId1::",
    "contactId": "userId1",
    "displayName": "Heather Reeds"
  },
  { "communityId": "",
   ...
  },
  ...
]
```

Retrieving all groups

To retrieve a list of groups, use the *getGroups()* method of the buddylist object:

```
<script type ="text/javascript">
     // Success function
     gotGroups = function(resp) {
          alert("Retrieved groups:" + JSON.stringify(resp));
     }

     stproxy.buddylist.getGroups(true, gotGroups, generalErrorHandler);
</script>
...
     <input type="submit" value="Get Groups" onclick="getGroups()" />
```

Table of parameters for getGroups() with a description of each parameter.

Parameter	Type	Description
isPrivate	Boolean	If this is set to *true*, return the list of private groups; otherwise return the list of public groups.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

The callback method is called with an array of groups, each of which contains the following information:

```
{
   "type":"private/public",
   "id":"GroupID",
   "displayName":"Display Name",
}
```

Retrieving a group

To retrieve a list of users in a group, use the *getGroup()* method of the buddylist object:

```
<script type ="text/javascript">
    // Success function
    gotGroup = function(resp) {
       alert("Retrieved users:" + JSON.stringify(resp));
    }

    function getGroup() {
     stproxy.buddylist.getGroup("MyGroup", true, gotGroup,
                                  generalErrorHandler);
    }
</script>
...
   <input type="submit" value="Get Group" onclick="getGroup()" />
```

Table of parameters for getGroup() with a description of each parameter.

Parameter	Type	Description
groupId	String	The ID of the group to be retrieved
isPrivate	Boolean	If this is set to *true*, return the list of private groups; otherwise return the list of public groups.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

This returns an object which includes a sequence of groups, each of which contains:

```
"type":"private/public",
"id":"GroupID",
"displayName":"Display Name",
"children":{
    {<user-0 info>},
    {<user-1 info>},
    ...,
    {<user-n info>}
}
```

The child user information is:

```
{
   "communityId":"<communityID",
   "id":"<user.id>::<communityID>",
   "isExternal":true/false,
   "contactId":"<contact.id>",
   "displayName":"Display Name"
}
```

Adding a user to a group

To add a user to a group, use the *addUser*() method of the *buddylist* object:

```
<script type ="text/javascript">

     // Success function
     userAdded = function() {
        alert("User added");
     }

     function addUser() {
        var userId = "New User”;
        var groupId = "MyGroup”;
        stproxy.buddylist.addUser("New User”, "MyGroup”,
                                  userAdded, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Add User" onclick="addUser()" />
```

Table of parameters for addUser() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the user to be added to the buddylist.
groupId	String	The ID of the group to which the user is to be added.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

Removing a user from a group

To remove a user from a group, use the *removeUser()* method of the *buddylist* object:

```
<script type ="text/javascript">
     // Success function
     userGone = function() {
          alert("User deleted");
     }

     function removeUser() {
          stproxy.buddylist.removeUser("User Id”, "MyGroup”, userGone,
                                                        generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Remove User" onclick="removeUser()" />
```

Table of parameters for removeUser() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the user to be removed from the buddylist.
groupId	String	The ID of the group to which the user belongs.
callBack	Function	The function executed on successful completion
errorCallBack	Function	The function called if an error is encountered

This returns the following on success:

```
{
  "userId": "myUser”,
  "groupId": "myGroup”
}
```

Renaming a user

To rename a user, use the *renameUser()* method of the *buddylist* object:

```
<script type ="text/javascript">
     // Success function
     userRenamed  = function() {
          alert("User renamed");
     }

     function renameUser() {
          stproxy.buddylist.renameUser("Old Id”, "New Display Name”,
                                        userRenamed, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Rename User" onclick="renameUser()" />
```

Table of parameters for renameUser() with a description of each parameter.

Parameter	Type	Description
userId	String	The current user ID.
newUserDisplayName	String	The new name to be used
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

On success, this returns the following to the callback:

```
{
  "newUserDisplayName":"myNewName",
  "userId":"user.id"
}
```

Sending an announcement

To send an announcement to a list of users, use the ***sendAnnouncement()*** method of the *buddylist* object:

```
<script type ="text/javascript">
     // Success function
     announcementSent = function() {
          alert("Announcement sent");
     }

     function sendAnnouncement() {
          var receivers = document.getElementById("receivers").value;
          var message = document.getElementById("message").value;
          var isAllowed = document.getElementById("isAllowed").value;
          stproxy.buddylist.sendAnnouncement(receivers, message, isAllowed,
                                announcementSent, generalErrorHandler);
     }
</script>
...
<p>
     Users list: <input id="receivers" type="text" size="20" />
     Message: <input id="message" type="text" size="20" />
     Allowed: <input id="isAllowed" type="checkbox" />
     <input type="submit" value="Announcement" onclick="sendAnnouncement()" />
</p>
```

Table of parameters for sendAnnouncement() with a description of each parameter.

Parameter	Type	Description
Receivers	String[]	A list of users to whom the announcement is sent
Message	String	The message to be displayed
isAllowed	Boolean	If the user can respond.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

User status

Set the user's current status

To set the current user's status, use the *set()* method on the status object:

```
<script type ="text/javascript">
     // Success function
     statusSet = function() {
          alert("Status set");
     }

     function setStatus() {
        var statusValues = [
          stproxy.awareness.OFFLINE,
          stproxy.awareness.AVAILABLE,
          stproxy.awareness.AWAY,
          stproxy.awareness.DND,
          stproxy.awareness.IN_MEETING
        ];

      var myStatus = document.getElementById("mystatus").selectedIndex;
      var message = document.getElementById("statusMessage").value;
      stproxy.status.set(statusValues[myStatus], message,
                         statusSet, generalErrorHandler);
   }
</script>
...
<p>
     Select status:
     <select id="mystatus" name="mystatus">
          <option>Offline</option>
          <option selected="selected">Available</option>
          <option>Away</option>
          <option>Do not disturb</option>
          <option>In a meeting</option>
     </select>
     Message: <input id="message" type="text" size="20" /><br />
     <input type="submit" value="Change My Status" onclick="setStatus()" />
</p>
```

Table of parameters for status.set() with a description of each parameter.

Parameter	Type	Description
status	Numeric	The user's new status. See the *initialStatus* parameter to *login* above for allowed values.
statusMessage	String	The string to be displayed
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

On success, this simply calls the callback with no parameters.

WatchList

The WatchList is the mechanism by which Sametime maintains the presence awareness associated with the names in the current application. It consists of the list of users in your contact list, as well as any added programmatically (any name on the current web page).

Adding users to the WatchList

To add users to the WatchList, use the *add()* method of the WatchList object:

```
// Success function
usersAdded = function() {
     alert("User added");
}

function addWLUser(users) {
     stproxy.watchlist.add(users, usersAdded, generalErrorHandler);
}
```

Table of parameters for add() with a description of each parameter.

Parameter	Type	Description
Users	String[]	An array containing the names of the users to be added, or a single String containing the name of a single user.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Removing users from the WatchList

To remove a user from the WatchList, use the *remove()* method of the WatchList object:

```
// Success function
userGone = function() {
     alert("Users deleted");
}

function removeWLUser(users) {
     stproxy.watchlist.remove(users, userGone, generalErrorHandler);
}
```

Table of parameters for remove() with a description of each parameter.

Parameter	Type	Description
Users	String[]	An array containing the names of users to be removed, or a String containing the name of the single user to be removed.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

On success, this returns the ID of the user who was removed as:

```
{ "id": "user.id" }
```

Adding a group to the WatchList

To add a group to the WatchList, use the *addGroup()* method of the WatchList object:

```
<script type ="text/javascript">
     // Success function
     usersAdded = function(groupId) {
          alert("Group " + groupId + " added");
     }

     function addWLGroup(group) {
          stproxy.watchlist.addGroup(group, usersAdded, generalErrorHandler);
     }
```

Table of parameters for add() with a description of each parameter.

Parameter	Type	Description
GroupId	String	The ID of the group to add.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Removing a group from the WatchList

To remove a group from the WatchList, use the *removeGroup()* method of the WatchList object:

```
<script type ="text/javascript">
     // Success function
     usersAdded = function(groupId) {
          alert("Group " + groupId + " removed");
     }

     function addWLGroup() {
          var group = document.getElementById("group").value;
          stproxy.watchlist.removeGroup(group, usersAdded, generalErrorHandler);
     }
</script>
...
<p>
     User names: <input id="group" type="text" size="20" />
     <input type="button" value="Remove Group From WatchList"
                          onclick="addWLGroup()" />
</p>
```

Table of parameters for add() with a description of each parameter.

Parameter	Type	Description
GroupId	String	The ID of the group to remove.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Remove all users from the WatchList

To remove all users from the WatchList, use the *clear()* method of the WatchList object:

```
<script type ="text/javascript">
     // Success function
    cleared = function() {
          alert("WatchList cleared: All users deleted");
     }

     function clearWL() {
          stproxy.watchlist.clear(cleared, generalErrorHandler);
     }
</script>
...
<p>
     <input type="submit" value="Clear WatchList" onclick="clearWL()" />
</p>
```

Table of parameters for clear() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Suspend WatchList updates

To temporarily prevent WatchList updates from being sent to the application, use the *suspend()* method of the WatchList object:

```
<script type ="text/javascript">
     // Success function
     suspended = function() {
          alert("WatchList updates suspended");
     }

     function suspendWL() {
          stproxy.watchlist.suspend(suspended, generalErrorHandler);
     }
</script>
...
<p>
     <input type="submit" value="Suspend WatchList" onclick="suspendWL()" />
</p>
```

Table of parameters for suspend() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Resume WatchList updates

To resume prevent WatchList updates from a suspended state, use the ***resume()*** method of the WatchList object:

```
<script type ="text/javascript">
          // Success function
     resumed = function() {
          alert("WatchList updates resumed");
     }

     function resumeWL() {
          stproxy.watchlist.resume(resumed, generalErrorHandler);
     }
</script>
...
<p>
     <input type="submit" value="Resume WatchList" onclick="resumeWL()" />
</p>
```

Table of parameters for resume() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Get WatchList user status

To retrieve the status of a user on the WatchList, use the *getStatus()* method of the WatchList object:

```
<script type ="text/javascript">
      // Success function
     gotStatus  = function(resp) {
          alert("Status:" + JSON.stringify(resp));
     }

     function getStatus() {
          var userId = document.getElementById("userid").value;
          stproxy.watchlist.getStatus(userId, gotStatus, generalErrorHandler);
     }
</script>
...
<p>
     User name: <input id="userid" type="text" size="20" />
     <input type="submit" value="Resume WatchList" onclick="getStatus()" />
</p>
```

Table of parameters for getStatus() with a description of each parameter.

Parameter	Type	Description
userId	String	The current user ID.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

This returns the user's status in the format:

```
{
   "status”: "<presence status>”,
   "statusMessage”:”<status text>”
}
```

Chat

The *chat* object is used to control an instant messaging session between users.

Starting a 1-to-1 chat

To start a chat, use the *openChat()* method:

```
<script type ="text/javascript">
      function startChat() {
          var userId = document.getElementById("userid").value;
          stproxy.openChat(userId);
     }
</script>
...
<p>
     User name: <input id="userid" type="text" size="20" />
     <input type="submit" value="Start Chat" onclick="startChat()" />
</p>
```

Table of parameters for openChat() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the chat partner.

Starting a multi-way chat

To start a many-to-many chat, use the *openGroupChat()* method:

```
     function startChat() {
          var userIds = document.getElementById("userids").value.split(";”);
          var topic = document.getElementById("topic").value;
          stproxy.openGroupChat(userIds, topic);
     }
```

Table of parameters for openGroupChat() with a description of each parameter.

Parameter	Type	Description
userIds	String[]	The array of IDs of the chat partners.
topic	String	The chat topic

Chat models

When a chat is initiated, or a similar chat is resumed, a chat model object is created to allow the chat interactions to be controlled more easily. The chat model can be retrieved as follows for the 1-to-1 chat:

```
var myChatModel = stproxy.getChatModel(userId, {"isIncoming”:false})
```

Table of parameters for getChatModel with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the chat partner
isIncoming	Boolean	TRUE if this is an invitation, and FALSE if this is started locally

For an multi-way chat, it is very similar:

```
var myGroupChatModel = stproxy.getGroupChatModel(placeId)
```

Table of parameters for getGroupChatModel with a description of each parameter.

Parameter	Type	Description
placeId	String	The placeId of the Group Chat

When the chat ends, you must call *chatModel.close()*.

Privacy list

The *privacy* object is used to control visibility to other users.

Retrieving the privacy list

To retrieve the privacy list, use the *get* method of the *privacy* object:

```
<script type ="text/javascript">
      // Success function
     gotList  = function(resp) {
          alert("Privacy list:" + JSON.stringify(resp));
     }

     function getPL() {
      stproxy.privacy.get(gotList, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Get Privacy List" onclick="getPL()" />
```

Table of parameters for privacy.get() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

This returns an object that contains a list of the users in the privacy list, as well as indicating if it is exclusive or inclusive:

```
{
   "isExcluding" : <boolean>,
   "list": ["userId1", "userId2", … "userIdn"]
}
```

Updating the privacy list

To change the privacy list, use the *add* method of the *privacy* object:

```
<script type ="text/javascript">
      // Success function
     privAdded  = function() {
          alert("Updated the privacy list");
     }

     function addUsers() {
          var userList = document.getElementById("userlist").value.split(",");
          var excluding = document.getElementById("excluding").value;
          stproxy.privacy.add(userList, excluding, privAdded,
                                                   generalErrorHandler);
     }
</script>
...
<p>
     User name: <input id="userlist" type="text" size="20" />
     Is excluding: <input id="excluding" type="checkbox" />
     <input type="submit" value="Add to Privacy list" onclick="addUsers()" />
</p>
```

Table of parameters for privacy.add() with a description of each parameter.

Parameter	Type	Description
users	String[]	An array containing the complete updated privacy list
isExcluding	Boolean	When TRUE, it indicates that the list of names in the list is to be blocked; when set to FALSE, it indicates that all users *except* those in the list are to be blocked.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

This returns an object that is identical to that returned by *privacy,get*.

Quick Find

The *quickfind* object is used to search for users

Retrieving a list of users

To retrieve a list of users and groups that match a given string, use the generalized *get* method of the *quickfind* object:

```
<script type ="text/javascript">
      // Success function
     found  = function(resp) {
          alert("Found users:" + JSON.stringify(resp));
     }

     function getQF() {
          stproxy.quickfind.get("heather", found, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="Quickfind" onclick="getQF()" />
```

Table of parameters for quickfind.get() with a description of each parameter.

Parameter	Type	Description
searchString	String	The string to match against user names
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

This returns a list of the users whose names match the searchString parameter:

```
persons: ["userId1", "userId2", … "userIdn"]
```

There are two specialized forms of this API that are used to retrieve only users and only groups, rather than this generic version. The parameters are identical, the only real difference being that if you wish to located only one of users or groups, using the specialized version is much more efficient.

The APIs are:

```
stproxy.quickfind.getUsers("heather", found, generalErrorHandler);
```

and

```
stproxy.quickfind.getGroups("heather", found, generalErrorHandler);
```

Person

The *person* object is used to retrieve information on a user.

Retrieving user information

To retrieve a list of items that provide information on a user, use the *getUserInfo* method of the *person* object:

```
<script type ="text/javascript">
      // Success function
     gotInfo  = function(resp) {
          alert("Got user information:" + JSON.stringify(resp));
     }

     function getUI() {
          stproxy.person.getUserInfo("A.N.Other", gotInfo, generalErrorHandler);
     }
</script>
...
     <input type="submit" value="User Information" onclick="getUI()" />
```

Table of parameters for getUserInfo() method with a description of each parameter.

Parameter	Type	Description
user	String	The user for whom the information is retrieved.
callBack	Function	The function executed on success.
errorCallBack	Function	The function called if an error is encountered.

This returns the user's information, for example:

```
info:[{ "Telephone":"555-1234",
        "Location":"Whatever City",
        "Title":"System Manager",
        "Name":"Heather Reeds",
        "PhotoURL":"http%3A%2F%2Fproxy.server.com%2FHeather_Reeds.jpg",
        "Company":"Acme Corporation"
      }]
```

The content of the returned message depends on the content of the user's entry in the directory service – there may be fewer fields or there may be more.

Meetings

The m*eeting* object is used to manage on-line meetings.

Inviting to an existing meeting-room

To invite users to an existing meeting-room, use the following:

```
<script type ="text/javascript">
      // Success function
     started  = function() {
          alert("Started the meeting");
     }

     function startMeeting() {
          var userList = document.getElementById("userlist”).value.split(",”);
          var topic = document.getElementById("topic”).value;
          var url = document.getElementById("url”).value;
          stproxy.meeting.inviteToMeetingRoom(topic, userList, url,
                                              started, generalErrorHandler);
     }
</script>
...
<p>
     Invitees: <input id="userlist" type="text" size="120" />
     Topic: <input id="topic" type="text" size="120" />
     Meeting URL: <input id="url" type="text" size="120" />
     <input type="submit" value="Start Meeting" onclick="startMeeting()" />
</p>
```

Table of parameters for inviteToMeetingRoom() with a description of each parameter.

Parameter	Type	Description
Topic	String	The topic of the meeting room
inviteList	String[]	The array containing the list of users to invite to the meeting.
url	String	The address of the meeting room. If omitted, it will attempt to find the meeting by name from the Topic
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Inviting to a new meeting room

To invite users to a meeting and create a new room, use the following:

```
<script type ="text/javascript">
      // Success function
     started  = function() {
          alert("Started the meeting");
     }

     function startNewMeeting() {
          var userList = document.getElementById("userlist”).value;
          var topic = document.getElementById("topic”).value;
          var url = document.getElementById("url”).value;
          stproxy.meeting.createInstantMeeting(topic, userList,
                                               callBack, errorCallBack);
     }
</script>
...
<p>
     Invitees: <input id="userlist" type="text" size="120" />
     Topic: <input id="topic" type="text" size="120" />
     <input type="submit" value="Start Meeting" onclick="startMeeting()" />
</p>
```

Table of parameters for createInstantMeeting() with a description of each parameter.

Parameter	Type	Description
topic	String	The meeting title
inviteList	String[]	The array containing the list of users to invite to the meeting.
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Retrieve the list of meeting-rooms

To find the list of available meeting-rooms, use the following:

```
     // Success function
     gotRooms  = function() {
          alert("Got the rooms");
     }

     function getRooms() {
          stproxy.meeting.getRooms(gotRooms, errorCallBack);
     }
```

Table of parameters for getRooms() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Telephony

The integration of telephony functions is necessarily simplified, in that it is possible to call a user using their ID or using a specified number. Calls can be placed either by using a user ID, in which case the user's default phone number is used, or by explicitly specifying a number. In both cases, the caller's phone will ring first and, when that end of the call is initiated, the partner's phone is called and the connection is established.

