Sametime

Version 9.0

Sametime 9.0
Software Development Kit

Java Toolkit Developer's Guide

IBM

Edition Notice

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

This edition applies to version 9.0 of IBM Sametime (program number 5725–M36) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Introduction....5

Intended Audience....5

Requirements....5

How to Use This Guide....5

Related Documents....6

Additional Information....6

Chapter 2. Community Services....7

Introduction....7

Announcement Service....8

Awareness Service....9

Buddy List Service....17

Community Service....19

Directory Service....21

FileTransfer Service....23

Instant Messaging Service....32

Lookup Service....39

Multicast Service....40

Names Service....42

Places Service....43

Post Service....55

Storage Service....56

Token Service....57

XML Buddy List Service....58

Appendix A. Enabling Your Domino Applications....61

Using Tokens....61

Locating the Sametime Server....61

Using the Sample Domino Application....62

Enabling Your Template....62

Appendix B. Component Dependencies for Loading and Packaging....66

Methods of Loading Components....66

Restrictions....67

Available Components....68

Component Dependencies....68

Appendix C. Sametime Identifiers........69

Awareness Service........69

Instant Messaging Service........70

Storage Service........70

User In Place Attributes........72

Place Activity Types........72

Post Service........72

Appendix D. Representing External Users........73

Overview........73

Appendix E. Sametime Connectivity........74

Overview........74

The HybridPollingConnection class........74

The connectivity agent framework........75

Appendix F. Core Types........76

Overview........76

Sametime ID Types........76

Sametime Object Types........76

Appendix G. Multiple Language Support........78

Appendix H. Signing Applets........79

Appendix I. Deprecated AWT and Community UI Components........80

Introduction........80

Add Dialog........81

Awareness List........86

Directory Panel........90

Place Awareness List........95

Privacy Panel........100

Resolve Panel........105

Announcement UI Component........108

Chat UI Component........110

Community UI Component........124

FileTransfer UI Component........125

Appendix J. Deprecated Meeting Services Java APIs........134

Chapter 1. Introduction

Intended Audience

This guide is intended for Java™ developers who have read the *IBM® Lotus® Sametime® Java Toolkit Tutorial* and need a more in-depth understanding of the Sametime toolkit in order to create their own Sametime-enabled Java applications and applets. The guide is written to provide quick and easy reference to the different components of the toolkit and related subjects.

This guide does not include information about general Java programming. For more information on the Java language and Java programming, please go to http://www.java.sun.com. This guide does not cover some of the general topics related to Sametime programming that are covered in the *Tutorial*.

Requirements

For information about software requirements for the Sametime Java Toolkit, see releaseNotes.txt, included with the toolkit.

How to Use This Guide

Chapters and appendixes

This manual is an in-depth reference to the Sametime Java Toolkit. Besides this introductory chapter, it contains a primary chapter on the Community Services components, and a handful of appendices which cover additional subjects and issues.

The Community Services chapter is divided into sections that cover each of the toolkit services or components. The possible headings within each section are:

- **Overview** – A high-level description of the service or component and its uses
- **Diagram** – A Unified Modeling Language (UML) diagram that provides a high-level graphic description of the service or component
- **Description** – An in-depth description of how to use the service or component
- **Platform Dependencies** – A description of any platform limitations of the service or component
- **Package** – The Java package name of the service or component in the toolkit hierarchy
- **See Also** – Additional samples and/or sections that are related to this service or component
- **Example** – A code sample that shows how this service or component is used

Ten appendixes cover additional subjects and issues:

- **Appendix A** – Enabling Your Domino™ Applications
- **Appendix B** – Component Dependencies for Loading and Packaging
- **Appendix C** – Sametime Identifiers
- **Appendix D** – Representing External Users
- **Appendix E** – Sametime Connectivity

- **Appendix F** – Core Types
- **Appendix G** – Multiple Language Support
- **Appendix H** – Signing Applets
- **Appendix I** – Deprecated AWT and Community UI Components
- **Appendix J** – Deprecated Meeting Services Java APIs

Samples

Many of the examples found in this guide can be run directly from the toolkit Samples page. For instructions on accessing and running the samples, refer to Chapter 3 of the *Sametime Java Toolkit Tutorial*.

Guide Conventions

The following conventions are used in this guide:

- Sample code is in `Courier New`
- Sample code that has been added to a previous sample step is in **`bold Courier New`**.
- New terms and emphasis are in *italics*.

Related Documents

- *Sametime Java Toolkit Javadoc Reference*
- *Sametime Java Toolkit Tutorial*

Additional Information

Additional information can be found at the following Web sites:

http://www.lotus.com/sametime

`http://www.ibm.com/developerworks/lotus`

Chapter 2. Community Services

Introduction

The following Sametime Community Services are provided by the toolkit:

- Announcement Service
- Buddy List Service
- Community Service
- Directory Service
- FileTransfer Service
- Instant Messaging Service
- Lookup Service
- Multicast Service
- Names Service
- Places Service
- Post Service
- Storage Service
- Token Service

Announcement Service

Overview

An announcement is a one-way message that can be sent to one or more people. The Announcement Service provides the ability to send and receive announcements between Sametime users.

Description

Every Sametime user can send announcement messages to other users in the community. You can send an announcement to a list of STUser and STGroup (public group) instances. The message will be sent only to users who are online.

To use the Announcement Service, you must load the BLService component into the Sametime session object, like any other toolkit component.

To send an announcement, use the sendAnnouncement() method.

To receive announcements, you must implement the AnnouncementListener interface, which contains the single method announcementReceived(). Use the addAnnouncementListener() method to add the listener to the list of listeners for your Announcement Service instance.

Package

com.lotus.sametime.announcement

See Also

The Announcement UI section of this guide

Awareness Service

Overview

The Awareness Service provides the ability to know the online status of different kinds of Sametime objects such as users, groups, and servers. Using this service, you can register to receive notifications of changes in the online status of users and the online attributes of users and Sametime servers. This service is sometimes referred to as "People Awareness."

Description

The Awareness Service provides the ability to receive notifications on changes in the status of Sametime objects. The table below summarizes the supported objects, all derived from STObject, and the type of notifications you can receive on each object.

Table listing supported objects and notification types.

Object Name	Status Notification	Attribute Notification
STUser	✓	✓
STGroup	✓	✓
STServer		✓

Notifications for STGroup objects are not actually about the group. They are about the online users in the group, as defined by the administrator in the directory. You will not receive any notifications for users in the group who are offline, except to notify you that an online user is now offline.

Sametime server attributes are used to broadcast different capabilities of a Sametime server to all clients connected to that server. For example, a server attribute is used to notify clients whether directory browsing is available on this server.

User Status

Status events are generated whenever a Sametime user changes his online status and are returned as STWatchedUser objects derived from STUser. A status event could be a change from offline to online, from online to offline, or a status change while being online.

An online user in Sametime has four standard statuses: Active, Away, Automatic Away, and Do Not Disturb. The Automatic Away is displayed in the UI with the same icon as Away, but indicates that the application assumes that you are not available. For example, Sametime Connect changes your status to Automatic Away when you have not touched your mouse or keyboard for a predetermined amount of time.

Sametime Attributes

Attribute events are generated whenever a Sametime user or server changes or removes one of its online attributes. A Sametime attribute (STAttribute) has a *key* and a *value*, where the value can be a Boolean, integer, long, string, or byte array. The last login to change the attribute value overrides any previous value of the attribute. When an attribute event is generated, the attribute is returned as an STExtendedAttribute object. STExtendedAttribute is derived from STAttribute and includes additional information on the attribute.

The two special types of Sametime attributes are Existential Attributes and Heavy Attributes.