```
function placeACall(myNum, userNum, userId) {
   if (userNum && userNum != "")
      stproxy.call.byNumber(userNum, myNum, generalSuccess, generalError)
     else
      stproxy.call.byId(userId, myNum, generalSuccess, generalError)
}
```

Table of parameters for call with a description of each parameter.

Parameter	Type	Description
UserNumber	String	The number to call in *call.byNumber()*
userId	String	The ID of the user to call in *call.byId()*
myNumber	String	The caller's number
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Preferences

From Sametime 9.0, a set of persistent settings are supported. These are:

Table of supported prefernces

Name	Type	
BRING_CHAT_WINDOW_TO_FRONT	Boolean	Causes the chat window to be displayed over other windows when a new chat message arrives
NOTIFICATION_PLAY_A_SOUND	Boolean	Causes a sound to be played when a new chat message arrives
DISPLAY_PHOTO_IN_TABBED_CHAT	Boolean	Displays the partner's photo in the tab of a tabbed chat
CONTACT_LIST_EXPAND	Array	Automatically expand the BuddyList on load
NOTIFY_PARTNERS_LEAVE_CHAT	Boolean	Tell the user when a chat partner has left the chat
SAVE_CONTACT_LIST_ON_EXIT	Boolean	Save the state of the contact list when the user exits
DISPLAY_OFFLINE_USERS	Boolean	Display offline users in the BuddyList. Setting this to FALSE causes only online users to be displayed.

Get all preferences

To retrieve the current value of a preference, use the *get()* method of the *preferences* object:

```
 // Success function
gotPrefs  = function(resp) {
     alert("Preferences:" + JSON.stringify(resp));
}

function getStatus() {
     stproxy.preferences.get(gotPrefs, generalErrorHandler);
}
```

Table of parameters for get() with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Get a single preference

To retrieve a single preference use the *getPreference()* method of the *preferences* object:

```
// Success function
gotPrefs  = function(resp) {
     alert("Preference:" + JSON.stringify(resp));
}

function getStatus() {
     stproxy.preferences.getPreference("SAVE_CONTACT_LIST_ON_EXIT”,
                                        gotPrefs, generalErrorHandler);
}
```

Table of parameters for getPreference() with a description of each parameter.

Parameter	Type	Description
Preference	String	The preference name
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Set preference setting

To retrieve the current value of a preference, use the set() methods of the preference object:

```
// Success function
setPrefs  = function() {
     // No response data for this call
     alert("Preferences set");
}

function getStatus() {
     stproxy.preferences.set({"SAVE_CONTACT_LIST_ON_EXIT”: true},
                setPrefs, generalErrorHandler);
}
```

Table of parameters for preferences.set() with a description of each parameter.

Parameter	Type	Description
preference	String	The preference name
callBack	Function	The function executed on success
errorCallBack	Function	The function called if an error is encountered

Miscellaneous

There are a number of API calls that don't fall into any particular category.

Get the version number

To retrieve the version number of the Sametime Proxy server, use *getBuildNumber*:

```
// Success function
function gotion(resp) {
     alert("The current build number is :" + JSON.stringify(resp));
}

stproxy.getBuildNumber(gotVersion, generalErrorHandler);
```

Table of parameters for getBuildNumber () method with a description of each parameter.

Parameter	Type	Description
callBack	Function	The function executed on success.
errorCallBack	Function	The function called if an error is encountered.

This returns the build information as a JSON object with two fields, the application and the installation version numbers.

Play a sound

To force the system to play the sound usually associated with a message, use *sound.play()*:

```
stproxy.sound.play();
```

This takes no parameters and does not return anything.

Get the icon for a user status

When a user's status has changed, you can update the display with the usual status icons. You can retrieve the icon associated with a particular status bt means of *getIconURL*

```
var imgsrc = stproxy.getIconURL(stproxy.awareness.AWAY);
document.getElementById("statusImg").src = imgsrc;
...
<img id=”statusImg” />
```

Table of parameters for getBuildNumber () method with a description of each parameter.

Parameter	Type	Description
status	Numeric	The user's status

Resolve a user

It can be useful to resolve a user's name to her userId which is required by many API calls.

```
function resolved(user) {
   console.log("Success: " + JSON.stringify(user));
}

function action() {
    stproxy.resolveUser("Heather Reeds", generalSuccess, generalError);
}
```

Table of parameters for getBuildNumber () method with a description of each parameter.

Parameter	Type	Description
userName	String	The user's name, or some other identifier that you want to resolve to an ID
callBack	Function	The function executed on success.
errorCallBack	Function	The function called if an error is encountered.

This returns a JSON object with two fields, *resolvedName* containing the user's name and *id* containing the user's ID.

Server Responses

Responses to these API calls are all delivered asynchronously via the long-poll channel, in a message that contains the response as JSON. Note that the responses from multiple calls can be returned in a single response.

While these calls result in the callback methods being called, updates that arrive as a result of other situations trigger events that cause the execution of the methods below.

Events

The application can subscribe to a number of events in the Sametime code, using common JavaScript techniques, for example using *dojo.hitch* or *dojo.connect*. As has been emphasized before, it is very important that these are not simply overridden, since that will remove the current functionality which is required for the correct functioning of the system. As an example, the following code adds the ability to set the user's status text on the page in response to a change to the livename model:

```
var livename = new sametime.LiveName({"userId": userId});
domnode.innerHTML = "";
domnode.appendChild(livename.domNode);
dojo.connect(livename.model, "onUpdate", livename, function() {
   var statusTxt = dojo.byId(this.userId + "stStatusElem");
   statusTxt.innerHTML = this.model.statusMessage;
});
```

Generic events

```
stproxy.onUnauthorizedResponse(code, error)
```

This is triggered when the server responds with the HTTP code 401 or 403, typically as the result of a challenge by a reverse proxy or some other security mechanism.

Table of parameters for onUnauthorizedResponse() with a description of each parameter.

Parameter	Type	Description
code	Numeric	The error code
error	String	A description of the error

```
stproxy.onServerAdminMessage (message)
```

When the Sametime Administrator send a global message to all clients connect to the Sametime Community server

Table of parameters for onServerAdminMessage() with a description of each parameter.

Parameter	Type	Description
message	String	The text of the message

Login

```
stproxy.login.onLogin(pers, community)
```

This is executed after a successful login. However, it is more usual to use the callbacks associated with the login method.

Table of parameters for onLogin() with a description of each parameter.

Parameter	Type	Description
pers	Object	The returned person object
communitiyId	String	The returned Community ID

The person object can contain the following fields:

Table of fields for person object with a description of each field.

Field	Type	Description
id	String	The user's ID
isAnnon	Boolean	Whether this is an anonymous user or not
status	Numeric	The user status value
statusMessage	String	The users status message
UserName	String	The user's name

```
stproxy.login.onLogout()
```

Called after successful logout.

```
stproxy.login.onError(code, error)
```

This is triggered when an error occurs during log in, although, like onLogin, it is more usual to use the error callback in the login method.

Table of parameters for onError() with a description of each parameter.

Parameter	Type	Description
code	Numeric	The error code
error	String	A description of the error

Error conditions

```
stproxy.error.onForceLogout(title,message)
```

This is called when the user is forced to log out, typically because of logging into a different client or location.

Table of parameters for onForceLogout() with a description of each parameter.

Parameter	Type	Description
title	String	The title of the event
message	String	A description of the issue

```
stproxy.error.onNodeDown()
```

This is called when a node in a cluster fails over successfully.

```
stproxy.error.onNodeUp()
```

This is called when a node in a cluster is restarted after a fail over.

```
stproxy.error.onCommunityServerDown()
```

This is called when the communoity server is unreachable.

```
stproxy.error.onServerDown()
```

This is called when the WAS server is unreachable.

```
stproxy.error.onSessionExpired()
```

This is called when the server session expires.

```
stproxy.error.onApplicationDown()
```

This is called when the SametimeProxy application on the WAS server has shut down.

```
stproxy.error.onError(code, error)
```

This is a generic failure that is not processed elsewhere.

Table of parameters for onError() with a description of each parameter.

Parameter	Type	Description
code	Numeric	The error code
error	String	A description of the error

Attributes

```
stproxy.attributes.onUpdate(attributeId, value, userId)
```

When the attributes associated with a user are updated, this is called.

Table of parameters for onUpdate() with a description of each parameter.

Parameter	Type	Description
attributeId	String	The attribute ID
value	String	The new value
userId	String	The ID of the user

BuddyList

```
stproxy.buddylist.onLocation(person)
```

similar to onUpdate but is called when a user changes their location rather than status.

Table of parameters for onLocation() with a description of each parameter.

Parameter	Type	Description
person	Object	The updated person object

```
stproxy.buddylist.onAnnouncement(userId, Message, isResponseAllowed)
```

This is called when an announcement is received.

Table of parameters for onAnnouncement() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who sent the announcement message
Message	String	The value of the message
isRspAllowed	Boolean	Whether the user is allowed to respond or not.

Watchlist

```
stproxy.watchlist.onUpdate(update)
```

This is fired when information on one or more users in the watchlist changes.

Table of parameters for onUpdate() with a description of each parameter.

Parameter	Type	Description
update	Object	A watchlist update

The watchlist update object can contain the following fields:

Table of fields for Update object with a description of each field.

Field	Type	Description
id	String	The user's ID
status	Numeric	The user's status
statusMessage	String	The users status message

```
stproxy.watchlist.onLocation(person)
```

similar to onUpdate but is called when a user changes their location rather than status.

Table of parameters for onLocation() with a description of each parameter.

Parameter	Type	Description
person	Object	The updated person object

Chat

```
stproxy.chat.onChatOpen(userId)
```

This is triggered when the chat is initialized by the chat partner, but before the chat window is opened.

Table of fields for onOpen with a description of each field.

Parameter	Type	Description
UserId	String	The user's ID

```
stproxy.chat.onNewChatReceived(userId, displayName)
```

This is triggered when a new chat request arrives.

Table of parameters for onNewChatReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
displayName	String	The display name of the user

```
stproxy.chat.onTypingMessage(userId, isTyping)
```
This is called when the user starts or stops typing.

Table of parameters for onTypingMessage() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
isTyping	Boolean	Whether the user us typing or not

```
stproxy.chat.onNewMessage(userId, msg)
```
This is called when any chat message is received, i.e. for all users.

Table of parameters for onNewMessage() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
msg	String	The chat message

```
stproxy.chat.onMessageReceived(userId, msg, imageIds)
```
This is called when a chat message is received. For previous releases, this was triggered in the chat window. However, since chats now are tabbed in a single window, it is recommended to listen to the chat model's events instead.

Table of parameters for onMessageReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
msg	String	The chat message
imageIds	String Array	A list of the Ids of the images used in the message, if any. The images are retrieved using the image API.

```
stproxy.chat.onOfflineMessageReceived(userId, msg, displayName)
```
This is called when an offline chat message is received.

Table of parameters for onOfflineMessageReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who sent the message
msg	String	The chat message
placeId	String	The user's display name

```
stproxy.chat.onChatData(isRichText,userId)
```
This indicates the capabilities of the chat partner.

Table of parameters for onChatData() with a description of each parameter.

Parameter	Type	Description
isRichText	Boolean	Whether the user can use rich text or not
UserId	String	The user's ID

```
stproxy.chat.onClose(userId)
```
This is executed when the chat is closed.

Table of parameters for onClose() with a description of each parameter.

Parameter	Type	Description
UserId	String	The user's ID

Meeting

```
stproxy.meeting.onInvitation(invitationDetails)
```
This is called when a meeting invitation is received.

Table of parameters for onInvitation() with a description of each parameter.

Parameter	Type	Description
invitationDetails	Object	A n object describing the meeting

N-way Chat

```
stproxy.nwaychat.onNewMessage(placeId, userId, message)
```
This is called when any new n-way chat message is received.

Table of parameters for onMessageReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
msg	String	The chat message
placeId	String	Reference to the meeting

```
stproxy.nwaychat.onMessageReceived(placeId, userId, message, imageIds)
```

This is called when a chat message is received in an n-way chat.

Table of parameters for onMessageReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
msg	String	The chat message
placeId	String	Reference to the meeting
imageIds	String Array	A list of the Ids of the images used in the message, if any. The images are retrieved using the image API.