Existential Attributes

An existential attribute is an attribute that has no value. Its existence or non-existence is by itself an indication of some state. For example, Sametime uses an existential attribute to indicate whether or not a user has a video camera installed. An existential attribute stays set until the last login that set it logs out. For example, if a user Joe sets an extended attribute, anyone watching Joe will receive an attributeChanged() event. Then a second login of Joe sets the same existential attribute. No one will receive any event because this attribute was set by the first login. Now the first login of Joe logs out. (He closes the client.) Again, no one will receive any attributeChanged() event because the second login is still "holding" this attribute. Only when the second login removes the attribute or logs out will all the other users of the community receive the attrRemoved() event. Existential attributes are used to publish properties shared by all logins of a given user. Use the isExistential() method of STExtendedAttribute to find out whether or not an attribute is existential.

Heavy Attributes

Heavy attributes are attributes that have a "heavy" value. The Awareness Service defines some byte limit over which a value is considered heavy. All users who are aware of a specific user receive notifications on his attributes changes. If this attribute is large, the notification will create a lot of network traffic and load the Sametime server.

Instead, if the attribute is considered heavy, the Awareness Service notifies all the interested users that a specific user has changed his attribute, but it does *not* provide the value part of the attribute. Use the getActualSize() method of STExtendedAttribute to determine if a returned attribute is heavy. The attribute is heavy if it is not existential and the returned actual size is 0.

To retrieve the value of a heavy attribute, call the queryAttrContent() method of the WatchList object that contains the watched Sametime object. If you want to know the expected size of the heavy attribute before querying for it, use the getSize() method of STExtendedAttribute.

Attribute Filter

Most applications are only interested in a subset of the possible attributes. The Awareness Service requires that you define a list of attribute keys about which you want to receive notifications. This list is a single global list for all watch lists. It is called the *attribute filter*.

Using the Awareness Service

The Awareness Service provides several functions:

- Tracking the availability of the Awareness Service
- Creating WatchList objects
- Finding a specific user's status
- Setting the attribute filter
- Changing your own online attributes

Tracking the Availability of the Awareness Service

Tracking Awareness Service availability is important so that you know when you are receiving awareness updates in real-time and when you are not receiving any notifications due to a network or server problem. At any point in time, you can call the isServiceAvailable() method of the Awareness Service to find out current service availability.

A better way to track the service is to use the AwarenessServiceListener, which provides two events: serviceAvailable() and serviceUnavailable(). By registering to this listener you will receive events whenever the service availability changes. If for any reason the service becomes unavailable, the Awareness Service component will immediately attempt to restore connection to the service and will notify you when the connection has been restored.

Creating WatchList Objects

Awareness of users and other Sametime objects is achieved using a WatchList object. You create a new WatchList by calling the Awareness Service method createWatchList(). See the Using Watch Lists section below for information on how to use WatchList objects.

Finding a Specific User's Status

If you want to determine the status of a specific user and you know he is already in one of your WatchList objects, you can call the Awareness Service method findUserStatus(). This method returns the status of the user if found after searching all the WatchList objects.

Setting the Attribute Filter

Use the setAttrFilter(), addToAttrFilter(), and removeFromAttrFilter() methods of the Awareness Service to modify the global attribute filter. Modify the attribute filter before adding any items to your watch lists; by default, the filter is empty and no attribute notifications are received.

Changing Your Own Online Attributes

You can change your own login attributes by using the changeMyAttr() and removeMyAttr() methods of the Awareness Service. Use a MyAttributeListener object to receive events on whether these attribute changes and removals were successful.

Using Watch Lists

As previously mentioned, awareness on Sametime objects is achieved with the WatchList class. A WatchList is a list of watched Sametime objects. Once you have created a WatchList object by using the createWatchList() method of the Awareness Service you can:

- Add Listeners to receive status and attribute change events.
- Add/Remove Sametime objects to/from the WatchList.
- Query for the content of heavy attributes.

You can add three types of Sametime objects to your watch lists: STUser, STGroup, and STServer objects. The table below describes the methods related to these watch list objects.

Table listing watch list object methods and descriptions.

Method	Description
addItem()	Adds a single object to a watch list.
addItems()	Adds multiple objects to a watch list.
removeItem()	Removes a single object from the watch list and stops receiving events about the object.
removeItems()	Removes multiple objects from the watch list and stops receiving

	events about the objects.
reset()	Removes all current Sametime objects from the watch list.
close()	Allows the Awareness Service to “clean up” after the object is removed. The object cannot be used after a call to this method is made.

Receiving Events From a WatchList

The two types of events that can be received on Sametime objects in a WatchList are status events and attribute events.

Receiving Status Events

To receive status events, implement the StatusListener interface and add it to the WatchList with the addStatusListener() method. StatusListener has two event handlers:

- **userStatusChanged()** – This event handler is called when one or more users have changed their online status. Use the getWatchedUsers() method of the event object to retrieve the array of STWatchedUser objects whose status has changed.

- **groupCleared()** – This event handler is called when the Awareness Service requests to clear the list of users in a Sametime group. Use the getGroup() method of the event object to retrieve the STGroup object whose users should be cleared. Any UI displaying the online users of a Sametime group should remove these users accordingly. Immediately after this event, the Awareness Service will resend a userStatusChanged() event for each of the online users of the group. This event is relatively rare; but it can happen if, for example, an administrator changes the contents of a Sametime group in the directory.

Receiving Attribute Events

To receive attribute events, implement the AttributeListener interface and add it to the WatchList with the addAttributeListener() method. AttributeListener has four event handlers. Use the getWatchedObject() method of the event object to find out which item in the list each event refers to for all four event handlers. The event handlers are:

- **attrChanged()** – Called when the attributes of an item in the WatchList have changed. Use the getAttributeList() method of the event object to retrieve the array of changed attributes.

- **attrRemoved()** – Called when an attribute of a Sametime object in the WatchList has been removed by its owner. For example, if a user we are currently watching has first set some attribute concerning himself and then decided to remove this attribute, you will first receive the attrChanged() event and then the attrRemoved() event concerning this user.

- **attrContentQueried()** – Called if the content of a heavy attribute has been successfully retrieved. To retrieve the contents of a heavy attribute you must call the queryAttrContent() method of the WatchList. Use the getAttribute() method of the event object to retrieve the returned heavy attribute.

- **queryAttrContentFailed()** – Called if the queryAttrContent() request has failed for any reason. Use the getAttributeKey() method to find out the attribute key this failed event refers to and use the getReason() method to retrieve the reason code for the failure.

Package

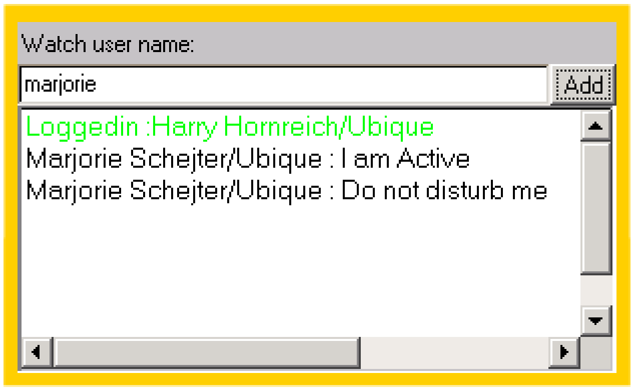
`com.lotus.sametime.awareness`

See Also

The Awareness List section of this guide

Example

This example demonstrates use of the Awareness Service by showing how you can add users to a list of "watched" users. Whenever the online status of a user in the list changes, the change is displayed by the example applet.