```
stproxy.nwaychat.onTyping(placeId, userId, isTyping)
```

This is called when the user starts or stops typing in an n-way chat.

Table of parameters for onTyping() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
isTyping	Boolean	Whether the user us typing or not
placeId	String	Reference to the meeting

```
stproxy.nwaychat.onInvitationReceived(placeId, topic, userId)
```

This is called when the user receives an invitation to an n-way chat.

Table of parameters for onInvitationReceived() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
topic	String	The topic of the chat
placeId	String	Reference to the meeting

```
stproxy.nwaychat.onInvitationAccepted(userId, topic)
```

This is called when the user accepts an n-way chat.

Table of parameters for onInvitationAccepted() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
topic	String	The topic of the chat

```
stproxy.nwaychat.onInvitationDeclined(userId, topic)
```

This is called when the user declines an n-way chat.

Table of parameters for onInvitationDeclined() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
topic	String	The topic of the chat

```
stproxy.nwaychat.onUserJoined(placeId, userId)
```

This is called when the user accepts an n-way chat.

Table of parameters for onUserJoined() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
placeId	String	Reference to the meeting

```
stproxy.nwaychat.onUserLeft(placeId, userId)
```

This is called when the user leaves an n-way chat.

Table of parameters for onUserLeft() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
placeId	String	Reference to the meeting

Livename Model

A livename model is created whenever a new livename is created. However, this is not necessarily the case. The user can create a new livename model, add the user to the watchlist, and then respond to updates on that livename model, managing page updates separate from the UI. To create a livename model you can use:

```
stproxy.getLiveNameModel({ "userId”: userid, "isInBuddyList”:true,
                           "forceWatchlist”: false })
```

Table of parameters for getLiveNameModel() with a description of each parameter.

Parameter	Type	Description
userId	String	The user who started the chat
isInBuddyList	Boolean	TRUE if the person is in the buddyList, FALSE otherwise
forceWatchlist	Boolean	TRUE to add the user to the watchlist, FALSE to not add

Alternatively, simply access the *model* property of a livename.

The properties of the livename model are:

Table of properties liveNameModel with a description of each property.

Property	Type	Description
id	String	The user ID
status	Numeric	The user's status
isExternal	Boolean	TRUE if the user is from a different community

```
LiveNameModel.prototype.onUpdate()
```

This is triggered when new data arrive for the associated user. It is important that you don't simply override this, but instead chain your code to any already-existing code, typically using a call like *dojo.connect()*.

Chat Models

When a chat is initiated, or a similar chat is resumed, a chat model object is created to allow the chat interactions to be controlled more easily. The chat model can be retrieved as follows for the 1-to-1 chat:

```
var chatArgs = { "isAnonymous" : false,
                 "isRichText" : true,
                 "isEmbedded" :  true };
stproxy.getChatModel(userId, chatArgs)
```

Table of parameters for getChatModel() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the chat partner
isAnonymous	Boolean	TRUE if this is a chat with a gues user, otherwise FALSE
IsRichText	Boolean	TRUE if you wish to start a chat using rich text in the messages
IsEmbedded	Boolean	TRUE if the chat is embedded in a page

For an multi-way chat, it is very similar:

```
stproxy.getGroupChatModel(args, isIncoming)
```

Table of parameters for getGroupChatModel() with a description of each parameter.

Parameter	Type	Description
args	Object	*See below*
isIncoming	Boolean	TRUE if this is an invitation, and FALSE if this is started locally

In this case, if *isIncoming* is true, args contains the fields placeId, topic, userId and isAutoJoin ; if it's false, it contains an array of the userIds and the topic.

When the chat ends, you must call *chatModel.close()*.

There are a number of properties associated with a chat model:

Table of properties for chatModel with a description of each property.

Property	Type	Description
is1to1Chat	Boolean	*See below*
isIncoming	Boolean	TRUE if this is an invitation, and FALSE if this is started locally
userId	String	The user's ID
IsRichText	Boolean	TRUE if the chat uses rich text

Since the multi-way chat does not have a specific partner and does not support rich text, these properties are missing in the Group Chat Model:

Table of missing properties for groupChatMdel with a description of each property.

Property	Type	Description
is1to1Chat	Boolean	*See below*
isIncoming	Boolean	TRUE if this is an invitation, and FALSE if this is started locally

The events associated with the chat model are:

```
ChatModel.onMessage(message)
```

Called when a message is received.

Table of parameters for onMessage() with a description of each parameter.

Parameter	Type	Description
message	String	The chat message

```
ChatModel.onFocus(isOpen)
```

Called when the local chat window receives focus.

Table of parameters for onFocus() with a description of each parameter.

Parameter	Type	Description
isOpen	Boolean	Whether the window is already open or not

```
ChatModel.onTyping(isTyping)
```
Called when the user's typing status changes.

Table of parameters for onTyping() with a description of each parameter.

Parameter	Type	Description
isTyping	Boolean	Whether the user is typing or not

```
ChatModel.onClose()
```
Called when the local chat window is closed.

```
ChatModel.onAnnouncement(message, isResponseAllowed)
```
Called when an announcement is received.

Table of parameters for onAnnouncement() with a description of each parameter.

Parameter	Type	Description
message	String	The message
isResponseAllowed	Boolean	Whether the user may respond or not

```
ChatModel.onMeetingInvitation(topic, url)
```
Called when an invitation to a meeting has been received.

Table of parameters for onMeetingInvitation() with a description of each parameter.

Parameter	Type	Description
topic	String	The meeting topic
url	String	The URL to the meeting

```
ChatModel.onPartnerNotActive()
```
Called when the chat partner goes offline.

```
ChatModel.onPartnerActive()
```
Called when the chat partner comes online.

```
ChatModel.onConvertToNway(groupChatModel)
```
Called when the chat is switched to a n-way chat (see below).

Table of parameters for onConvertToNway() with a description of each parameter.

Parameter	Type	Description
groupChatModel	Object	The updated chat model

```
ChatModel.onRichTextData(richText)
```
Called when the chat data message arrives to indicate if the chat is to use rich text.

Table of parameters for onRichTextData() with a description of each parameter.

Parameter	Type	Description
richText	Boolean	Whether rich text is allowed or not

```
ChatModel.onShutDown()
```
Called when the chat is closed.

```
ChatModel.onOpenError(code, error)
```
Called when an error occurs when starting the chat.

Table of parameters for onOpenError() with a description of each parameter.

Parameter	Type	Description
code	Numeric	The error code
error	String	The error message

```
ChatModel.onSendMessageError(message)
```
Called when an error occurs when sending a message.

Table of parameters for onSendMessageError() with a description of each parameter.

Parameter	Type	Description
message	String	The message that failed

```
ChatModel.onInviteUsersError(userIds, topic)
```
Called when an error occurs when inviting users to the chat.

Table of parameters for onInviteUsersError() with a description of each parameter.

Parameter	Type	Description
userIds	String[]	The list of names of users
topic	String	The meeting topic

GroupChatModel

```
GroupChatModel.onMessage(message)
```

Called when a message is received.

Table of parameters for onMessage() with a description of each parameter.

Parameter	Type	Description
message	String	The chat message

```
GroupChatModel.onFocus(isOpen)
```

Called when the local chat window receives focus.

Table of parameters for onFocus() with a description of each parameter.

Parameter	Type	Description
isOpen	Boolean	Whether the window is already open or not

```
GroupChatModel.onTyping(userId, isTyping)
```

Called when the user's typing status changes.

Table of parameters for onTyping() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the user to whom this applies
isTyping	Boolean	Whether the user is typing or not

```
GroupChatModel.onUserLeft(userId)
```

Called when a user leaves the chat.

Table of parameters for onUserLeft() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the user to whom this applies

```
GroupChatModel.onUserJoined(userId)
```

Called when a user joins the chat.

Table of parameters for onUserJoined() with a description of each parameter.

Parameter	Type	Description
userId	String	The ID of the user to whom this applies

```
GroupChatModel.onInviteUsers(userIds)
```
Called when a chat is about to start.

Table of parameters for onInviteUsers() with a description of each parameter.

Parameter	Type	Description
userIds	String[]	The ID s of the user s invited

```
GroupChatModel.onShutDown()
```
Called when the chat is closed.

```
GroupChatModel.onOpenError(code, error)
```
Called when an error occurs when starting the chat.

Table of parameters for onOpenError() with a description of each parameter.

Parameter	Type	Description
code	Numeric	The error code
error	String	The error message

```
GroupChatModel.onSendMessageError(message)
```
Called when an error occurs when sending a message.

Table of parameters for onSendMessageError() with a description of each parameter.

Parameter	Type	Description
message	String	The message that failed

```
GroupChatModel.onInviteUsersError(userIds, topic)
```

Called when an error occurs when inviting users to the chat.

Table of parameters for onInviteUsersError() with a description of each parameter.

Parameter	Type	Description
userIds	String[]	The list of names of users
topic	String	The meeting topic

Chapter 4. Programming the Sametime AJAX Proxy – the User Interface

The Web Client's user interface is based on the Dojo Toolkit. This means that the usual extensibility functions available within that toolkit apply to this UI.

Page setup

The general includes of JavaScript, etc. for the UI are essentially the same as for the base components:

```
<script type="text/javascript">
        var stproxyConfig = {
                server: ""
        }

        var djConfig = {
                parseOnLoad: true
        };
</script>

<script type="text/javascript"
        src="/stwebclient/dojo.blue/dojo/dojo.js"></script>
<script type="text/javascript"
        src="/stwebclient/include.js?widget=widgets"></script>
```

It is a good idea to follow the instructions for the Base Components to ensure that you have the correct structure and initialization. Previous versions of this document described the use of the four central JavaScript files, *livenamejs*, *widgets.js*, *livenameLight.js* and *widgetsLight.js*, as well as the CSS file. While these are still supported, there is a new mechanism to incorporate the necessary JavaScript and CSS using parameters to *include.js*:

Param	Description
widget	*widgets* – all the Sametime widgets *widgetsLight* – the widgets without Dojo bindings *livename* – only the livename *livenameLight* – the livename without Dojo Default – no widgets
auto	*true* – Load the dock widget (Default) *false* – No dock widget
setBiDi	*true* – Set the directionality to right-to-left *false* – Set the directionality to left-to-right (Default)
lang	Override the browser language
noHub	*true* – The application has already loaded the OpenAjax Hub *false* – Load the OpenAjax Hub transparently

Configuration

As well as the setting explained at the start of Chapter 2, other settings controlled from the *stproxyConfig* object are:

- **plugins** – this is a JavaScript Object that contains a list of plugin names to be enabled or disabled. Possible values are listed in the table below.

For example to disable the *File* toolbar menu in the main buddylist window, and the *Remove from Contact list* from the LiveName context menu, and the *Tools* menu in the chat window, you can use:

```
var stproxyConfig = {
        server:  "",
        plugins: {
            "mmpFile": false,
            "cmpTools": false,
            "lnmpRemove": false
        }
}
```

The default behavior is that all plugins are enabled. Using the *stproxyConfig.plugins* above would mean that any subsequent chat window that is opened will have these menus disabled. However *stproxyConfig.plugins* is dynamic, which means that any object that any chat or LiveName widget that is created will use the current state of the *stproxyConfig.plugins* object. For example, to disable the Tools menu in the chat for User A, you would set *"cmpTools": false,* but subsequently creating a chat for User B, the menu can be re-enabled by programmatically setting *"cmpTools": true* before creating the chat.

Table of plugins for UI configuration.

Chat window menu plugins *cmp*	
cmpFile	File menu
cmpClose	File – Close menu entry
cmpTools	Tools menu
cmpCall	Tools – Call menu entry
cmpChatInvite	Tools – Invite menu entry
cmpAddToContacts	Tools – Add to contacts menu entry
cmpMeetingInvite	Tools – Invite to meeting menu entry
cmpHelp	Help menu
cmpDemo	Help – Demo menu item
cmpAbout	Help – About menu item
Live name icon plugins *lnip*	
lnipTelephony	Telephony status icons
lnipAwarenessInternal	Awareness status icons
lnipAwarenessExternal	Awareness status icons for external users
Live name context menu plugins *lnmp*	
lnmpChat	Chat menu item
lnmpCall	Call menu item
lnmpMeetingInvite	Invite to meeting menu item
lnmpSendAnnouncement	Send announcement menu item
lnmpBizCard	Show business card menu item

lnmpAddContact	Add contact menu item
lnmpAddSubgroup	Add subgroup menu item
lnmpAddToContacts	Add to contacts menu item
lnmpRenameGroup	Rename group menu item
lnmpRemove	Remove contact menu item
Main menu plugins *mmp*	
mmpFile	File menu
mmpNewContact	File – New contact menu item
mmpNewGroup	File – New group menu item
mmpLogout	File – Log out menu item
mmpTools	Tools menu
mmpChatInvite	Tools – Invite to chat menu item
mmpMeetingInvite	Tools – Invite to meeting menu entry
mmpPrivacyList	Tools – Privacy menu entry
mmpSendAnnouncement	Tools – Send announcement menu entry
mmpHelp	Help menu
mmpDemo	Help – Demo menu item
mmpAbout	Help – About menu item

The UI Widgets

The individual pieces of the Web Client UI can be incorporated into your application using the usual techniques associated with the use of Dojo. In fact, each UI element is available as a Dojo widget, i.e. a separate reusable UI widget.

When a new widget is created programmatically, it must be inserted on the page to allow the browser to display it. This us typically done using something like:

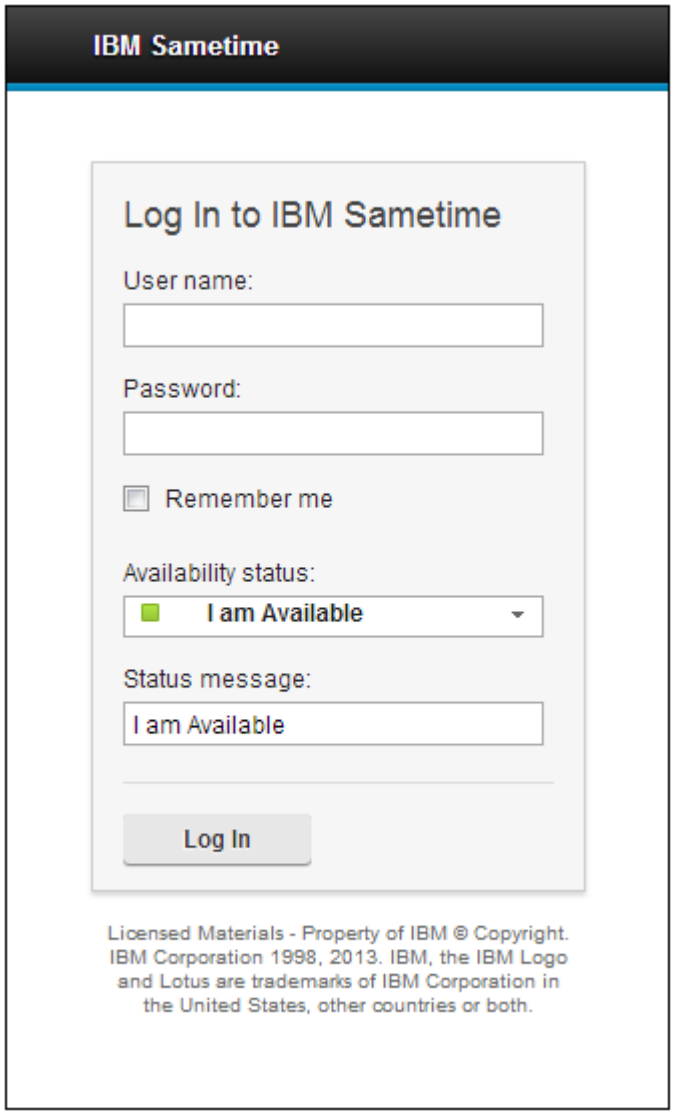
```
dojo.byId("divID”).appendChild(newComponent.domNode);
```

The WebClient

The WebClient widget is used as a complete browser instant messaging client.

Tabble detailing the WebClient widget.

Markup	`<div dojoType="sametime.WebClient"></div>`		
JavaScript	`var client = new sametime.WebClient({}, divID)`		
Parameters	*none*		
Methods	*none*		



WebClient widget

To use this as a buddylist object, set *stproxyConfig.tokenLogin* to true, so this login screen is not displayed when connecting over SSO. This behavior provides a mechanism to enforce login

requirements while facilitating faster login. Note that you should set the required dimensions of the widget using CSS, to ensure that it displays correctly. For example, the following is what was used for the above:

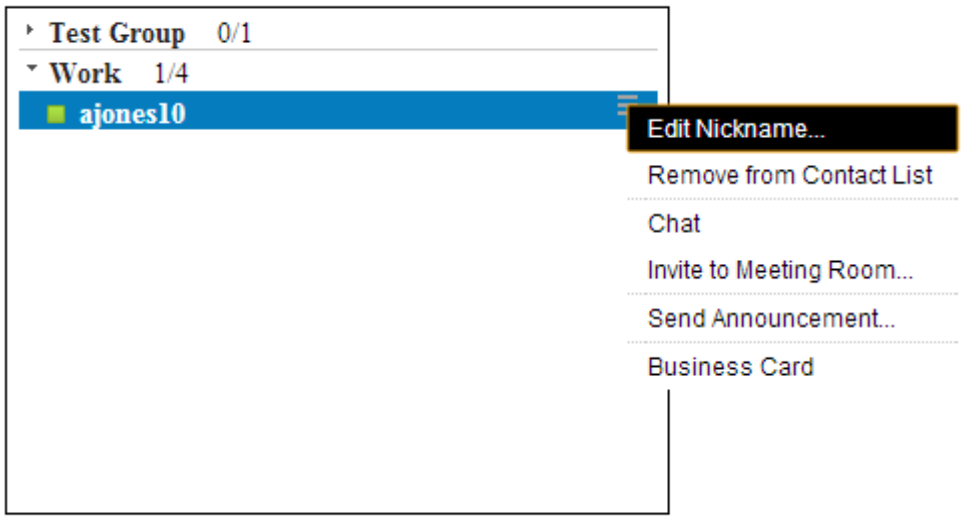
```
style="width:330px;height:550px;border:1px solid black;"
```

The BuddyList

The BuddyList widget displays the current user's list of contacts.

Table detailing the BuddyList widget.

Markup	`<div dojoType="sametime.BuddyList"></div>`
JavaScript	`var client = new sametime.BuddyList ({}, divID)`
Parameters	*none*
Methods	*none*



BuddyList widget

Awareness

The awareness widget is used to manage the user's status:

Table detailing the Awareness widget.

Markup	`<div dojoType="sametime.Awareness"></div>`
JavaScript	`var client = new sametime.Awareness({})`
Parameters	*none*
Methods	`onSet(statusCode, statusMessage)`
	`OnCancel()`
	`onSetError(code, error)`

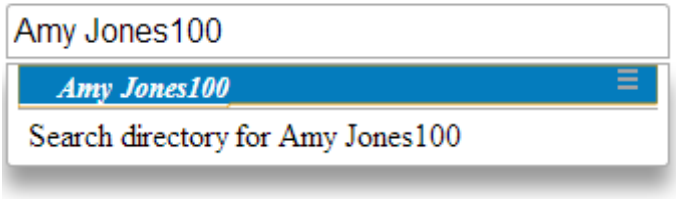
LiveName widget

QuickFind

The QuickFind widget is used to search for a user name:

Table detailing the Awareness widget.

Markup	`<div dojoType="sametime.QuickFind"></div>`
JavaScript	`var client = new sametime.QuickFind ({})`
Parameters	*none*
Methods	`onSelect(widget)` `- called when enter pressed or clicked on a user or group`



QuickFind widget

Note that the QuickFind object when included in this way will only return people, i.e. it will not return groups.

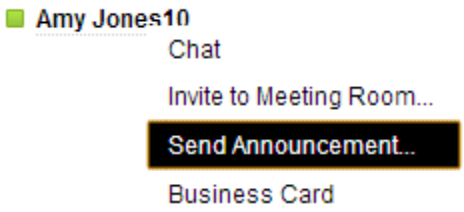
LiveNames

The LiveName widget is used to display a user's status on the page:

Table detailing the LiveNames widget.

Markup	`<div dojoType="sametime.LiveName" userid=”ID”></div>`		
JavaScript	`var LN = new sametime.LiveName ({"userId”: "ID”})`		
Parameters	userId	Required	User ID
	displayName	Optional	The name to be displayed
Methods	activate() - enables the widget deactivate() - disables the widget startChat() - start a chat with the user setDisplaynameText(name) - change the displayed name getDisplayedName() - returns the text of the user name showBusinessCard() - pops up the business card		

Note that if the *displayName* is supplied, this is used as the name displayed in the LiveName. If it is omitted and the user is in the BuddyList, the displayed name from here is used; otherwise the user name is shown in the display.



LiveName widget

Since IBM Websphere Portal v7 and later include support for the Sametime Proxy, you can create LiveNames in your portlets. The generated markup is compatible with the usual person markup, such that they function seamlessly in that environment, providing the expected look & feel.

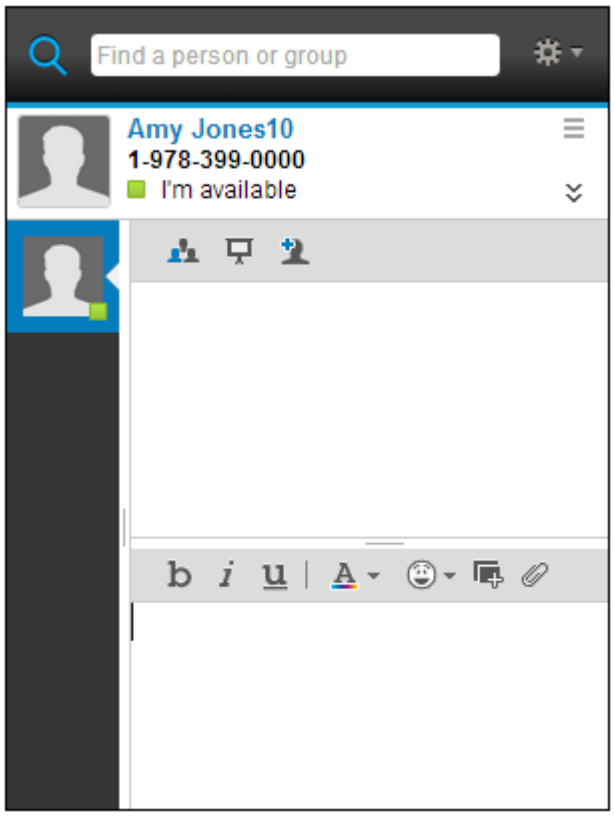
Chat

The Chat widget is used to chat to another user:

Table detailing the Chat widget.

Markup	Declarative creation not supported	
JavaScript	`var myModel = new sametime.ChatModel(userId, args);` `var myChat  = new sametime.Chat({model: myModel });`	
Parameters	args	See the description of ChatModel above for its parameters. An object that can have the following fields: isAnonymous – *TRUE* if the chat is from an anonymous user isEmbedded – *TRUE* if the chat is embedded in a web page isRichText – *TRUE* if the chat should support rich text
Hook-ins		
Methods	Close() - MUST be called when chat is to be closed onClose() - triggered when the chat is closed getAllUserIds() - returns an array of all the participants in a chat addNewChat(model) – Adds a new chat tab using the supplied model addUsers(users, title) – adds the array of users to the chat to promote it to an n-way chat.	

While new chats can be programmatically added to the existing embedded chat as a new tab, incoming chats will open as usual in a separate window.



Chat widget

Group Chat

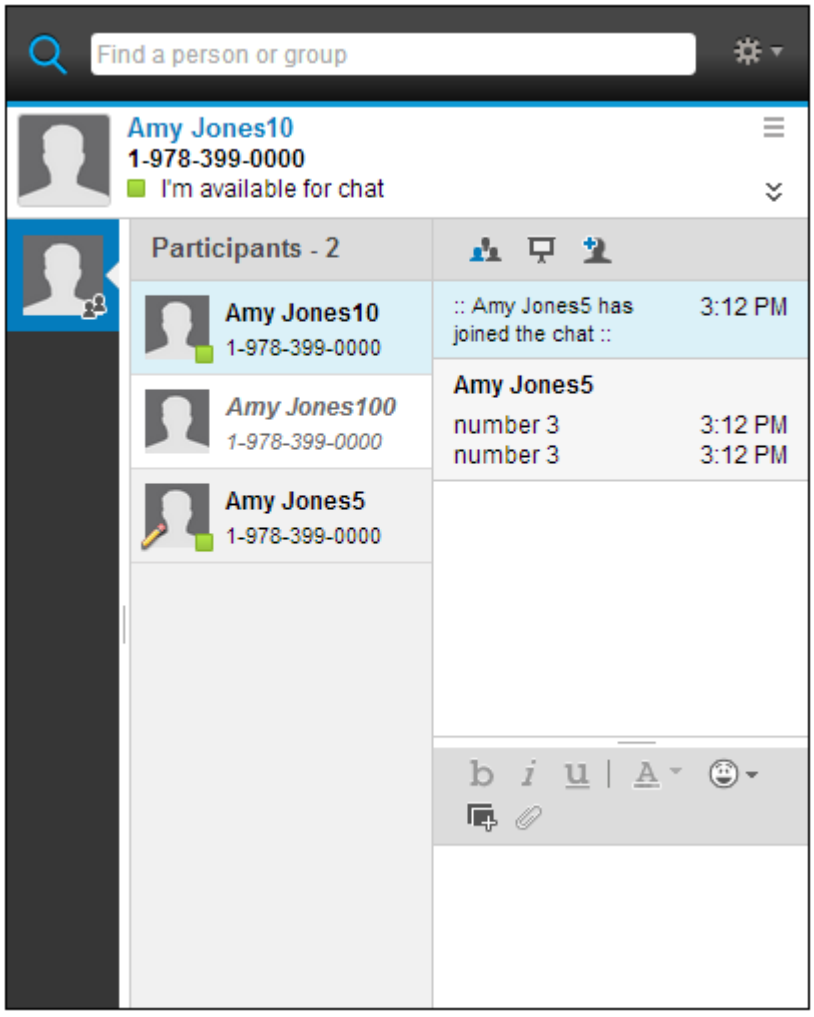
The Group Chat widget is used to chat to a group of users. This is achieved by first creating a chat widget and then inviting the other participants.

Table detailing the Group Chat widget.

Markup	Declarative creation not supported	

JavaScript	`var myModel = new sametime.ChatModel(userId, args);`
	`var myChat  = new sametime.Chat({model: myModel });`
	`myChat.inviteUsers(["tom", "dick", "harry" ], "My Group Chat");`	

Parameters	UserIDs title	An array of user Ids The title of the chat
Methods	See above for 1-1 chat	



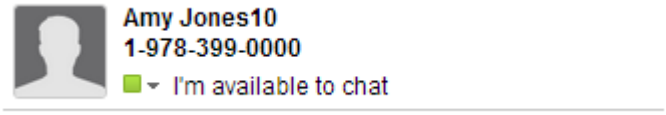
Group chat widget

User Information

This provides the markup that can be put into a popup window, as for the business card below, or within a window, as in the top of the chat window.

Table detailing the User Information widget.

Markup	`<div dojoType="sametime.UserInfo"></div>`		
JavaScript	`var client = new sametime.UserInfo ({})`		
Parameters	userId	Required	The ID of the user for whom you want the info
Methods	addOnLoad(callback) – Executes the callback when the userInfo is loaded toggleDetails() - hide/show the details part of the userinfo.		



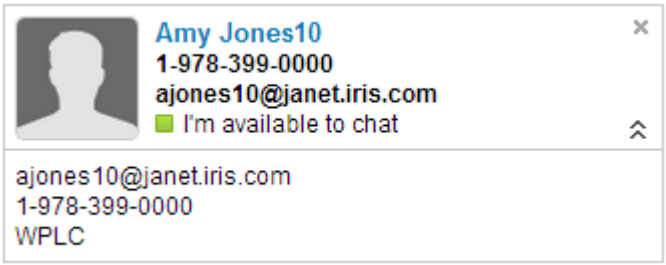
UserInfo widget

Business Card

This displays the User Information within a popup div.

Table detailing the Business Card widget.

Markup	`<div dojoType="sametime.BusinessCard"></div>`		
JavaScript	`var client = new sametime.BusinessCard ({})`		
Parameters	userId	Required	The ID of the user for whom you want to display a business card
Methods	onLoad(callback) – Provides a callback to be executed when the card is loaded open(livename, posX, posY) - pops up the business card for the specified LiveName at the specified position on the screen. close() - closes a visible business card		



BusinessCard widget

Note that you can create a BusinessCard anywhere on the screen by using:

```
stproxy.uiControl.open(userId, positionX, positionY)
```

LiveName Photo

This displays the user's photo, allowing the user's status to be overlaid on the image.

Table detailing the LiveName Photo widget.

Markup	`<div dojoType="sametime.LivenamePhoto"></div>`	
JavaScript	`var client = new sametime.LiveNamePhoto ({})`	
Parameters	userId showStatusIcon	The ID of the user for whom you want to display a photo If the user's presence status should be displayed
Methods	onMessage – This handles how the photo handles messages. The default behavior is to display the number of outstanding messages as a “badge”, i.e. inside a red circle.	

LivenamePhoto widget

Extending UI Widgets

Since the Web Client UI is based on the Dojo Toolkit, the possible extensions of widgets are huge. Below are a few examples of typical extensions.

Note that when adding general functionality to items like menus, timing is critical: the extension must be included into the system before the relevant objects have been instantiated in the page.

Typically, this requires that these *addXXX* functions be executed before the user logs in, and preferably before any Dojo parsing of the page. Typically, the addition is called in-line as the page loads:

```
// Add the menu plugins before anything else ...
stproxy.uiControl.addMainMenuPlugin({
  ...
});

// .. and keep separate from the login step
stproxy.addonLoad(function() {
   stproxy.login.loginByToken( … );
});
```

Adding a menu entry to the main window

To add a new top-level menu to the main window's menu bar, you use the following:

```
stproxy.uiControl.addMainMenuPlugin({
  id:     "myMenuID",
  label:  "MyMenu",
  menuId: "myNewMainMenuID",
  isShowEntry: function(widgets) {
      return true;
  }
});
```

IsShowEntry() is a method used widely in the Sametime code, to provide a mechanism to include or exclude a particular item: if it returns false, the item is not displayed; if it returns true, the item is displayed. In a situation where the display of a menu or similar is determined by some other functionality, this can be calculated dynamically, returning true or false as appropriate.

To add menu items to this menu, you use the same function, but add the *onClick()* function to process the item when it is selected:

```
stproxy.uiControl.addMainMenuPlugin({
  id: "myNewMenuID",
  label: "MyItem Number 1",
  menuId: "myNewMainMenuID",
  isShowEntry: function(widgets) {
      return true;
  },
  onClick: function(mainWindow) {
      alert("You clicked me!!");
  }
});
```

This shows a user's bespoke menu, but extensions can be made to the standard menus by setting the field *menuID* to the appropriate value:

- stproxy_fileMenu
- stproxy_toolsMenu
- stproxy_helpMenu

Adding a menu entry in the chat window

This is essentially the same as adding an item to the main window's menu, but refers to the chat menus instead:

```
stproxy.uiControl.addChatMenuPlugin({
  id: "myNewChatMenuID",
  label:  "My Chat Menu",
  menuId: "stproxy_toolsMenu",
  isShowEntry: function(chat) {
      return true;
  },
  onClick: function(chat) {
      alert("You clicked me!!");
  }
});
```

The *chat* parameter passed into the two methods is the current chat object.

Similar to the main menu, there is a set of standard chat menus, to which you can add extra menu items, referenced by the *menuId* field. These have the same IDs as those in the main menu:

- stproxy_fileMenu
- stproxy_toolsMenu
- stproxy_helpMenu

Adding a menu entry to LiveNames

Again, this follows a similar pattern:

```
stproxy.uiControl.addLiveNameMenuPlugin({
  id: "myMenu"
  label: "My Chat Menu Item",
       // Decide if the entry should be shown
  isShowEntry: function(widgets) {
      this.widgets = widgets; // Save the clicked item(s)
      // Only allow one selection at a time
      if (widgets.length == 1)
            return true;
      else
            return false;
  },
  onClick: function(evt) {
      alert("You clicked me!!");
  }
});
```

The *evt* parameter to the *onClick* method is the click event, containing the usual JavaScript event data items.

Adding a menu entry to a LiveName/group context menu

This is similar to the previous two menu functions:

```
stproxy.uicontrol.addLiveNameMenuPLugin({
  label: "myNewContextMenuEntry”,
  isShowEntry: function(items) {
      // return true if the item is to be displayed in the context
      // menu for the item(s), or false if it is not displayed.
  },
  onClick: function(items) {
      // Process the selected item(s)
  }
});
```

Items is an array, and can be a single selected LiveName or Group, or can contain all groups and LiveNames from a multi-selection.

When working with the menu extensions, there are some useful variables which can be queried in the onClick method. The LiveName widget contains the following fields:

Table listing LiveName widget fields with a description of each field.

Field name	Content
isLiveName	This is set to *true* if the widget is a LiveName, and *false* otherwise, e.g. if the widget is that this is a group.
userId	The ID of the user
resolvedName	The resolved name of the user
displayName	The name of the user in a form suitable for display
model	The liveNameModel associated with this LiveName
disableClicks	Set this to *true* and double-click will not open a chat
disableHoverBizCard	Set this to *true* to prevent the business card opening on hover

Sametime can be queried for the current plugins, and these can be used in your applications. To retrieve a specific plugin, use stproxy.uiControl.getPlugin(id). For example, if you want to only set awareness using an icon rather than a full LiveName, you can use the following:

```
var myModel = stproxy.getLiveNameModel("id");
var statusPlugin = stproxy.uiControl.getPlugin("lnipAwarenessInternal");
var myIcon = dojo.byId("iconDiv");

myModel.onUpdate = function() {  //  or use dojo.Connect()
  var icon = statusPlugin.getIcon(myModel);
  myIcon.setAttribute("src", icon[0]);
  myIcon.setAttribute("alt", icon[1]);
}
```

Adding additional icons to a LiveName

Once again, extending functionality follows the plugin mechanism, with icons being added to the right of the default icons (which are to the left of the name text in a livename).

If you want to indicate extra functionality by means of an icon, you can use the following:

```
stproxy.uiControl.addLiveNameIconPlugin({
  type: Internal-External,
  _iconMapper: {
      value: [ icon-path, status string ]
  },
  getIcon: function(model) {
      return Icon-Path;
  }
});
```

The various parts of this are:

- *Internal-External*
  This is set to *stproxy.pluginType.INTERNAL* if the icon applies to internal users, or *stproxy.pluginType.EXTERNAL* if it applies to users who connect via a gateway, or *stproxy.pluginType.BOTH* for both sets of users.
- value: [icon-path, status string]
  The *value* refers to the possible status values as described above under Login, and the information in the brackets refer to the location of the icon and the associated description of the status.

Typically, the getIcon method uses the status to index into the array of icons. For example, a solution that added an icon to display the status of the LiveName might look something like:

```
stproxy.uiControl.addLiveNameIconPlugin({
  type: stproxy.pluginType.INTERNAL,
  _iconMapper: {
      stproxy.awareness.OFFLINE : [
                    stproxyConfig.server + "/offline.gif",  "Offline" ],
      stproxy.awareness.AVAILABLE : [
                    stproxyConfig.server + "/available.gif","Online" ],
      stproxy.awareness.AWAY : [
                    stproxyConfig.server + "/away.gif",     "Away" ],
      stproxy.awareness.DND : [
                    stproxyConfig.server + "/dnd.gif",      "DND" ],
      stproxy.awareness.IN_MEETING : [
                    stproxyConfig.server + "/meeting.gif",  "Meeting" ],
  },
  getIcon: function(model) {
      return (model.status >= stproxy.awareness.OFFLINE &&
              model.status <= stproxy.awareness.IN_MEETING) ?
            _iconMapper[model.status] : stproxyConfig.server + "/unknown.gif";
  }
});
```

Note that the *model* passed to the *getIcon* method is a *LiveNameModel*, described below.

Extending LiveNames directly

The LiveName behavior and look can be modified by specifying a custom class to manage its display. This class uses a custom *template* that defines the UI layout of the widget, and then this class is specified as the class to be used when displaying the LiveName:

```
dojo.declare("sametime.MyCustomLiveName", sametime.LiveName,
{
  templateString: "<div ...><img dojoAttachPoint='photo'> ...
                   <div dojoAttachPoint='name'> ...
                   <div dojoAttachPoint='statusMsg'> ...",
  postCreate: function() {
    ...
    this.photo.src = getPhotoUrl();
    this.statusMsg.innerHTML = getStatusMsg();
    // No need to populate "name" as it is already done by the parent
  }
});

// Specify that it's to use our class
stproxy.liveNameClass = sametime.MyCustomLiveName;
```

Customizing the chat window

The chat window default action is to open as a pop-up. This can be overridden so that it is embedded in a web page. In this example instead of creating a popup window, the chat windows are incorporated into a tabbed container:

- First, create the tabbed container:

```
<div dojoType="dojo.layout.TabContainer">
    <div dojoType="dijit.layout.ContentPane">
        <div dojoType="sametime.Chat" isEmbedded=”true”></div>
    </div>
</div>
```

- Then override the window-open function to create a new tab panel instead of the popup window:

```
stproxy.createChat = function(partnerId) {
    var chat = new sametime.Chat({partnerId: partnerId});
    var tab = new dijit.layout.ContentPane({closable: true,
          title:stproxy.STProxy.getDisplayName(partnerId)});
    tab.containerNode.appendChild(chat.domNode);
    tabContainer.addChild(tab);
    tabContainer.selectChild(tab);

    return chat;
}
```

Customizing the group chat window

The chat window default action is to open as a pop-up. This can be overridden so that it is embedded in a web page. In this example instead of creating a popup window, the chat windows are incorporated into a tabbed container:

- First, create the tabbed container as for the one-to-one chat.
- Then override the window-open function to create a new tab panel instead of the popup window:

```
stproxy.createChat = function(partnerId) {
    var chat = new sametime.GroupChat({partnerId: partnerId});
    var tab = new dijit.layout.ContentPane({closable: true,
          title:stproxy.STProxy.getDisplayName(partnerId)});
    tab.containerNode.appendChild(chat.domNode);
    tabContainer.addChild(tab);
    tabContainer.selectChild(tab);

    return chat;
}
```

Associating additional custom data with a user

The data associated with a user or users in the watchlist can be augmented with data from other systems. For example, you might retrieve data from your HR system, and wish to have it associated with the LiveNames on a page so that these extra data items can be displayed in a customized business card.

Adding the data is quite simple – call the onUpdate method with the new data:

```
stproxy.watchList.onUpdate({"id": myUserId, "myData1": 100,
                            "myData2": "foo", ... });
```

Chapter 5. Style classes

Changing the look and feel of the Sametime Web Client is simply a question of applying standard CSS styles. You can easily apply your own styling by overriding a specific style property. Note that the web client can also be run in disconnected mode, to allow you to test any styling changes.

The default styling is defined in a set of CSS files in the directory:

```
SametimeProxy.ear/stproxyweb.war/dojo.blue/sametime/themes
```

Images

There are a number of images associated with the user display, indicating the user's current status. These are all contained in the *stproxy.uiControl.iconPaths* object and can be replaced to provide an alternative status display. They are:

Table listing images associated with the user display.

iconAvailable	Displayed when the user is available
iconAway	Displayed when the user is away
iconDnd	Displayed when the user does not want to be disturbed
iconInMeeting	Displayed when the user is in a meeting
iconOffline	Displayed when the user is offline

There are equivalent entries for when the user is on a mobile device:

- iconAvailableMobile
- iconAwayMobile
- iconDndMobile
- iconInMeetingMobile

There are also equivalent graphics for external communities:

Table listing names of equivalent images used for external communities.

Google	*AOL*
iconGTalkAvailable	iconAOLAvailable
iconGTalkAway	iconAOLAway
iconGTalkDnd	iconAOLDnd
iconGTalkInMeeting	iconAOLInMeeting
iconGTalkOffline	iconAOLOffline

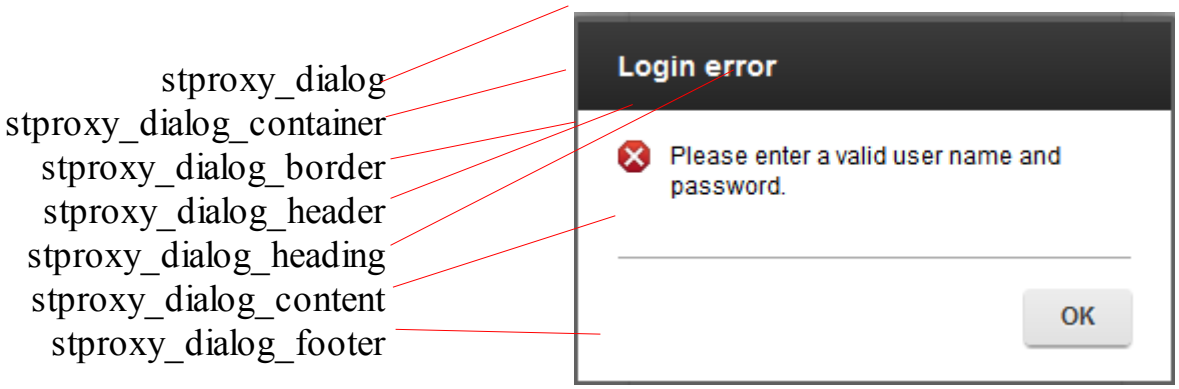
These icons should be changed using JavaScript; the styles mentioned below do not update the icons.

Cheat Sheets

Styling cheat sheets are provided for each of the widgets, to help you identify the style name associated with the various parts of the widget. All UI elements have *stproxy_widget* as their root style. Many of the widgets share common elements, and these have styles that apply across all parts of the product:

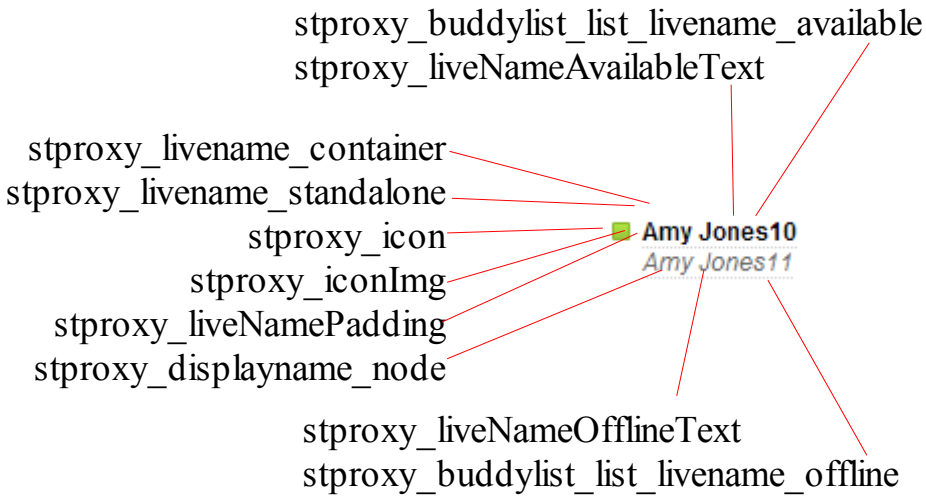
Dialog

The dialog container is reused across the web client, so its styling is important:

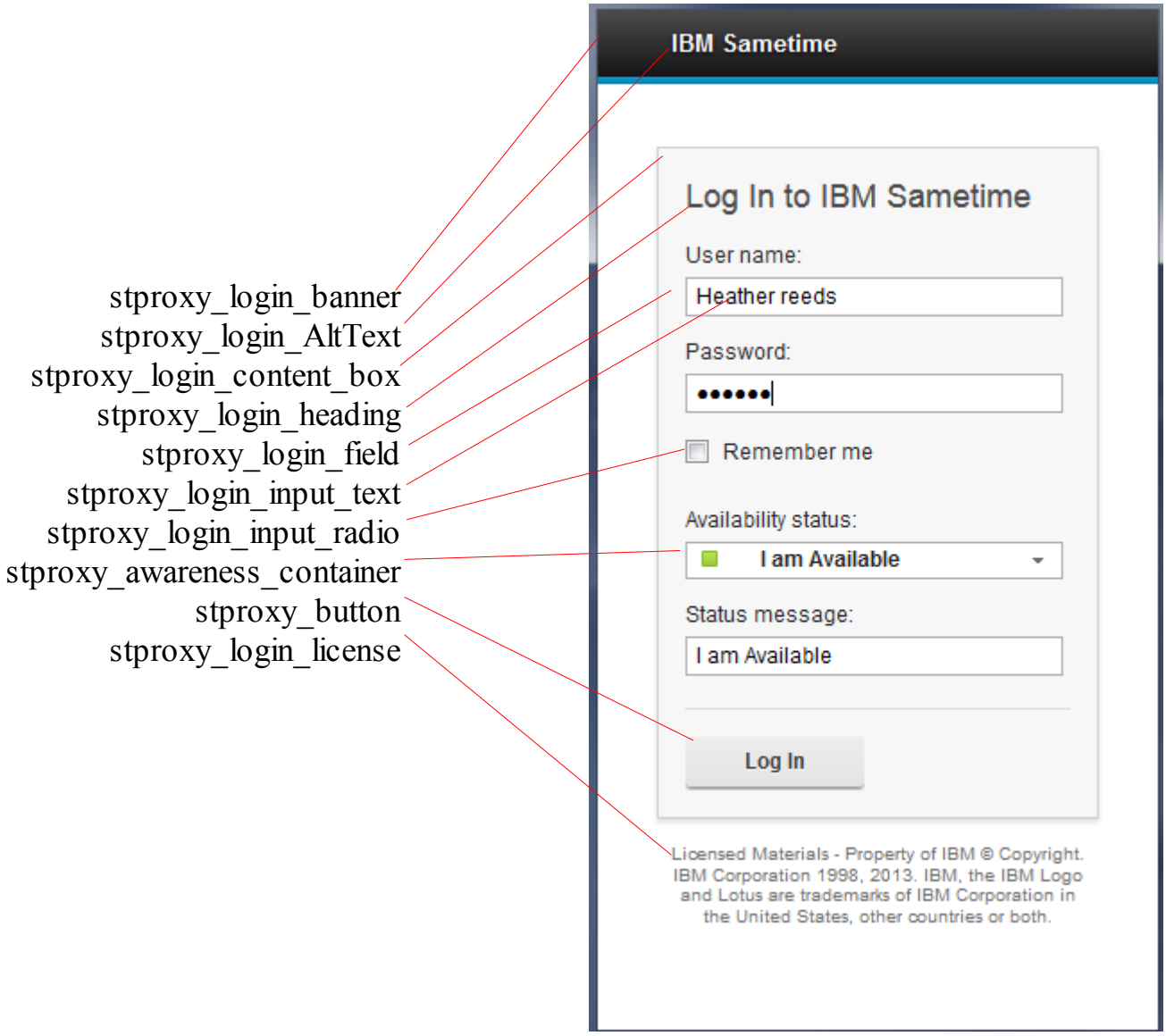


LiveName

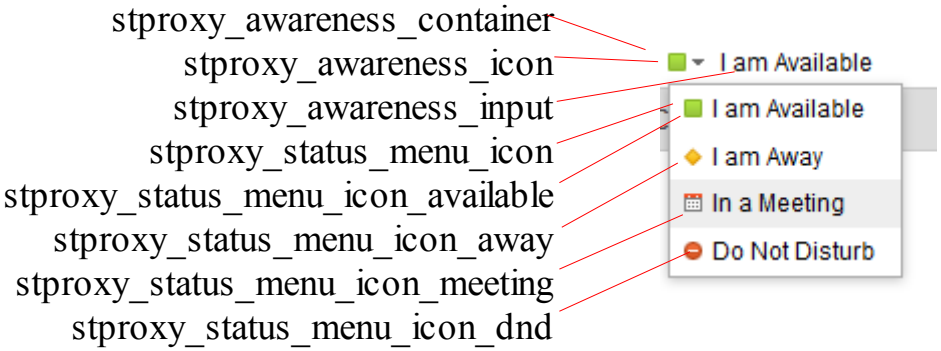
Similarly, LiveName items are used in many situations:



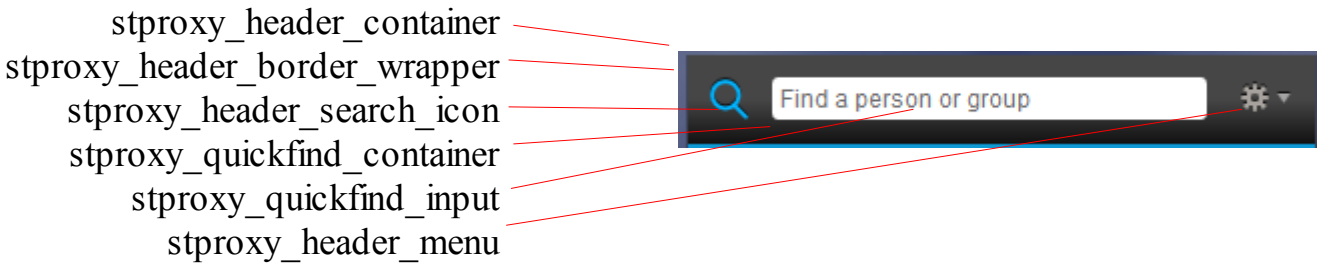
Login


stproxy_login_banner
stproxy_login_AltText
stproxy_login_content_box
stproxy_login_heading
stproxy_login_field
stproxy_login_input_text
stproxy_login_input_radio
stproxy_awareness_container
stproxy_button
stproxy_login_license
IBM Sametime
Log In to IBM Sametime
User name:
Heather reeds
Password:
Remember me
Availability status:
I am Available
Status message:
I am Available
Log In
Licensed Materials - Property of IBM © Copyright. IBM Corporation 1998, 2013. IBM, the IBM Logo and Lotus are trademarks of IBM Corporation in the United States, other countries or both.

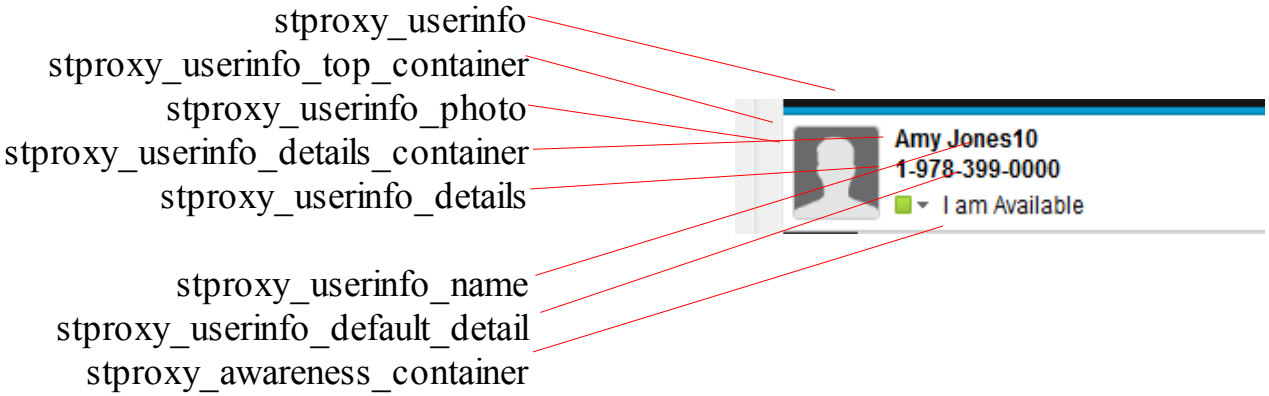
User Presence Status



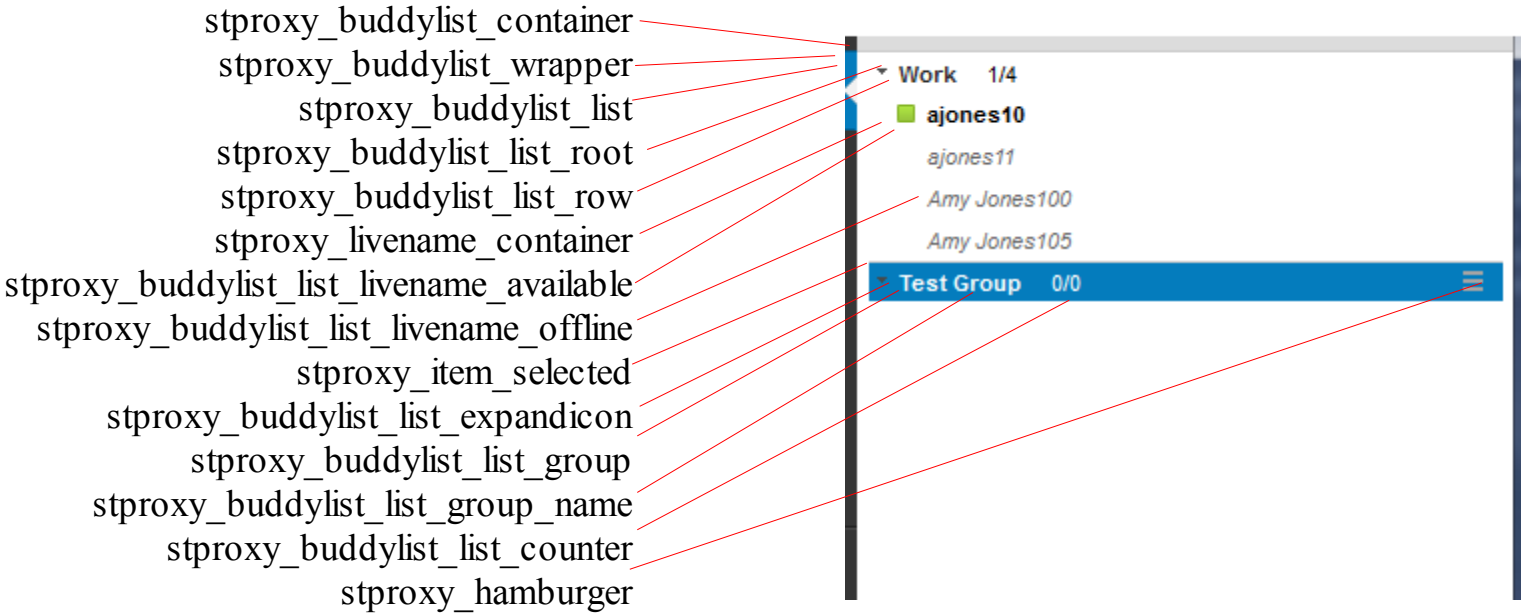
QuickFind



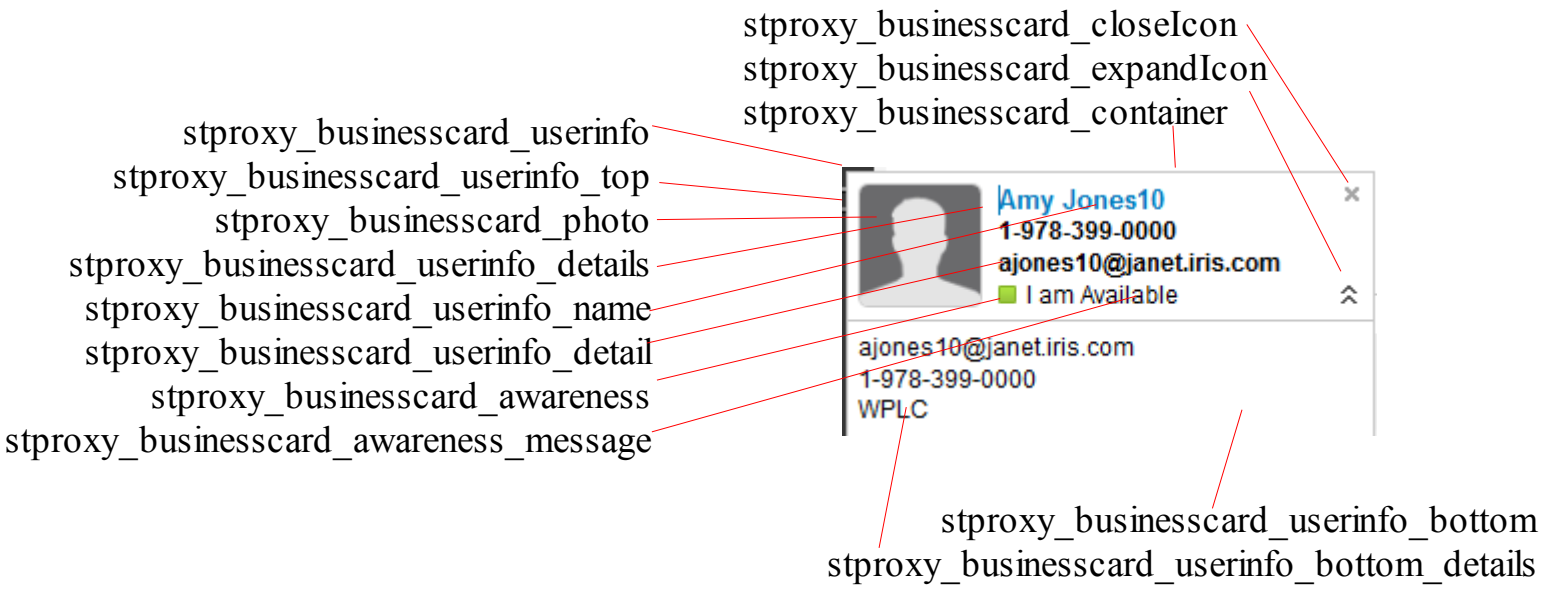
UserInfo



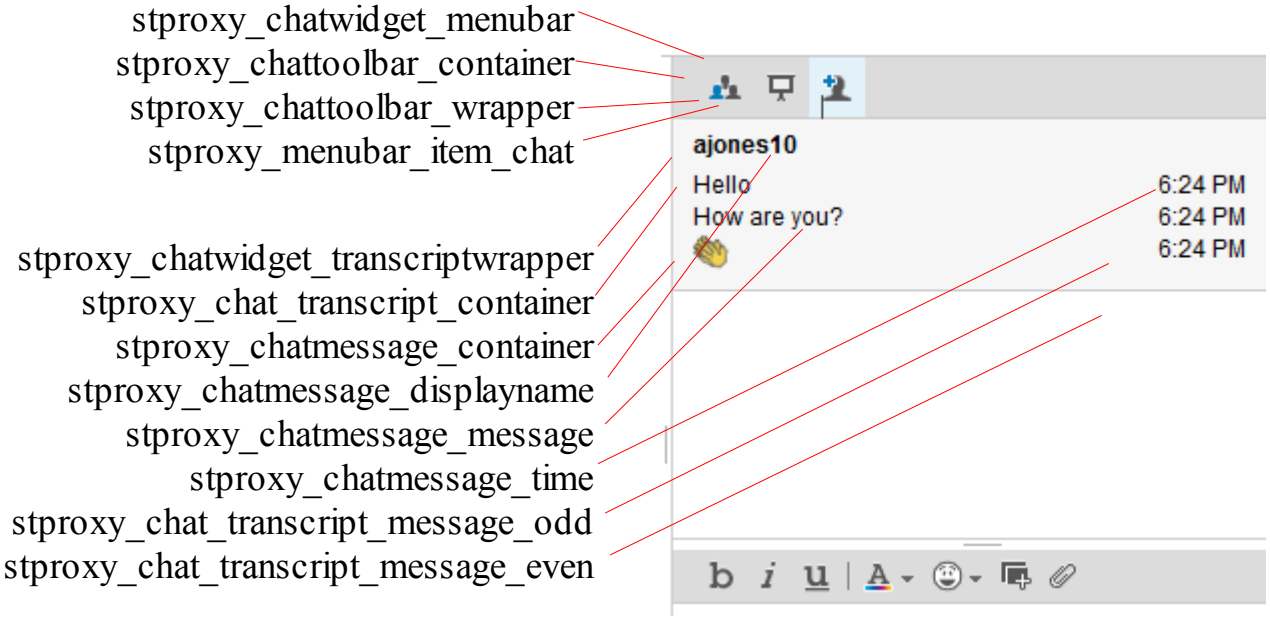
Buddylist



Business Card



Chat Window



Chapter 6. Using the REST API

Introduction

Representational State Transfer (REST) refers to the simple interfaces that transmit data over HTTP, with no additional messaging layer. The Sametime 9 Browser client SDK exposes functionality using a REST API, returning data in the JSON format as the HTTP content type *application/json*. While utilizing a REST API usually simply requires the calling of a URI and inspecting the returned HTML return code and associated data, the specific requirements of maintaining information on contacts and live names requires a little more work than usual: when a REST call is initiated, the response is retrieved via a technique known as long poll which is described below.

REST maps traditional data manipulation concepts to HTTP verbs in a consistent and simple manner:

Table listing REST mapping to HTTP verbs

Traditional	HTTP
Create	POST
Read	GET
Update	PUT
Delete	DELETE

The data in Sametime 9 Browser client can be accessed by any mechanism that supports HTTP, and since JSON is used as the data format for the returned data, you can use this API from any program that can send and receive this formats over HTTP. Your program can process the data itself, or use third party libraries such as Json.NET, to assist with the task. In other words, programs to access the data can be created using markup like DHTML in browsers, desktop-web hybrids such as Adobe AIR, Microsoft Silverlight and Mozilla Prism, scripting languages like Perl and Python and traditional programming languages like Java, C++, C#, etc.

Please note that using the Sametime Proxy's REST API requires an in-depth knowledge of programming for the web in the chosen language; it also requires a thorough understanding of the concepts used in HTTP server push techniques. For example, someone wishing to use Java should be expert in the use of the java.net and java.lang.Thread packages, and should thoroughly understand how AJAX push or Comet protocols function.

Parameters

Parameters are designated as required or optional. Superfluous parameters are ignored. If multiple instances of a parameter are specified, the value of the parameter that is last in the URI is used. Parameter names are CASE SENSITIVE. Omitting a mandatory parameter results in a Bad Request error (400).

Default values

If an optional parameter is omitted, its default value is used when a new resource is created, using the PUT method. In the case where an existing resource is updated, the value of any parameter that is not specified remains unchanged.

Type definitions

Parameters are passed in the HTTP request, and are always of type String. However, certain parameters expect specific content in these strings. Incorrect data results in an error being returned in the form of an HTTP error of 400 (Bad Request).

The following is a list of many of the data types that may be used in your applications:

Table listing samples of data types that can be used in your application.

Type	Description
String	A string
Integer	Only integral numeric values permitted
Boolean	Only the values TRUE and FALSE permitted (case insensitive)
User	The user's Email address
Date	A date in the format MM/DD/YYYY for example: 12/30/2009
Double	Only numeric values permitted

Arrays

Arrays are passed as a string array of concatenated values, where the elements of the array are separated by the separator character "," (comma). An empty array is expressed by the null string. Arrays are indicated in the parameter types below as the type followed by square brackets; for example:

```
String[ ]
```

Returned information

The data that are returned in response to a request are delivered via the long poll mechanism, not in the response to the initial request, the response to which is a simple acknowledgement which can contain any of the usual HTTP codes. These responses have the following format:

```
{"returnCode":200}
```

Long poll

While using Ajax allows a web application to retrieve data from the server, it does not allow the server to push data to the application. This means that the application can request an update for a particular set of data on the screen, but the server cannot simply send this update independently. The problem with this is that any regular polling performed by the application is expensive in terms of resources.

A long poll is a mechanism to allow the server to send data at any time to the application client. The way it works is that the application sends an asynchronous GET request to the server, at the address:

```
 /stwebapi/RTCServlet?format=json
```

so that the server can respond whenever it wants to. After the response is processed (or times out) the client issues another request and waits for the server again.

Note that this means that the application has one permanently open connection to the server, with occasional connections also calling the REST services. Since some browsers have a limit of a maximum of two open connections to a server, this must be considered when creating the application.

The long poll returns the data fom the original asynchronous request, and these data have the following format, illustrated here by the example of adding a user to the watchlist. The data include internal IDs, and timestamps, and each response message can contain the responses to a number of asynchronous requests.

```
{"update":
  [{  "sid":"2dd6b243-bf93-4f4b-93a5-b7461045598d",
      "op":"add",
      "cn":"2dd6b243-bf93-4f4b-93a5-b7461045598d",
      "type":"channel",
      "key":"",
      "value":{
         "action":["watchlist","addUser"],
         "userId":"ID",
         "displayName":"A.N.Other",
         "returnCode":200
       },
       "time":1273249695607
    },
    { "sid":"2dd6b243-bf93-4f4b-93a5-b7461045598d",
      "value":{
         ...
     ok    "returnCode":200
       },
      "time":1273249695607}
   ]
}
```

While it is probably unnecessary to process some of these fields, below is a description of the general content. The list contains the constant string "update” followed by an array of update entries.

Field	*Type*	*Content*
sid	String	Session ID
op	String	The operation that was executed, which can be: "add" - the item was added "change" - the item was changed
cn	String	This usually contains the *sid* value again, but it can also contain the value "UserMap" for map data.
type	String	• "channel" - indicates data returned from the server • "map" - refers to updating an entry in the user map
key	String	If the *type* is set to "map", this contains the value of an item in the user data map, otherwise it is an empty string.
value	Value Array	An array of responses
time	TimeStamp	Indicates the time of the message

Table detailing Value

Field	*Type*	*Content*
value	String Array OR String	Contains two strings: • The object being actioned (also if one string) • The action being executed
Value data	Object Array	The data associated with the value – these depend on the value.
Return Code	Number	The HTTP return code - typically "200"

Managing the long poll

Understanding the sequence of behaviors to manage the log poll is important. It is important that the poll itself is run in a parallel process to the rest of the application processing. In a browser, this is typically achieved by means of a loop which makes a non-blocking asynchronous call to the relevant URL

There are three phases in the long poll sequence: start, loop and stop.

Starting the poll

- After logging into Sametime by calling POST on the */stwebapi/user/connect* REST API (see below), a JSON response is received, which contains the Sametime session identifier (*sid*), something like:

```
{"LtpaToken":"...,"sid":"sessionid-value", … }
```

As part of the response to the login call, a number of cookies are returned:

- **sid** – The Sametime session identifier

- **loginName** – the login name by which the user is known to the system
- **ltpaToken** – The authentication cookie from the system. Note that there may be variants on the authentication cookie name.
- **JSESSIONID** – The Websphere server session ID. This can be reconfigured as a different name in Websphere, in which case the name is adjusted accordingly.
- **SelectorMap** – This is actually returned by the long poll.
- **HeadUpdaterWindowName** – This cookie is used exclusively by the base components to identify the window which is currently managing the communications to the server.

- Use the value of the *sid* to construct the URL to connect to the long poll servlet, again using POST to the URL, providing the username and setting the method parameter to *put*.

```
/stwebapi/RTCServlet/sessionid-value/user?userName=username&method=put
```

- This returns a JSON response which contains the *Rtc4web-Nonce* value.

```
{"Rtc4web-Nonce":"3361d283-066b-4d38-87a2-609618986c22",
"loginId":"A.N.Other","isAnon":true}
```

The GET loop

- The long-poll servlet is continuously called using a GET to the servlet's URL The value of the Rtc4web-Nonce should be set in subsequent calls to the long-poll servlet, as an HTTP request header value. Calls should also have a timeout of 30 seconds.