Loggedin :Harry Hornreich/Ubique
Marjorie Schejter/Ubique : I am Active
Marjorie Schejter/Ubique : Do not disturb me

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.util.*;
import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.constants.*;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.awareness.*;
import com.lotus.sametime.commui.*;
import com.lotus.sametime.guiutils.chat.ChatArea;

public class AwarenessApplet extends Applet
  implements LoginListener,CommUIListener, AwarenessServiceListener,
             StatusListener, ActionListener
{
  private STSession m_session;
  private CommunityService m_comm;
  private CommUI m_commUiService;
  private AwarenessService m_awareness;
  private WatchList m_watchList;

  private String m_myName ="";
  private TextField m_watchName = new TextField();
  private Button m_addBtn = new Button("Add");
  private Font m_font = new Font("Dialog",12,Font.PLAIN);
  private ChatArea m_transcript = new ChatArea(150,m_font,50);

  public void init()
  {
    try
    {
      m_session = new STSession("ImApplet" + this);
      m_session.loadAllComponents();
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }

    m_session.start();

    initLayout();
    login();
  }

  /**
   * set the applet layout
   */
  private void initLayout()
  {
    setLayout(new BorderLayout(0,0));
    Font labelFont = new Font("Dialog",14,Font.PLAIN);

    Panel pnl = new Panel(new BorderLayout(0,0));
    pnl.add(new Label("Watch user name:"),BorderLayout.NORTH);
    pnl.add(m_watchName,BorderLayout.CENTER);
    pnl.add(m_addBtn,BorderLayout.EAST);
    m_addBtn.addActionListener(this);

    add(pnl,BorderLayout.NORTH );
    add(m_transcript,BorderLayout.CENTER);
    validate();
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    m_comm = (CommunityService)
    m_session.getCompApi(CommunityService.COMP_NAME);

    m_comm.addLoginListener(this);

    m_comm.loginByPassword(getCodeBase().getHost().toString(),
    getParameter("loginName"),
    getParameter("password"));
  }

  /**
   * Loggedin event receive only if loggedin successfully
   */
  public void loggedIn(LoginEvent event)
  {
    m_myName =
        m_comm.getLogin().getMyUserInstance().getDisplayName();
    m_transcript.writeSeparator("Loggedin :"+m_myName,
                                Color.green);
    m_awareness = (AwarenessService)
        m_session.getCompApi(AwarenessService.COMP_NAME);
    m_awareness.addAwarenessServiceListener(this);

    m_commUiService = (CommUI )
        m_session.getCompApi(CommUI.COMP_NAME);
    m_commUiService.addCommUIListener(this);
  }
```

```
/**
 * Loggedin event receive only if loggedout
 */
public void loggedOut(LoginEvent event)
{
  m_transcript.writeSeparator("Loggedout :"+m_myName, Color.green);
}

/**
 * if the send button was pressed take the name form the textField
 * and resolve for appropriate user
 */
public void actionPerformed(ActionEvent event)
{
  m_commUiService.resolve(m_watchName.getText());
}

/**
 * resolved event receive aftere you send request to resolve a name
 */
public void resolved(CommUIEvent event)
{
  m_watchList.addItem(event.getUser());
}

/**
 * resolveFailed event receive aftere you send request to resolve
 * a name but no approrriate user was found.   */
public void resolveFailed(CommUIEvent event)
{
  m_transcript.writeSeparator("resolve failed type another name" ,
                              Color.red);
}

/**
 * Indicates that the awareness service is currently available.
 */
public void serviceAvailable(AwarenessServiceEvent event)
{
  m_watchList = m_awareness.createWatchList();
  m_watchList.addStatusListener(this);
}

/**
 * Indicates that the awareness service is currently unavailable.
 */
public void serviceUnavailable(AwarenessServiceEvent event)
{
  m_transcript.writeSeparator("Awareness service unavailable" ,
                              Color.red);
}

/**
 * Indicates one or more users status' have changed.
 */
public void userStatusChanged(StatusEvent event)
{
  STWatchedUser[] users =(STWatchedUser[])event.getWatchedUsers();
  for(int i=0; i<users.length; i++)
  {
    STUserStatus userStatus = users[i].getStatus();
    String statusDescription =
       users[i].getStatus().getStatusDescription();
    if (users[i].getStatus().getStatusType()== 0)
    {
      statusDescription= "user is offline";
```

```
        }
        String message = users[i].getDisplayName()+" : " +
                         statusDescription;
        m_transcript.writeSeparator(message , Color.black);
      }
    }

    /**
     * Indicates the awareness service has cleared the list of users
     * in a Sametime group. Any UI displaying the online users of a
     * Sametime group should remove these users accordingly.
     */
    public void groupCleared(StatusEvent event)
    {
      m_transcript.writeSeparator("group cleared" , Color.red);
    }

    public void destroy()
    {
      m_comm.logout();
      m_session.stop();
      m_session.unloadSession();
    }
  }
```

Buddy List Service

Overview

The Buddy List Service provides the ability to get a user's contact list from the user storage on the Sametime server, or to store it there. It also provides conversion methods between String and BL objects. These capabilities enable the developer to avoid dealing with the low-level protocols defined by the Storage Service.

Note: In Sametime 7.5, a new XML based Buddy List format was introduced. The Sametime 7.5+ client handles synchronizing this new format with the old format to allow for backwards compatibility with older clients and servers. To access the XML buddy list directly, see the Xml Buddy List Service later in this chapter.

Description

Before beginning to use the Buddy List Service, you must:

- Load the BLService component into the Sametime session object, like any other toolkit component.
- Implement BLServiceListener, and add it to your Buddy List Service instance by calling the method:

```
        addBLServiceListener();
```

BLServiceListener provides two events to let you know if the BLService is currently available:

- void serviceAvailable(BLEvent evt);
  This indicates that the Buddy List Service is currently available.
- void serviceUnavailable(BLEvent evt);
  If the service has failed, you receive the serviceUnavailable() event. The service component automatically reconnects to the service when it is available again. You will then receive the serviceAvailable() event.

The BL Object

The BL object contains groups that are encapsulated by the PrivateGroup and PublicGroup classes implementing the BLGroup interface.

To add, remove, and retrieve groups and users, see the *Sametime Java Toolkit Referenc*e.

Get the Contact List from the User Storage

The developer can get a contact list using this method, which is defined in the BLServiceListener:

```
void getBuddyList();
```

Note that this method does not return any value. The asynchronized result is retrieved by the event handler.

```
void blRetrieveSucceeded(BLEvent evt);
```

The BL object can be extracted from the BLEvent:

```
BL m_buddyList = evt.getBL();
```

In case of failure, the user gets the BLEvent through the event handler:

```
        void blRetrieveFailed(BLEvent evt);
```

Set the Contact List to the User Storage

The following method (defined in the BLServiceListener) is used to write a new Contact list to user storage:

```
        void setBuddyList(BL newBuddyList)
```

If the set action succeeds, the user should get the BLEvent through the event handler, which is defined in the BLServiceListener:

```
        void blSetSucceeded(BLEvent evt);
```

In case of failure, the user gets the BLEvent through the event handler:

```
        void blSetFailed(BLEvent evt);
```

BuddyList Updated

The user should get the BLEvent through the following method when the contact list has been updated by some other login. This method is defined in the BLServiceListener:

```
        void blUpdated(BLEvent evt);
```

Converting between String and BL Objects

The Buddy List Service provides parsing methods that convert String to BL object and vice versa. These methods are used, for example, when reading the BL object from a text file or writing it to a text file:

```
BL stringToBuddyList(buddyListString)
String buddyListToString(BL buddyList)
```

Package

com.lotus.sametime.buddylist

See Also

The Storage Service section of this chapter.

Community Service

Overview

The Community Service is the core service you must use in order to log in and log out from the Sametime server. In addition to login services, the Community Service provides the ability to receive administrator messages, to change your status and privacy in the community, and to request to change your user name if you logged on anonymously.

Description

The Community Service provides three methods to log in to the community:

- loginByPassword() – Enables login using a login name and password.
- loginByToken() – Enables login using a login name and a temporary login token generated by the Token Service or by some other means. See the Token Service section in this chapter for more information on how to create a temporary login token.
- loginAsAnon() – Enables login as an anonymous user to the community. Anonymous login to a Sametime Community should be enabled by the administrator for this type of login to succeed.

Once logged in, you can log out at any time by calling the logout() method.

By default, the toolkit first attempts to log in through a plain socket connection via port 1533. If this attempt fails, the toolkit attempts to connect via an HTTP tunneling connection on port 8082. This default behavior can be changed by calling the setConnectivity() method with an array of connections to try before calling one of the login methods. The toolkit provides Connection classes for direct connection, HTTP, socks4, socks5, and HTTPS.

Receiving Login Events

To receive login or logout events, you must add a LoginListener object to the Community Service. The loggedIn() event is called when the user has logged in to the community by password or token. The loggedOut() event is called if a login request failed or if the user was disconnected from the Sametime server. Use the getReason() method on the event object to retrieve the precise reason for the logout.

Receiving Admin Messages

The Sametime administrator can send administration messages to all the users in the community. To receive such messages, add an AdminMsgListener to the Community Service. When the administrator sends a message, the adminMsgReceived() event is called with the message text in the event object.

Sensing Availability of Services

The Community Service can be used to determine the status of a Sametime server-side service. If a server-side component fails, you can be notified when it becomes available again. Add a ServiceListener object, and then call the senseService() method with the *service type* for which you want to determine the status. When a service of the requested type is available again, you will receive the serviceAvailable() event of the listener.

The Login Object

Once logged in, you can use the getLogin() method to get a Login object. Using this object, you can query and modify your current status and privacy settings. You can also use this object to change your user name if you logged in anonymously. The Login object is valid only as long as you are logged in and is not be stored for later reference.

Querying and Changing Your Online Status

Use the getMyStatus() method of the Login object to retrieve your current online status. To change the status, call the changeMyStatus() method. To track changes in your online status, add a MyStatusListener object to the Login object. You will receive the myStatusChanged() event for each change in the user status. If you are logged in more than once, you will receive events on status changes made by your other logins.

Note: You can also change the default status that will be used on subsequent logins. For example, you may want to specify a default status of "Mobile Active" for mobile users when they log in. To specify the default status, call the setLoginStatus() method of the CommunityService object.

Querying and Changing Your Privacy

Use the getMyPrivacy() method of the Login object to retrieve your current privacy settings. To change your privacy settings, call the changeMyPrivacy() method. To track changes in your privacy settings, add a MyPrivacyListener object to the Login object. You will receive the myPrivacyChanged() event for each change in the user privacy settings. If you are logged in more than once, you will receive events on privacy changes made by your other logins. If you try to change your privacy settings and the request cannot be performed by the Sametime server, you will receive the changeMyPrivacyDenied() event.

Changing an Anonymous Login User Name

Use the changeMyUserName() method of the Login object to request to change your anonymous login user name. Add a MyNameListener object to track the request results. If the request is approved, you will receive a myUserNameChanged() event. If the request is denied, you will receive the changeMyUserNameDenied() event. Note that changing your user name (the name other users see), is only possible if you logged in anonymously.

Other Login Object Information

The Login object contains useful information about the current login. Use the getServer() method to retrieve the Sametime server you are logged in to. To retrieve your user name, user ID, or login ID, use the getMyUserInstance() method. Use the getServerVersion() method to obtain the version of the server you are currently logged in to.

Package

```
com.lotus.sametime.community
```

See Also

The Community UI section of this guide

Example

See the Live Names sample in the Awareness List section in Appendix I for code that shows how to change your online status.

Directory Service

Overview

The Directory Service enables you to browse the Sametime community directories. It provides a list of all available directories and allows you to query for chunks of entries in each directory.

Depending on the directory configuration, directory browsing might not be available on your Sametime server. For example, if your Sametime server is configured to use an LDAP directory, directory browsing is not available. See the Checking for Directory Browsing Support section below for more information. When directory browsing is not available, the developer should limit the application to the use of the Lookup Service which provides basic resolve capabilities.

Description

Before using the Directory Service you must:

1. Implement DirectoryServiceListener and add it to the service by calling the addDirectoryServiceListener() method.

2. Call the queryAllDirectories() method of Directory Service to retrieve the list of all available directories on the Sametime server you are logged in to.

DirectoryServiceListener provides two types of events. The events serviceAvailable() and serviceUnavailable() let you know if the Directory Service is currently available on the Sametime server. If you perform an operation on the Directory Service and the Directory Service is unavailable, the serviceUnavailable() event is generated. The Directory Service component automatically attempts to restore a connection to the Directory Service on the server and notifies you with the serviceAvailable() event when successful.

The other events you can receive from the DirectoryServiceListener are responses to the queryAllDirectories() method. If the query fails, you receive the allDirectoriesQueryFailed() event. Use the getReason() method of the event object to get the reason code for the failure. If the query is successful, you receive the allDirectoriesQueried() event. Use the getDirectories() method of the Event object to get an array of Directory objects representing all the directories on the Sametime server you are logged in to.

Using Directory Objects

Once you have an array of Directory objects representing the directories on the Sametime server, you can use the getTitle() method of Directory to display a list of the directories. You can browse one directory or several directories at once if needed.

To start browsing the directories:

1. Implement DirectoryListener and add it to your Directory object by calling the addDirectoryListener() method.

2. Change the default chunk size of the directory if needed by calling the getMaxEntries() and setMaxEntries() methods.

3. Open the directory for browsing by calling the open() method.

4. Once you call the open() method, you receive one of two DirectoryListener events:

    - **directoryOpened()** – The directory has been opened successfully.
      Use the getDirectory() method of the event object to find out to which directory this event refers. Use the getDirectorySize() method of the event object to find the total number of entries in the directory.

  - **directoryOpenFailed()** – The directory could not be opened for browsing. Use the getDirectory() method of the event object to find out to which directory this event refers. Use the getReason() method of the Event object to find the reason for the failure.

5. After you receive the directoryOpened() event, query the directory for chunks of users by calling one of the queryEntries() methods. See the "Querying the Directory" that follows for more details.

6. Close the directory when finished browsing by calling the close() method.

Querying the Directory

Once you have opened the directory, you can query for chunks of entries by moving backward or forward in the chunks or by searching for a chunk starting with a certain string prefix. There are two queryEntries() methods for each of these possibilities, and each returns a request ID you should store to compare with the query response events. For each query you receive one of two events in your DirectoryListener:

- **entriesQueried()** – Entries from the directory have been queried successfully. Use the getRequestId() method of the event object to match the event with the appropriate request. Use the getEntries() method of the event object to get an array of STObjects. Each STObject could be a STUser or STGroup object. Use the isAtStart() and isAtEnd() methods of the event object to find out if the directory is at its first or last chunk.

- **entriesQueryFailed()** – A query entries request has failed. Use the getRequestId() method of the event object to match the event with the appropriate request. Use the getReason() method of the event object to find the failure code.

Checking for Directory Browsing Support

As mentioned previously, not every Sametime server supports directory browsing.
Follow these steps to find out if your directory supports directory browsing:

1. Add the following attribute key to your Awareness Service attribute filter:
   com.lotus.sametime.awareness.AwarenessConstants.BROWSE_ENABLED

2. Create a Watch List on your Awareness Service and add an AttributeListener to it.

3. Use the Community Service to find your own server's STServer object by calling getLogin().getServer().

4. Add the returned STServer object to your WatchList object.

5. If you receive an attrChanged() event for the server object with the above attribute key, then you know that the Sametime server supports directory browsing. If you do not receive this attribute from the server, then the server does not support directory browsing.

Package

`com.lotus.sametime.directory`

See Also

The Directory Panel section of this guide

FileTransfer Service

Overview

The FileTransfer Service provides the ability to send and receive files between Sametime users.

The initiator creates a FileTransfer object and starts the transfer. The other side can accept or decline the transfer. After the file transfer starts, both sides receive progress notifications, and can stop the transfer before it completes successfully.

Note that if you are creating an applet that supports file transfer, you must sign the applet, to allow local file access.

Description

The FileTransfer class represents a file transfer session between two clients. It is used for both sending and receiving files.

A FileTransfer object can be created in either of two ways:

- **Explicitly by the sender** – The sender uses the createFileTransfer() method of the FileTransfer Service.

- **Implicitly on the receiver's side** – When someone initiates a file tranfer to you, the toolkit creates a FileTransfer object. This FileTransfer object is passed to the FileTransferInitiated() event handler in the event object. In order to receive this event, the receiver must implement the FileTransferServiceListener and add it to the FileTransferService instance using the addFileTransferServiceListener() method.

Checking and Setting Attributes

In order to send files, your application must connect to a Sametime server that supports file transfer. After the connection is established, and before any file transfer is attempted, your application should check that existential server attribute 9009, AwarenessConstants.FILE_TRANSFER_MAX_FILE_SIZE, is set. For information about checking server attributes, please refer to "Awareness Service" in Chapter 4.

To receive files, your application must set existential user attribute 6, AwarenessConstants.FILE_TRANSFER_SUPPORTED. This attribute is set automatically (provided the server attribute is set) if you use the FileTransferUI component, but must be set manually otherwise. To do so, call AwarenessService.changeMyAttr().

Sending Files

To send a file, you first create a FileTransfer object, using the createFileTransfer() method of the FileTransfer Service.

This method returns a FileTransfer object. To use this method, you must provide the following parameters:

- **remoteUser** – The STUser object of the user you are sending the file to.
- **fileIn** – The FileInputStream that you want to send.
- **fileName** – A string with the full pathname of the file.

- **fileDesc** – A string with the description of the file. The receiver will get the name and the description in the FileTransfer object.

To use the FileTransfer object, you must first call the addFileTransferListener() method to register the FileTransfer listener object. Then you can call the start() method to start the transfer. To cancel a transfer, call FileTransfer.stop().

The results of the file transfer operation are returned to the object implementing the FileTransferListener interface.

The possible events are:

- **fileTransferStarted()** – The transfer has started. This notification is received after the other side accepts the file transfer.
- **bytesTransferredUpdate()** – This notification is received several times during the transfer to indicate progress. You can get the exact number of bytes sent from the FileTransferEvent object.
  By default, you will get this notification every time an additional 5% of the data is sent. You can change this 5% value by calling the setUpdateInterval() method of the FileTransfer object before starting the transfer.
- **fileTransferCompleted()** – The transfer has completed successfully.
- **fileTransferDeclined()** – The other side has declined the file transfer.
- **fileTransferStopped()** – The other side has stopped the file transfer (after it was started).

Receiving Files

To receive a file transfer, you need to implement FileTransferServiceListener and add it to your FileTransferService instance using the addFileTransferServiceListener() method of the FileTransfer Service. When someone initiates a file transfer to you, the toolkit creates a FileTransfer object. This FileTransfer is passed to the FileTransferInitiated() method.

From the FileTransfer object, you can get the file transfer details, such as the sending remote user, the file name and description, and the file size.

If you want to decline the transfer, use the decline() method of the FileTransfer object.

If you want to accept the transfer, call the addFileTRansferListener()method to register the FileTransferListener object. Then you can call the accept() method to start the file transfer.

The results of the file transfer operation are returned to the object that implemented the FileTransferListener interface.

The possible events are:

- **bytesTransferredUpdate()** – This notification is received several times during the transfer to indicate progress. You can get the exact number of bytes received from the FileTransferEvent object. By default, you will get this notification every time another 5% of the data is received. You can change this 5% value by calling the method setUpdateInterval() of the FileTransfer object before calling the accept() method.
- **fileTransferCompleted()** – The transfer has completed successfully.
- **fileTransferStopped()** – The other side has stopped the file transfer after it was started.

If the sender cancels a file transfer, any partially received file is automatically deleted by the FileTransfer object. Note that you cannot reuse the FileTransfer object to receive the same or another file. Instead, create a new object.

Package

com.lotus.sametime.filetransfer

See Also

The FileTransfer UI section of this guide

Example

The following example consists of two classes: FileSenderSample and FileReceiverSample.

The sender sends a file to the receiver right after logging in. After the transfer is completed, the sender logs out.

The receiver logs in and then waits for a file transfer. Upon receiving a file transferr, the receiver accepts it. After it is completed, the receiver waits for another transfer.

To run this example, simply copy the code and change the constants defined at the beginning of each class to real values from your community. Be sure to run the receiver before you run the sender; the receiver must be already logged in when the sender tries to send the file.

```
  import java.io.*;
import java.util.Date;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.util.Debug;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.filetransfer.*;

public class FileSenderSample implements LoginListener, FileTransferListener,
ResolveListener
{
      // replace all these constants with real values from your
      //community
      private final static String SERVER_NAME = "server_name";
      private final static String SENDER_NAME = "sender_user_name";
      private final static String PASSWORD = "sender_password";
      private final static String RECEIVER_NAME = "receiver_user_name";
      private final static String FILE_NAME= "c:\\source_dir\\filename";

    private STSession m_session;
    private CommunityService m_comm;
    private FileTransferService m_fileTransSvc;
    private FileTransfer m_fileTransferSession;
    private String m_remoteUserName;
    private String m_fileName;

   /**
     * Constructs a new file sender client.
     */
    public FileSenderSample(String host, String userName,
                            String password,
                            String remoteUserName, String fileName)
    {
        m_remoteUserName = remoteUserName;
        m_fileName = fileName;

        try
        {
                  // start the session
                  m_session = new STSession("FileSenderSample " + this);
                  m_session.loadSemanticComponents();
                  m_session.start();

                  m_comm = (CommunityService)

                  m_session.getCompApi(CommunityService.COMP_NAME);
                  m_comm.addLoginListener(this);
                  m_comm.enableAutomaticReconnect(5, 10000);
                  m_fileTransSvc = (FileTransferService)
                  m_session.getCompApi(FileTransferService.COMP_NAME);
                  m_fileTransSvc.addFileTransferServiceListener(this);
        }
        catch(DuplicateObjectException e)
        {
          e.printStackTrace();
        }

        // login to the community
        m_comm.loginByPassword(host,
                               userName,
                               password);
    }

    /**
     * Main
     */
    public static void main(String[] args)
    {
        FileSenderSample fileSenderSample1 = new FileSenderSample(SERVER_NAME,
                          SENDER_NAME,
                          PASSWORD,
                          RECEIVER_NAME,
                          FILE_NAME);
    }

      //
      // LoginListener
      //

    /**
     * Logged in event.
     */
    public void loggedIn(LoginEvent event)
    {
       System.out.println("Logged In");

       LookupService lookupSvc = (LookupService)
            m_session.getCompApi(LookupService.COMP_NAME);

       Resolver resolver = lookupSvc.createResolver(true, true,
                                                    true, false);
       resolver.addResolveListener(this);
       resolver.resolve(m_remoteUserName);
    }

     /**
    * Logged out event.
    */
    public void loggedOut(LoginEvent event)
    {
        System.out.println("Logged Out");
        if(!event.isReconnecting())
        {
            m_session.stop();
        }
    }

      //
      // ResolveListener
      //

    /**
     * A resolve request has been performed, and a match was found.
     *
     * @param            event The event object.
     * @see              ResolveEvent#getName
     * @see              ResolveEvent#getResolved
     */
    public void resolved(ResolveEvent event)
    {
        try
        {
            //Separate the file name from the full path name.
            int index = Math.max(m_fileName.lastIndexOf("\\"),
                                 m_fileName.lastIndexOf("/"));
            String fileName =
                  m_fileName.substring(++index, m_fileName.length());

            m_fileTransferSession = m_fileTransSvc.createFileTransfer(
                                      (STUser) event.getResolved(),
                                      new FileInputStream(m_fileName),
                                      fileName,
                                      "This is the description...");

            m_fileTransferSession.addFileTransferListener(this);
            m_fileTransferSession.start();
        }
        catch(IOException e)
        {
            m_fileTransferSession.stop();
            e.printStackTrace();
        }

    }

    /**
     * A resolve request has been performed, and multiple matches have
     * been found.
     *
     * @param            event The event object.
     * @see              ResolveEvent#getName
     * @see              ResolveEvent#getResolvedList
     */
    public void resolveConflict(ResolveEvent event)
    {
        Debug.println("FileSender Resolve conflict");
        m_comm.logout();
    }

    /**
     * A resolve request failed.
     *
     * @param            event The event object.
     * @see              ResolveEvent#getReason
     * @see              ResolveEvent#getFailedNames
     */
    public void resolveFailed(ResolveEvent event)
    {
        System.out.println("FileSender Resolve Failed");
        m_comm.logout();
    }

      //
      // FileTransferListener
      //

    /**
     * Notification sent when a file transfer session has been started.
     */
    public void fileTransferStarted(FileTransferEvent event)
    {
        Date date = new Date();
        System.out.println("File Transfer Started: " +
                           event.getFileTransfer().toString() +
                           " Time: " + date.toString());
    }


    /**
     * Notification sent when a file transfer session has been completed
     * successfully.
     */
    public void fileTransferCompleted(FileTransferEvent event)
    {
        Date date = new Date();
        System.out.println("File Transfer Completed: " +
                           event.getFileTransfer().toString() +
                           " Time: " + date.toString());
            m_comm.logout();
    }


    /**
     * Notification when an attempt to send a file to remote client has
     * been declined by the remote client.
     */
    public void fileTransferDeclined(FileTransferEvent event)
    {
        System.out.println("File Transfer Declined: " +
                           event.getFileTransfer().toString() +
                           " Reason: " +
                           Integer.toHexString(event.getReason()));
        m_comm.logout();
    }

    /**
     * Notification sent when a session has been stopped/terminated.
     */
    public void fileTransferStopped(FileTransferEvent event)
    {
        System.out.println("File Transfer Stopped reason: " +
                           Integer.toHexString(event.getReason()) +
                           "Session: " +
                           event.getFileTransfer().toString());
        m_comm.logout();
    }

    /**
     * Notification send periodicly according indicating the nubmer of
     * bytes of transferred up till now.
     */
    public void bytesTransferredUpdate(FileTransferEvent event)
    {
        System.out.println("Bytes transferred: " +
            event.getFileTransfer().getNumOfByteTransferred());
    }
}
```

```
import java.io.*;
import java.util.Date;
import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.community.*;
import com.lotus.sametime.core.util.Debug;
import com.lotus.sametime.filetransfer.*;

public class FileReceiverSample implements LoginListener,  FileTransferListener,
FileTransferServiceListener
{
      // replace all these constants with real values from your
      // community
    private final static String SERVER_NAME   = "server_name";
    private final static String RECEIVER_NAME = "receiver_user_name";
    private final static String PASSWORD      = "receiver_password";
    private final static String DIR_NAME      = "c:\\target_dir\\";

    private STSession m_session;
    private CommunityService m_comm;
    private FileTransferService m_fileTransSvc;
    private FileTransfer m_fileTransferSession;

    public FileReceiverSample(String host, String userName,
                              String password)
     {
        try
        {
          // create the session
          m_session = new STSession("BaseClient " + this);
          m_session. loadSemanticComponents ();
          m_session.start();

          m_comm = (CommunityService)
                    m_session.getCompApi(CommunityService.COMP_NAME);
          m_comm.addLoginListener(this);
          m_comm.enableAutomaticReconnect(5, 10000);
          m_fileTransSvc = (FileTransferService)
            m_session.getCompApi(FileTransferService.COMP_NAME);
        }
        catch(DuplicateObjectException e)
        {
          e.printStackTrace();
        }

        // login to the community
        m_fileTransSvc.addFileTransferServiceListener(this);
        m_comm.loginByPassword(host,
                               userName,
                               password);
    }

    /**
     * Main.
     */
    public static void main(String[] args)
    {
        FileReceiverSample fileReceiverSample1 = new FileReceiverSample(
            SERVER_NAME,RECEIVER_NAME, PASSWORD);
    }

      //
      // LoginListener
      //

    /**
    * Logged in event.
    */
    public void loggedIn(LoginEvent event)
    {
       System.out.println("Logged In");
    }

    /**
    * Logged out event.
    */
    public void loggedOut(LoginEvent event)
    {
        System.out.println("Logged Out");
        if(!event.isReconnecting())
        {
            m_session.stop();
        }
    }

      //
      // FileTransferListener
      //

    /**
     * Notification sent when a file transfer session has been started.
     */
    public void fileTransferStarted(FileTransferEvent event)
    {
        Date date = new Date();
        System.out.println("File Transfer Started: " +
                           event.getFileTransfer().toString() +
                           " Time: " + date.toString());
    }

    /**
     * Notification sent when a file transfer session has been completed
     * successfully.
     */
    public void fileTransferCompleted(FileTransferEvent event)
    {
        Date date = new Date();
        System.out.println("File Transfer Completed: " +
                           event.getFileTransfer().toString() +
                           " Time: " + date.toString());
    }

    /**
     * Notification when an attempt to send a file to remote client has
     * been declined by the remote client.
     */
    public void fileTransferDeclined(FileTransferEvent event)
    {
        System.out.println("File Transfer Declined: " +
                           event.getFileTransfer().toString() +
                           " Reason: " +
                           Integer.toHexString(event.getReason()));
    }

    /**
     * Notification sent when a session has been stopped/terminated.
     */
    public void fileTransferStopped(FileTransferEvent event)
    {
        System.out.println("File Transfer Stopped reason: " +
                           Integer.toHexString(event.getReason()) +
                           "Session: " +
                           event.getFileTransfer().toString());
    }

    /**
     * Notification send periodicly according indicating the nubmer of
     * bytes of transferred up till now.
     */
    public void bytesTransferredUpdate(FileTransferEvent event)
    {
        System.out.println("Bytes transferred: " +
      event.getFileTransfer().getNumOfByteTransferred());
    }

      //
      // FileTransferServiceListener
      //

    /**
     * A file trnasfer session has been intiated by a remote client.
     * @param The File transfer session.
     */
    public void FileTransferInitiated(FileTransferEvent event)
    {
        System.out.println("File Transfer session Initated");
        m_fileTransferSession = event.getFileTransfer();
        m_fileTransferSession.addFileTransferListener(this);

        Date date = new Date();
        System.out.println("File transfer initated, File: " +
                           m_fileTransferSession.getFileName() +
                           " File Desc: " +
                           m_fileTransferSession.getFileDesc() +
                           " File Size: " +
                           m_fileTransferSession.getFileSize() +
                           " Time: " + date.toString());
        try
        {
            FileOutputStream fileOut =
                  new FileOutputStream(
                        DIR_NAME +
                        m_fileTransferSession.getFileName());
                        m_fileTransferSession.setUpdateInterval(10);
                        m_fileTransferSession.accept(fileOut);
        }
        catch(IOException e)
        {
            e.printStackTrace();
            m_fileTransferSession.stop();
        }
    }
}
```

Instant Messaging Service

Overview

The Instant Messaging Service allows the sending of text and/or binary messages between clients. It is used for standard Instant Messages (IM), but it can also be used for passing any other messages, such as game moves and translated messages.

This service uses an IM type parameter to classify message sessions of different types. To allow clients from different vendors to interoperate, reserved message types should not be used for third-party purposes. The IM types 0 – 10000 are reserved for Lotus internal use. If you need to reserve IM types for your own commercial applications, please follow the registration instructions at http://www.lotus.com/sametimedevelopers.

Description

The IM class, representing an Instant Message (IM) session between two clients, is used for both sending and receiving instant messages. IM objects can be created in one of two ways:

- Using the createIm() method of the Instant Messaging Service.
- When someone initiates an instant message to you, the Sametime Toolkit creates an IM object. This IM is passed to the imReceived() event handler in the event object. In order to receive this event, you must:

  1. Implement ImServiceListener and add it to the InstantMessagingService using the addImServiceListener() method.
  2. Use the registerImType() method of InstantMessagingService to register the IM types your application recognizes and is willing to accept.

     This step is very important. Without it, no IMs will be received, because the Instant Messaging Service will automatically reject any incoming IMs that do not have a registered IM type. The initiating user will receive an openImFailed() event with the appropriate error code.

Opening IM Objects

IM objects must be opened in order to be able to use them for the actual passing of information. IMs created using the createIm() method are closed by default. Incoming IM objects created by the imReceived() event are open by default. You can use the isOpen() method at any time to find out whether an IM object is currently open or closed.

Add an IMListener object to your IM object and then call the open() method to open a closed IM. You will receive one of two events from the ImListener:

- **imOpened()** – The IM object has been opened successfully. Use the getIm() method of the event object to find the IM object this event refers to.
- **openImFailed()** – The IM object could not be opened for some reason. Use the getIm() method of the event object to find the IM this event refers to. Use the getReason() method of the event object to get the reason code for the failure.

Sending and Receiving Text or Data with IM Objects

Once you have created an open IM in one of the two ways described above, the same operations can be performed on the IM. Any side can send text or data to the other party at any time as long as the IM session is open. Use the sendText() method to send text to the other party that will receive a textReceived() event . Use the sendData() method to send binary data to the other party that will receive a dataReceived() event.

Closing IM Objects

Any party of the IM session can close the session at any time by calling the close() method of the IM object. The other side will receive an imClosed() event. Once an IM is closed by any party, the IM object needs to be reopened before it can be used again. Note, that an IM object is created with a specific target user in mind. The IM object can be used only for IM sessions with that user during the lifetime of the object.

Package

`com.lotus.sametime.im`

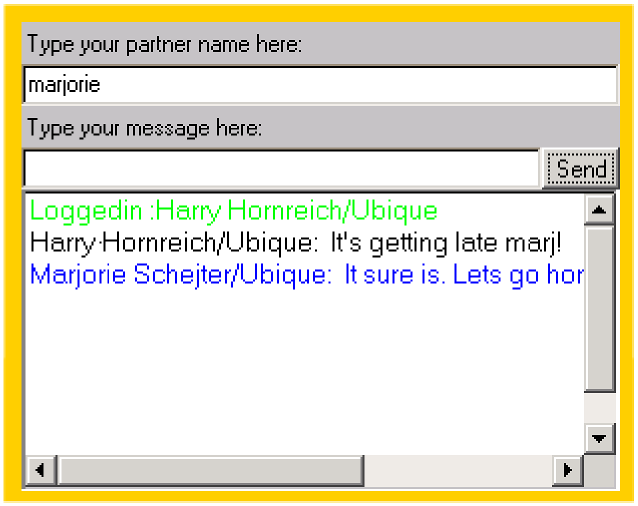
See Also

The Chat UI section of this guide

The Post Service section in this chapter of this guide

Example

This example demonstrates how the Instant Messaging Service is used to send and receive IMs from users. The sample logs you into the Sametime community and allows you to send an IM to any user by typing his name and the message. This example will also display any incoming IM messages sent to you.



```
Loggedin :Harry Hornreich/Ubique
Harry Hornreich/Ubique:  It's getting late marj!
Marjorie Schejter/Ubique:  It sure is. Lets go hor
```

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.constants.*;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.im.*;
import com.lotus.sametime.commui.*;
import com.lotus.sametime.guiutils.chat.ChatArea;
import com.lotus.sametime.guiutils.misc.*;
import com.lotus.sametime.resourceloader.*;

public class ImApplet extends Applet
  implements LoginListener, ImServiceListener, ImListener,
             CommUIListener, ActionListener

{
  private STSession m_session;
  private CommunityService m_comm;
  private InstantMessagingService  m_imService;
  private CommUI m_commUiService;
  private Vector m_ImOpened = new Vector();

  private String m_myName ="";
  private TextField m_imDestination =new TextField();
  private TextField m_imText= new TextField();
  private Button m_sendBtn = new Button("Send");
  private Font m_font = new Font("Dialog",12,Font.PLAIN);
  private ChatArea m_transcript = new ChatArea(150,m_font,50);

  public void init()
  {
    try
    {
      m_session = new STSession("ImApplet" + this);

      //we are not to loading the chatUi comp
      //because we are going to handle the events
      //in different way
      m_session.loadSemanticComponents();
      new CommUIComp(m_session);
      new ResourceLoaderComp(m_session);
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }

    m_session.start();
    initLayout();
    login();
  }

  /**
   * set the applet layout
   */
  private void initLayout()
  {
    setLayout(new BorderLayout(0,0));
    Font labelFont = new Font("Dialog",14,Font.PLAIN);
    Panel northPnl = new Panel(new BorderLayout(10,0));

    Panel details = new Panel(new BorderLayout(0,0));
    details.add(new Label("Type your partner name here:"),
                BorderLayout.NORTH);
    details.add(m_imDestination,BorderLayout.SOUTH);

    Panel messagePnl = new Panel(new BorderLayout(0,0));
    messagePnl.add(new Label("Type your message here:"),
                   BorderLayout.NORTH);
    messagePnl.add(m_imText,BorderLayout.CENTER);
    messagePnl.add(m_sendBtn,BorderLayout.EAST);
    m_sendBtn.addActionListener(this);

    northPnl.add(details,BorderLayout.NORTH);
    northPnl.add(messagePnl,BorderLayout.SOUTH);

    add(northPnl,BorderLayout.NORTH );
    add(m_transcript,BorderLayout.CENTER);
    validate();
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    m_comm = (CommunityService)
        m_session.getCompApi(CommunityService.COMP_NAME);

    m_comm.addLoginListener(this);

    m_comm.loginByPassword(getCodeBase().getHost().toString(),
                           getParameter("loginName"),
                           getParameter("password"));
  }

  /**
   * get an im object and send the text
   */
  public void sendText(Im im)
  {
    im.sendText(false,m_imText.getText());
    m_transcript.writeSeparator(m_myName + ":  " +
                      m_imText.getText(), Color.black);
    m_imText.setText("");
  }

  /**
   * Loggedin event receive only if loggedin successfully
   */
  public void loggedIn(LoginEvent event)
  {
    m_myName = m_comm.getLogin().getMyUserInstance().
                                    getDisplayName();
    m_transcript.writeSeparator("Loggedin :" + m_myName,
                             Color.green);
    m_imService = (InstantMessagingService)
        m_session.getCompApi(InstantMessagingService.COMP_NAME);
    m_imService.registerImType(ImTypes.IM_TYPE_CHAT);

    m_imService.addImServiceListener(this);

    m_commUiService = (CommUI )
    m_session.getCompApi(CommUI.COMP_NAME);
    m_commUiService.addCommUIListener(this);
  }

  /**
   * Loggedin event receive only if loggedout
   */
  public void loggedOut(LoginEvent event)
  {
    m_transcript.writeSeparator("Loggedout :" + m_myName,
                              Color.green);
  }

  /**
   * imReceived event received if someone try to open im to you,
   * check if you have already im with this person, if not add
   * Im listener
   */
  public void imReceived(ImEvent event)
  {
    Im im = event.getIm();
    boolean imExsit = false;
    Im currentIm = null;

    for(int i = 0; i < m_ImOpened.size(); i++)
    {
      currentIm = (Im)m_ImOpened.elementAt(i);
      if(currentIm.equals(im))
      {
      imExsit = true;
      im = currentIm;
      break;
      }
    }

    if(!imExsit)
    {
      m_ImOpened.addElement(im);
      im.addImListener(this);
    }
  }

  /**
   * if the send button was pressed take the name from the
   * textField and resolve for appropriate user
   */
  public void actionPerformed(ActionEvent event)
  {
    m_commUiService.resolve(m_imDestination.getText());
  }
  /**
   * resolved event receive after you send request to resolve
   * a name
   * check if you already have im with this person,
   * if you have sent the text else open new im.
   */
  public void resolved(CommUIEvent event)
  {
    STUser partner = event.getUser();
    Im im = m_imService.createIm(partner,
                        EncLevel.ENC_LEVEL_NONE,
                        ImTypes.IM_TYPE_CHAT);

    Im  currentIm = null;
    boolean imExsit = false;
    for(int i = 0; i < m_ImOpened.size(); i++)
    {
      currentIm = (Im)m_ImOpened.elementAt(i);
      if(currentIm.getPartner().equals(partner))
      {
        imExsit = true;
        im = currentIm;
        sendText(im);
        break;
      }
    }

    if(!imExsit)
    {
      m_ImOpened.addElement(im);
      im.addImListener(this);
      im.open();
    }
  }

  /**
   * resolveFailed event receive aftere you send request to resolve a
   * name but no approrriate user was found.
   */
  public void resolveFailed(CommUIEvent event)
  {
    m_transcript.writeSeparator("resolve failed type another name",
                                Color.red);
    m_imDestination.setText("");
  }

  /**
   * if the user start the im (by im.open()) imOpened event will
   * be receved if the im open successfully
   */
  public void imOpened(ImEvent event)
  {
    event.getIm().sendText(false,m_imText.getText());
    m_transcript.writeSeparator(m_myName + ":  " +
                       m_imText.getText(), Color.black);
    m_imText.setText("");
  }
```

```
   /**
    * if the user start the im (by im.open()) imOpened event will
    * be receved openImFailed if the user you try to open im to
    * is offline or DND etc.
    */
  public void openImFailed(ImEvent evt)
  {
    System.out.println("openImFailed reason :"+evt.getReason());
    m_transcript.writeSeparator
      ("Failed to open im ,type another user name", Color.red);
  }

  /**
   * if, from any reason the im was closed.
   * you will need to remove the im from the open im list
   */
  public void imClosed(ImEvent event)
  {
    Im im = event.getIm();
    boolean imExsit = false;
    Im currentIm = null;

    for(int i = 0; i < m_ImOpened.size(); i++)
    {
      currentIm = (Im)m_ImOpened.elementAt(i);
      if(currentIm.equals(im))
      {
        imExsit = true;
        m_ImOpened.removeElement(im);
        im.close(0);
        im.removeImListener(this);
        break;
      }
    }
  }

   /**
   * textReceived print it to the screen
   */
  public void textReceived(ImEvent event)
  {
    String partnerName = event.getIm().getPartner().getDisplayName();
    m_transcript.writeSeparator(partnerName + ":  " +
                            event.getText(), Color.blue);
  }

  /**
   * DataReceived print a message to the screen
   */
  public void dataReceived(ImEvent evt)
  {
    System.out.println("Data Received data type = " +
                       evt.getDataType());
  }

  public void destroy()
  {
    m_comm.logout();
    m_session.stop();
    m_session.unloadSession();
  }
}
```

Lookup Service

Overview

The Lookup Service provides the ability to resolve user and group names and query group content. Most Sametime operations that deal with users or groups require `STUser` or `STGroup` objects as parameters. The resolve operation matches a name to provide either an `STUser` or an `STGroup` object.

To query group content, you must provide an STGroup object. The service attempts to return the full list of STUser objects in the group as registered in the directory of the Sametime server. For example, this service can be used to display a complete list of all the users in a group to the end user.

Description

Resolving Names

To resolve names, first call the CreateResolver() method of LookupService. This method returns a resolver object. To use this method, you must provide the following parameters:

- **OnlyUnique** – Determines whether the created resolver object should return only unique matches
- **ExhaustiveLookup** – Determines whether the created resolver object should perform an exhaustive lookup through all directories or stop in the first directory where a match is found
- **ResolveUsers** – Determines whether the created resolver should search the directory for users matching the provided name
- **ResolveGroups** – Determines whether the created resolver should search the directory for groups matching the provided name

To use the resolver object, you must first call the addResolveListener() method to register the resolve listener object. Then you can call one of the two resolve() methods (one for a single name and one for multiple names). You can call the resolve() method as many times as necessary on the same resolver object.

The results of a resolve operation are returned to the object implementing the resolveListener interface as one of three possible events:

- **resolved()** – The request has been resolved successfully, and a unique match has been found. Use the event object's getResolved() method to get the result.
- **ResolveConflict()** – The resolve request has been resolved successfully and multiple matches were found. Use the event object's getResolvedList() method to get the list of found matches. It is up to the application to determine the correct behavior for such as response.
- **ResolveFailed()** – The resolve request has failed. Use the event object's getReason() method to determine the reason for failure.

Querying Group Content

To query a group's content, first call the CreateGroupContentGetter() method of the LookupService. This method returns a GroupContentGetter object. To use the GroupContentGetter object, call the addGroupContentListener() method to register the listener object. Then call the queryGroupContent() method, passing the STGroup object. You can call the same groupContentGetter object's queryGroupContent() method as many times as necessary.

The results of a queryGroupContent operation are returned to the object implementing the GroupContentListener interface as one of two possible events:

- **groupContentQueried()** – The group content has been successfully queried. Call the event object's getGroupContent() method to get the actual group content.

- **queryGroupContentFailed()** – The group content of the queried group could not be queried for some reason. Call the event object's getReason() method to get the failure reason.

Package

`com.lotus.sametime.lookup`

See Also

The Resolve Panel section of this guide

The Community UI section of this guide

Example

See the Live Names Sample in the Awareness List section of Appendix I of this guide for sample code that shows how to resolve user names using the Lookup Service.

Multicast Service

Overview

The Multicast Service provides the ability to send messages to multiple recipients.

Description

Sending Multicast Events

To send multicast messages events, you must invoke the sendMultiCast() method on the Community Service with a list of recipients and the type and data of the message. The recipients are an array of STObjects that you must resolve beforehand – they can be a STUser, a STGroup, etc. NOTE: The recipients will only receive the multicast event if they have registered to listen for them.

Receiving Multicast Events

To receive multicast messages events, you must add a MulticastListener object to the Community Service. The multiCastReceived() event is called when the user has been sent a multicast message. Use the getData(), getType(), getSender() and getRecipients() method on the event object to retrieve more information.

Package

`com.lotus.sametime.community`

See Also

The Java Toolkit API JavaDoc.

Names Service

Overview

The Names Service helps manage user nicknames and name delimiters across the entire application. This service is a local service and has no equivalent server-side component.

Sametime Connect is an example of a Sametime-enabled application that uses nicknames. A user can define the nickname “Joe” for a user in his contact list, and that user will appear as “Joe” anywhere he is shown. Another option in Sametime Connect is to have “Automatic Nicknames” for users. This is a way to quickly shorten all the names of the users in the contact list by having the name cut at some predefined delimiter such as “/” so “Joe Smith/Org” will be shown as “Joe Smith.”

Description

The Names Service provides two different name management services:

- Synchronizing user name and nickname changes
- Synchronizing name delimiter changes

Synchronizing User Names

The Names Service allows you to store references to STUser objects that include a user’s name, nickname, and name delimiter and to be aware of changes to any of these users names. Store an STUser object reference by calling the setUserName() method. Retrieve an STUser object reference by calling the getUser() method. You can retrieve the most updated nickname for a specific user by calling the getNickname() method.

Application components that want to be aware of user name and nickname changes can register as a NameServiceListener on the service and receive the nameChanged`()` event each time a new user name or nickname has been set for a specific user.

Synchronizing User Names Delimiter

The Names Service allows you to set a global names delimiter string using the setNameDelimiter() method of the Names Service and to retrieve its value using the getNameDelimiter()method. Application components that want to be aware of name delimiter changes can register as a NameServiceListener on the Names Service and receive the nameDelimiterChanged() event each time a new delimiter string has been set.

Package

com.lotus.sametime.names

Places Service

Overview

The Places architecture defines a virtual place where users can meet and collaborate. The Places Service exposes the Places architecture and allows you to create virtual places. The flexibility of the Places architecture allows you to develop varied applications that take advantage of place-based awareness.

The Sametime Places architecture unifies the Meeting and Community Services in Sametime. It binds the activities provided by the Meeting Services (such as audio, video, application sharing, and the whiteboard) with a place provided by the Community Services. This architecture is a natural evolution of the Who Is Here Service provided in Sametime 1.5.

For example, an event-hosting application such as an auditorium could separate the audience from