```
/stwebapi/RTCServlet?format=JSON
```

- When the request times out, the GET is repeated; if the request returns data, these data are processed and the GET is repeated. Note that to ensure that the data are not retrieved from the browser cache, it may be necessary to add a timestamp parameter to the URL, i.e. by adding &dojo.preventcache=*timestamp* to the end of the URL

Stopping the poll

- When the user logs out, it the long poll is stopped by calling PUT on the update:

```
/stwebapi/RTCServlet/sessionid-value/endUpdate
```

- Finally, send a DELETE:

```
/stwebapi/RTCServlet/sessionid-value/user?userName=username&broadcast=false
```

For more information on this technique, it is recommended that the user read available information on its use in the *Comet protocol* and similar solutions. A full description of this area is beyond the scope of this SDK documentation.

Calling REST

URL format

The REST API calls all have the same basic format:

```
{HOST-SEGMENT}/stwebapi/{RESOURCE_TYPE}[/IDENTIFIER][?PARAMETERS]
```

Where:

- *HOSTSEGMENT* refers to the server protocol, name and port
- *RESOURCE_TYPE* specifies the kind of API call, e.g. chat, buddylist, etc.
- *IDENTIFIER* refers to the specific object being accessed
- *PARAMETERS* is the list of parameters on the URI

The maximum length of a URL is effectively 2083 characters, since this is the maximum size that Microsoft Internet Explorer will accept (the limitations for Firefox, Safari and Opera are over 100000 characters). In other words, when you send a message to the server, i.e. make a REST API call, the total size of the string must be less than 2083 characters after encoding.

Note that when the Sametime Proxy Server is in the same domain as the web application, the REST API supports the usual *XmlHttpRequest* HTTP POST and GET operations, while PUT and DELETE operations are supported by POST, since some browsers do not directly support PUT and DELETE. If the Proxy Server and the application are in different domains, i.e. using cross-domain scripting, the HTTP verbs PUT, DELETE and POST must be sent as a GET operation with the parameter *method=”POST”*.

Response

The response to the REST call can be either a status indicating the state of the request, or a message object that contains the result with the following format:

Table listing attributes that can be used in the response to a REST call.

Attribute	Type	Description
messageType	Integer	Identifies the type of the request for which this response is generated
isStatus	Boolean	If true, message contains return status
Message	JSON	JSON name/value pair(s) that contains response. Return status is expressed as a single name/value pair, with the value containing the status code as below.

Functionality available through REST APIs

Since the API call is expressed as a URI, data can be read using any language that can process URIs and HTTP requests. Processing JSON data is described very well at http://www.json.org/.

```
private String getRequestData(String url) {
    // Return this string
    String strJSON;

   // Use Apache Commons httpclientfor simplicity
   HttpClient httpClient = new DefaultHttpClient();
    try {
        HttpGet httpGet = new HttpGet(url);
        ResponseHandler<String> responseHandler = new BasicResponseHandler();
        strJSON = httpClient.execute(httpGet, responseHandler);
    } finally {
        // Release resources
        httpClient.getConnectionManager().shutdown();
    }
    return strJSON;
}
```

Here the Apache Commons *Httpclient* is used to show how a Java program might access the JSON returned from an API call. Note that this is very simplistic and is merely meant to illustrate the concepts: a production program would be substantially more complex, e.g. it would typically catch exceptions.

REST API Security Considerations

Authentication is required in order to use the REST APIs. When the API call uses SSL, i.e. the protocol of the URI is HTTPS, the behavior of the connection is slightly different. While the API will generally behave so that the extra layer of security is transparent, the connection may fail security checks at an application level

The individual REST API calls are enumerated in Appendix A.

Appendix A. REST API Reference

The following sections describe the REST APIs in detail.

Community Service

These APIs handle the community operations and events relevant to the local user

Login

Table detailing Login request.

Request:	
URI	***REST verb***
/user/connect	POST

Table detailing Login request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
username	String	Optional	The user name
password	String	Optional	Password
loginType	String	Optional	Type of login client: this can be any of: • stproxy.client.WEB_CLIENT • stproxy.client.MOBILE_CLIENT • stproxy.client.MEETINGS_CLIENT • stproxy.client.INOTES_CLIENT • stproxy.client.OTHER
community	String	Optional	The community to which you want to connect
initialStatus	User Status	Optional	Initial status value (see *User Status*)
initialStatusMessage	String	Optional	Initial status message
loginMethod	String	Optional	The type of login, one of: • stproxy.loginType.AS_ANON • stproxy.loginType.BY_TOKEN • stproxy.loginType.BY_PASSWORD

The response from the login call is one of only two that directly return data in the response – all others return the data over the long poll.

Table detailing Login response fields.

Field	***Type***	***Content***
sessionid	String	The session ID of this client
loginName	String	The user's login name
reason	String	The login status, "0” on success
SametimeToken	String	The Sametime session token
version	String	The version number of the Sametime Proxy Server
telephony	Boolean	Whether or not telephony is enabled for this user
person	Person Data	Description of the user (see below)
ServerAttributes	Attributes Array	Describes the attributes supported by the server (see below)
UserPolicies	Policies Array	The policies associated with the user (see below)

Table detailing Person Data fields.

Field	***Type***	***Content***
status	Number	The logged in status of the user (see *User Status*)
StatusMessage	String	The user's status message
id	String	The user's ID

Table detailing Server Attribute fields.

Field	***Type***	***Content***
value	String	The attribute value expressed as a string, i.e. Boolean and numeric values are enclosed in quotes.
type	String	The attribute type which can be: • "1” - String • "2” - Boolean ("true” or "false”, "1” or "0”) • "3” - Numeric
id	String	The attribute ID

Table detailing Server Attribute and User Policy fields.

Field	***Type***	***Content***
value	String	The attribute or policy value expressed as a string, i.e. Boolean and numeric values are enclosed in quotes.
type	String	The attribute or policy type which can be: • "1” - String • "2” - Boolean ("true” or "false”, "1” or "0”) • "3” - Numeric
id	String	The attribute or policy ID

Logout

Table detailing Logout request.

Request:	
URI	***REST verb***
/user/connect	DELETE

Table detailing Logout request parameters.

Request parameters: NONE

Set Status

Table detailing Status request.

Request:	
URI	***REST verb***
/user/status	POST

Table detailing Status request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
status	User Status	Optional	The new status (see *User Status*)
statusMessage	String	Optional	Status description

Table detailing Status response fields.

Name	***Type***	***Content***
messageType	String	MY_STATUS_CHANGED
isStatu	Boolean	If true, status instead of reason
Message	Person	Status message , e.g. "I am away”

People Service

These APIs handles the operations and events related to the user’s contact list (buddy list) and online status, and any changes to the location information of the users added to the Watch List.

Retrieve BuddyList

To get the user’s entire buddy list:

Table detailing Retrieve BuddyList request.

Request:	
URI	***REST verb***
/buddylist	GET

Table detailing Retrieve BuddyList request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
isWatchList	Boolean	Optional	Whether the contacts should be added to the watch list
isWatchLocation	Boolean	Optional	Whether location information should be returned

Table detailing Retrieve BuddyList response fields.

Name	***Type***	***Content***
value	String Constant	"buddylist”
isStatus	Boolean	If true, status instead of reason
buddylist	Group Array	One or more groups and their members

Table detailing Group descriptor fields.

Name	***Type***	***Content***
type	String	This has the value • "public” - this is a public group • "private” – this is a private group
id	String	The group ID.
displayName	String	The group's display name
children	Group or Person array	An array of subgroups which have the same format as a group, or of members of the group. Note that public groups do not return the members until required.

Table detailing Person Description fields.

Name	***Type***	***Content***
communityId	String	This is usually empty, but if the user is from a different Sametime Community, it will contain the community's ID
isExternal	Boolean	This indicates if the user is an external user or not
id	String	The user's ID, qualified by the community ID, separated by "::”
contactId	String	The user's ID, without the community information
displayName	String	The user's display name

Retrieve all users

Table detailing Retrieve Users request.

Request:	
URI	***REST verb***

/buddylist/users	GET

Table detailing Retrieve Groups request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
isPrivate	Boolean	Required	*TRUE* to only retrieve users in private groups, *FALSE* to retrieve public users

Table detailing Retrieve Users response fields

Name	***Type***	***Content***
value	String Array	["buddylist", "AllPrivateUsers"] or ["buddylist", "AllPublicUsers"]
AllPrivateUsers or AllPublicUsers	Users Array	Array of user descriptors (See above)

Retrieve all groups

To retrieve either all public or all private groups:

Table detailing Retrieve Groups request.

Request:	
URI	***REST verb***
/buddylist/groups	GET

Table detailing Retrieve Groups request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
isPrivate	Boolean	Required	*TRUE* for all private groups, *FALSE* for all public groups.

Table detailing Retrieve Groups response fields.

Field	***Type***	***Content***
action	String Array	["buddylist", "AllPrivateGroups"] or ["buddylist", "AllPublicGroups"]
AllPrivateGroups or AllPublicGroups	Group Array	An array of group descriptors (see above)

Add a user

Table detailing Add User request.

Request:	
URI	***REST verb***
/buddylist/user	POST

Note:

- Since users can not be added to public groups, this refers to private groups only.
- Adding a new user to a private group which does not already exist results in that group being created, and the user being added to it.

Table detailing Add User request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
userId	String	Required	The ID of the user to add
groupId	String	Required	The ID of the group to which to add the user

Table detailing Add User response fields

Name	***Type***	***Content***
userId	String	The ID of the user added
action	String Array	["buddylist", "addUser"]
groupId	String	The ID of the group to which the user was added

Delete a user

Table detailing Delete User request.

Request:	
URI	***REST verb***
/buddylist/user	DELETE

Note:

- Since users can not be deleted from public groups, this refers to private groups only.

Table detailing Delete User request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
userId	String	Required	The ID of the user to delete
groupId	String	Required	The ID of the group from which to delete the user

Table detailing Add User response fields

Name	*Type*	*Content*
userId	String	The ID of the user deleted
action	String Array	["buddylist", "removeUser"]
groupId	String	The ID of the group from which the user was deleted

Rename a user

Table detailing Rename User request.

Request:	
URI	***REST verb***
/buddylist/user	PUT

Table detailing Rename User request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
userId	String	Required	The ID of the user to rename
newUserDisplayName	String	Required	The new display name

Table detailing Add User response fields

Name	*Type*	*Content*
action	String Array	["buddylist", "renameUser"]
userId	String	The ID of the user who was renamed
newUserDisplayName	String	The new display name

Add a Group

To add a group to the user's buddy list:

Table detailing Add Group request.

Request:	
URI	***REST verb***
/buddylist/group	POST

Table detailing Add/ Group request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
isPrivate	Boolean	Required	*TRUE* for a private group, *FALSE* for public
groupId	String	Required	ID of the group to be added

Table detailing Add Group response fields.

Field	***Type***	***Content***
action	Int	["buddylist", "addGroup"]
isPrivate	Boolean	*TRUE* for a private group, *FALSE* for public
groupId	String	ID of the group added

Delete a group

To add or remove a group to/from the user's buddy list:

Table detailing Delete Private Group request.

Request:	
URI	***REST verb***
/buddylist/group	DELETE

Note that deleting a private group will also remove the users in that group.

Table detailing Delete Group request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
isPrivate	Boolean	Required	*TRUE* for a private group, *FALSE* for public.
groupId	String		

Table detailing Delete Group response fields.

Field	***Type***	***Content***
action	String Array	["buddylist","removeGroup"]
isPrivate	Boolean	*TRUE* if this is a private group, *FALSE* for public
groupId	String	The group's ID

Rename Group

Table detailing Rename Group request.

Request:	
URI	***REST verb***
/buddylist/group	PUT

Table detailing Rename Group request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
newGroupId	String	Required	The new name for the group
oldGroupId	String	Required	The old name for the group

Table detailing Rename Group response fields.

Field	***Type***	***Content***
action	String Array	["buddylist","renameGroup"]
oldGroupId	String	Original name
newGroupId	String	New name

Send Announcement

The following URI is used to send an announcement to a group of buddies:

Table detailing Send Announcement request.

Request:	
URI	***REST verb***
/buddylist/alerts	POST

Note that all users on the list receive the announcement.

Table detailing Send Announcement request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
receivers	String	Required	Concatenated list of userIds, separated by vertical solidus ("|")
message	String	Required	Text of message
isAllowed	Boolean		Whether the recipient is allowed to respond

Table detailing Send Announcement response fields.

Field	***Type***	***Content***
action	String Array	["announcement", "received"]
isResponseAllowed	Boolean	If true, allow recipients to respond
senderId	String	ID of the user who sent the message
displayName	String	The display name of the sender
message	String	The message content

QuickFind

This API searches for the specifed user(s) and/or group(s), returning a list of the items found.

Table detailing QuickFind request.

Request:	
URI	***REST verb***
/quickfind	GET

Table detailing QuickFind request parameters.

Request parameters:			
Name	***Type***	***Required***	***Description***
searchString	String	Required	The name to be searched
isUser	Boolean	Optional	If this is *true* the search returns only users If this is *false* the search returns only groups If this is omitted, the search returns both users and groups.

Table detailing QuickFind response fields.

Field	***Type***	***Content***
action	String Array	["quickfind", "find"]
searchStiring	String	The search argument
result	String Array	"Groups": or "Persons" followed by an array of search responses.

Table detailing Person response fields.

Field	***Type***	***Content***
userName	String	The returned user's display name
contactId	String	The user's ID

Table detailing Group response fields.

Field	***Type***	***Content***
Type	String	The group type
id	String	The group's ID
Description	String	The group description

WatchList

These APIs handles the operations and events related to the user's WatchList, i.e. the list of users whose status the user is monitoring.

WatchList Updates

When the watchlist changes, a message is sent through the long poll with the update information.

Table detailing WatchList update fields.

Field	***Type***	***Content***
action	String Array	["watchlist", "watchlist"]
watchlist	User array	Array of user watchlist statuses

Table detailing WatchList status fields.

Field	***Type***	***Content***
status	Numeric	Presence status value
statusMessage	String	User's status message
id	String	The user's ID

Add user(s) to the WatchList

The following URI is used to add a user to the WatchList and retrieve their status:

Table detailing Add User to WatchList request.

URI	***Method***
/presence	POST

Table detailing Add User to WatchList request parameters.

Name	***Type***	***Required***	***Default***	***Description***
users	String[]	Optional		The list of userIds to be added to the watchlist, separated by "|"

Table detailing Add User to WatchList response fields.

Field	***Type***	***Content***
action	String Array	["watchlist", "addUser"]
userId	String	The user's ID
displayName	String	The added user's display name

Remove user(s) from the WatchList

The following URI is used to remove a user from the WatchList.

Table detailing Add User to WatchList request.

URI	***Method***
/presence	DELETE

Table detailing Add User to WatchList request parameters.

Name	*Type*	*Required*	*Default*	*Description*
users	String[]	Optional		The list of userIds to be added to the watchlist, separated by "|", or a single username

Table detailing Add User to WatchList response fields.

Field	*Type*	*Content*
action	String Array	["watchlist", "removeUser"]
userId	String	The user's ID

Add Group to WatchList

Table detailing Add Group to WatchList request.

URI	*Method*
/presence	POST

Table detailing Add Group to WatchList request parameters.

Name	*Type*	*Required*	*Default*	*Description*
groupId	String	Required		The ID of the group to be added

Table detailing Add Group to WatchList response fields.

Field	*Type*	*Content*
action	String Array	["watchlist", "addGroup"]
groupId	String	The group's ID

Remove Group from WatchList

Table detailing Remove Group from WatchList request.

URI	*Method*
/presence	DELETE

Table detailing Remove Group from WatchList request parameters.

Name	*Type*	*Required*	*Default*	*Description*
groupId	String	Required		The group's ID

Table detailing Remove Group from WatchList response fields.

Field	*Type*	*Content*
action	String Array	["watchlist", "removeGroup"]
groupId	String	The group's ID

Retrieve User Status

The following URI is used to retrieve the status of an individual user in the WatchList:

Table detailing Retrieve User Status request.

URI	***Method***
/presence/status	GET

Table detailing Retrieve User Status request parameters.

Name	***Type***	***Required***	***Default***	***Description***
users	String[]	Optional		The list of userIds to be added to the watchlist, separated by "|", or a single username

Table detailing Retrieve User Status response fields

Field	***Type***	***Content***
action	String Array	["watchlist", "userStatus"]
userId	String	The user's ID
userStatus	User status	The status object

Table detailing User Status fields

Field	***Type***	***Content***
status	Numeric	The user's status value
statusMessage	String	The user's status message

Delete the Watchlist

The following URI is used to delete the WatchList:

Table detailing the Delete WatchList request.

URI	***Method***
/presence	DELETE

After this call, no notification of status changes are provided on the members of the WatchList.

Suspend Watchlist updates

This can be used to temporarily disable updates to the WatchList.

Table detailing Suspend Watchlist Status request.

URI	***Method***
/presence/status	DELETE

Table detailing Suspend Watchlist Status request parameters.

Request parameters: NONE

This does not cause a response to be generated

Resume Watchlist updates

This can be used to re-enable updates to the WatchList.

Table detailing Resume Watchlist Status request.

URI	***Method***
/presence/status	POST

This causes the WatchList data to be sent as updates , as described above.

Table detailing Resume Watchlist Status request parameters.

Request parameters: NONE

Register for attribute updates

Table detailing Register for Attribute Updates request.

URI	***Method***
/presence/attributes	POST

Table detailing Register for Attribute Updates request parameters.

Name	***Type***	***Required***	***Default***	***Description***
attributes	String Array	Required		The list of attribute IDs to be monitored, separated by "|"

Table detailing Register for Attribute Updates response fields

Field	***Type***	***Content***
action	String Array	["attributes", "remove"]
value	String Array	Array of attribute IDs

Unsubscribe from attribute updates

Table detailing Unsubscribe from Attribute Updates request.

URI	***Method***
/presence/attributes	DELETE

Table detailing Unsubscribe from Attribute Updates request parameters.

Name	***Type***	***Required***	***Default***	***Description***
attributes	String Array	Required		The list of attribute IDs to be monitored, separated by "|"

Table detailing Unsubscribe from Attribute Updates response fields

Field	*Type*	*Content*
action	String Array	["attributes”, "add”]
value	String Array	Array of attribute IDs

Chat Service

These APIs handle the operations and events from one-to-one chat and group chat activities.

Start a 1-to-1 chat

The following URI is used to start a one to one chat:

Table detailing Start One-to-one Chat request.

URI	*Method*
/chat	POST

Parameters:

Table detailing Start One-to-one Chat request parameters.

Name	*Type*	*Required*	*Default*	*Description*
userId	String	Required		The user with whom the chat session is to be started (CN)

The response is:

Table detailing Start One-to-one Chat response fields.

Attribute	*Type*	*Description*
action	String Array	["chat”, "data”]
userId	String	The ID of the chat partner
richtext	Boolean	*TRUE* if this supports rich text, otherwise *FALSE*
displayName	String	The chat partner's display name

Send a 1-to-1 message:

To send a message to a one-to-one chat:

Table detailing Send 1-to-1 Message request.

URI	*Method*
/chat	POST

Parameters:

Table detailing Send 1-to-1 Message request parameters.

Name	*Type*	*Required*	*Default*	*Description*
msg	String	Required		The message

userId	String	Required		The user with whom the chat session was started with

Response:
No Response Data.

Typing Message

To send a typing message to a 1-to-1 chat:

Table detailing Typing 1-to-1 Message request.

URI	***Method***
/chat	PUT

Parameters:

Table detailing Typing 1-to-1 Message request parameters.

Name	***Type***	***Required***	***Default***	***Description***
typing	Boolean	Required		*TRUE*: the user is typing. *FALSE*: the user has stopped typing.
userId	String	Required		The user with whom the chat session was started with

While this API does not generate a response, when the partner is typing the following long poll message is received:

Table detailing Start One-to-one Chat response fields.

Attribute	***Type***	***Description***
action	String Array	["chat", "isTyping"]
userId	String	The ID of the chat partner
displayName	String	The chat partner's display name
isTyping	Boolean	True if the user is typing, FALSE othewise

Upgrade to n-way

To upgrade a one to one chat to an nway chat:

Table detailing Upgrade to N-way request.

URI	***Method***
/chat/nway	POST

Parameters:

Table detailing Upgrade to N-way request parameters.

Name	***Type***	***Required***	***Default***	***Description***
inviteList	String Array	Required		The list of invitees, separated by "|"

topic	String	Optional		The topic of the N-way chat
userId	String	Required		The user with whom the chat session was initially started.

Leave a 1-to-1 chat

To leave a 1-to-1 chat:

Table detailing Leave 1-to-1 Chat request.

URI	***Method***
/chat/nway	DELETE

Parameters:

Table detailing Leave 1-to-1 Chat parameters.

Name	***Type***	***Required***	***Default***	***Description***
placeId	String	Required		The placeId of the N-way chat

Start an n-way chat

To start an n-way chat:

Table detailing Start N-way Chat request.

URI	***Method***
/chat/nway	POST

Parameters:

Table detailing Start N-way Chat request parameters.

Name	***Type***	***Required***	***Default***	***Description***
inviteList	String Array	Required		The list of invitees, separated by "|"
topic	String	Optional		The topic of the meeting

The response is;

Table detailing Start N-way Chat accept response.

Attribute	***Type***	***Description***
action	String Array	["nway", "userEntered"]
localContactId	String	The ID of the local user
LocalAlias	String	The name of the local user
placeid	String	The place associated with this chat
topic	String	The chat topic

Table detailing Start N-way Chat reject response.

Attribute	*Type*	*Description*
action	String Array	["meeting", "getMeetingList"]
userId	String	The ID of the user who declined
placeId	String	The place associated with this chat
topic	String	The chat topic

Send a message to an n-way chat

The following URI is used to send a message to an n-way chat:

Table detailing Send N-way Message request.

URI	*Method*
/chat/nway	POST

Parameters:

Table detailing Send N-way Message request parameters.

Name	*Type*	*Required*	*Default*	*Description*
msg	String	Required		The message
placeId	String	Required		The placeId for this N-way chat session

Response:
No Response Data.

Typing message in an n-way chat

To send a typing message to an n-way chat:

Table detailing Typing N-way Message request.

URI	*Method*
/chat/nway	PUT

Parameters:

Table detailing Typing N-way Message request parameters.

Name	*Type*	*Required*	*Default*	*Description*
placeId	String	Required		The placeId for this N-way chat session
typing	Boolean	Required		TRUE: the user is typing. FALSE: the user has stopped typing.

Response:
No Response Data.

Leave an n-way chat

To leave an n-way chat:

Table detailing Leave N-way Chat request.

URI	***Method***
/chat/nway	DELETE

Parameters:

Table detailing Leave N-way Chat request parameters.

Name	***Type***	***Required***	***Default***	***Description***
placeId	String	Required		The placeId for this N-way chat session

Response:
No Response Data.

Get meetings information

Table detailing Meetings Information request.

URI	***Method***
/chat/meeting	GET

Table detailing Meetings Information request parameters.

Request parameters: NONE

Table detailing Meetings Information response.

Attribute	***Type***	***Description***
action	String Array	["meeting", "getMeetingList"]
list	Object Array	An array of Meeting Room information objects

Table detailing Meeting Room information objects.

Attribute	***Type***	***Description***
originType	String	
fromCalendar	Boolean	*TRUE* if started from calendar, *FALSE* otherwise
name	String	The room name
allowsVideo	Boolean	*TRUE* if the room is enabled for video
hasPassword	Boolean	*TRUE* if the room is secured with a password
id	String	The room ID
permaName	String	The normalized name of the room
activeUserCount	Numeric	The number of participants currently in the room
ownerName	String	The name of the room owner

isUnlisted	Boolean	*TRUE* if the room is unlisted, i.e. not returned in a search
originId	String	
managersList	String Array	Array of the user IDs of the room managers
org	String	
isRestricted	Boolean	*TRUE* if this is a restricted room
conferencingInfo	Object	Information on how to conference into the meeting
creator	String	The user ID of the person who created the room
joinPath	String	The path to join the room
url	String	The URL to join the room
largeMeetings	Boolean	*TRUE* if large meetings are supported
owner	String	The user ID of the room owner
createdDate	Numeric	The timestamp when the room was created
lastAccessed	Numeric	The timestamp when the room was last accessed
isEncrypted	Boolean	*TRUE* if the room is encrypted
description	String	The room description

Start a meeting

To start a meeting:

Table detailing Start Meeting request.

URI	***Method***
/chat/meeting	POST

Table detailing Start Meeting request parameters.

Name	***Type***	***Required***	***Default***	***Description***
inviteList	String Array	Required		The list of invitees
topic	String	Required		Meeting Description
createMeeting	Boolean	Required		*TRUE*: Create Instant Meeting Room. *FALSE*: Invite to an existing meeting room.
url	String	Required *		Only required/available for existing meeting room(s), i.e. when *createMeeting* is TRUE.

Table detailing Start N-way Chat accept response.

Attribute	***Type***	***Description***
action	String Array	["meeting", "createMeeting"]
localContactId	String	The ID of the local user
LocalAlias	String	The name of the local user
placeid	String	The place associated with this chat
topic	String	The chat topic
url	String	The URL of the meeting room

Telephony Service

These APIs handles the operations and events related to send click to call request to the telephony service provider, such as SUT.

Call By Id

To call a person using the userId:

Table detailing Call By Id request.

URI	*Method*
/call	POST

Parameters:

Table detailing Call By Id request parameters.

Name	*Type*	*Required*	*Default*	*Description*
userId	String	Required		The recipients userId
myNumber	String	Optional		The initiators Telephone number. Some telephony providers may require/accept this parameter.

Call By Number

To call a person using their telephone number:

Table detailing Call By Number request.

URI	*Method*
/call	POST

Table detailing Call By Number request parameters.

Name	*Type*	*Required*	*Default*	*Description*
userNumber	String	Required		The recipients telephone number
myNumber	String	Optional		The initiators Telephone number. Some telephony providers may require/accept this parameter.

User Info

Get a user's Info

Table detailing Get UserInfo request.

URI	*Method*

/userinfo	GET

Table detailing Get UserInfo request parameters.

Name	***Type***	***Required***	***Default***	***Description***
userId	String	Required		The recipients telephone number

Table detailing Get UserInfo response.

Attribute	***Type***	***Description***
action	String Array	["user", "info"]
userId	String	The ID of the user
info	Object	A variable set of name-value pairs that have been defined in the Community Server as being part of the business card.

Retrieve Privacy List

Table detailing Get Privacy List request.

URI	***Method***
/user/privacy	GET

Table detailing Get Privacy List request parameters.

Request parameters: NONE

Table detailing Get Privacy List response.

Attribute	***Type***	***Description***
action	String Array	["privacy", "get"]
isExcluding	Boolean	*TRUE* if the list contains those blocked, or *FALSE* if everyone except these is blocked
list	String Array	A list of user Ids

Set Privacy List

Table detailing Set Privacy List request.

URI	***Method***
/user/privacy	POST

Table detailing Set Privacy List request parameters.

Name	***Type***	***Required***	***Default***	***Description***
list	String Array	Required		A list of user IDs
isExcluding	Boolean	Required		*TRUE* if the list contains those blocked, or *FALSE* if everyone except these is blocked

Table detailing Set Privacy List response.

Attribute	*Type*	*Description*
action	String Array	["privacy", "set"]
isExcluding	Boolean	*TRUE* if the list contains those blocked, or *FALSE* if everyone except these is block
list	String Array	A list of user Ids

Preferences

Get system preferences

The current list of supported preferences is:

- BRING_CHAT_WINDOW_TO_FRONT
- NOTIFICATION_PLAY_A_SOUND
- DISPLAY_PHOTO_IN_TABBED_CHAT
- CONTACT_LIST_EXPAND
- NOTIFY_PARTNERS_LEAVE_CHAT
- SAVE_CONTACT_LIST_ON_EXIT
- DISPLAY_OFFLINE_USERS

Table detailing Get Preferences request.

URI	*Method*
/preferences	GET
/preferences/<preference>	GET

To query the value of a specific preference, append the preference name to the URL.

Parameters:

Table detailing Get Preferences request parameters.

Request parameters: NONE

Table detailing Get Preferences response.

Attribute	*Type*	*Description*
action	String Array	["preferences", "preferences"]
preferences	Object	A set of name-value pairs for system preferences. If a single preference is queried, this is a singleton set.

Set system preferences

Table detailing Set Preferences request.

URI	*Method*
/preferences	POST

Parameters:

Table detailing Set Preferences request parameters.

Name	*Type*	*Required*	*Default*	*Description*
preferences	Object	Required		A set of name-value pairs

Table detailing Set Preferences response.

Attribute	*Type*	*Description*
action	String Array	["preferences", "update"]

Core Objects

The following core objects are used as artifacts in the JSON objects returned from the server.

STUser

This object represents the user:

Table listing STUser attributes.

Attribute	Type	Description
Id	String	User's Sametime id
Name	String	User name
Nickname	String	Display name

Person

Extends the *STUser* object with location and status information:

Table listing Person attributes.

Attribute	*Type*	*Description*
Location Status	Location Status	User's location information User's current status (type and message text)

Location

This object represents the user's location.

Table listing Location attributes.

Attribute	*Type*	*Description*

countryName	String	Country name
zipCode	String	Postal/Zip code, if used
State	String	State
phoneNumber	String	Phone number
callMe	Boolean	User wants to be contacted by phone
userDefinedLocation	String	/directory/all

Group

This object represents a private or public group:

Table listing Group attributes.

Attribute	***Type***	***Description***
id	String	Group's (BL) id that uniquely identifies the group
name	String	Name of the group
type	String	A group can be public or private
description	String	Description of this group
members	STUser[]	Group members

IM

This is the base object representing 1-1 chat or n-way chat:

Table listing Chat attributes.

Attribute	***Type***	***Description***
localUser	Person	Person who logged in
Partner	Person[]	Chat partner(s)
Chat/conf id	String	Chat ID, or conf Id, if nway chat
isEncrypted	Boolean	Is the message encrypted?

N-way Chat

The n-way chat object extends the IM Object

Table listing N-way Chat attributes.

Attribute	***Type***	***Description***
Name	String	Conference name invitation message is used

Status

Status represents user's online status.

Table listing Status codes and their meanings.

0x0000	Offline
0x0008	In a meeting
0x0016	Active
0x0040	Not using this computer
0x0060	The user is away from the computer
0x0080	Do not disturb (DND)
0x0200	Mobile User - This bit indicates that the user is connected via a mobile device. Used as the base value for mobile status,e.g., 0x0216 for mobile active, 0x208 for mobile user in a meeting, 0x280 for mobile DND,etc.
0x0800	The user has unknown status. Returned when a user tries to change her status to OFFLINE

Appendix B. Error conditions

If an API call encounters an error, the response is a predictable HTTP error status. While the status codes are described in RFC 2616, "Hypertext Transfer Protocol HTTP/1.1", section 10, are relevant, the following are explicitly returned as a result of errors encountered by the API:

Table listing HTTP error codes, names, and descriptions.

Code	Name	Description
200	OK	The request has succeeded
201	Created	The request has been fulfilled and resulted in a new resource being created.
400	Bad Request	The request could not be understood by the server due to malformed syntax. Please check the logs and fix your input data.
401	Unauthorized	The request requires user authentication. Please make sure you have the rights to access the resource and/or check your login data.
403	Forbidden	The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated.
404	Not Found	The server has not found anything matching the request URI.
405	Method Not Allowed	The method specified in the RequestLine is not allowed for the resource identified by the RequestURI
409	Conflict	The request could not be completed due to a conflict with the current state of the resource.
410	Gone	The requested resource is no longer available at the server and no forwarding address is known.
500	Internal server error	Please check the logs.

Other error codes that can be returned by the server are listed below, with a clarification in parentheses where needed:

Table listing server error codes, names, and descriptions.

Code	Text (description)
1	Sametime is temporarily unavailable. (The community server is offline)
2	Please enter a valid username and password.
3	Service is currently unavailable
4	Operation could not be performed.

5	Operation could not be performed. (The group was not found)
6	Adding subgroups to a public group is not allowed.
7	Operation could not be performed. (The person was not found)
8	Search is currently unavailable.
9	Chat service is currently unavailable.
10	This group already exists in your contact list.
11	Please login to Sametime. (The user is not authenticated)
12	Please enter a valid username and password. (Incorrect credentials)
13	Sametime policy services are currently unavailable, Please try again.
14	Operation could not be performed. (Invalid parameters to an API call)
15	Guest log in is not allowed, please use a valid username and password.
16	The user has been logged out because she has logged into another instance
17	The user has logged in as a new user without logging out the old user..

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
5 Technology Park Drive
Westford Technology Park
Westford, MA 01886

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp.
Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks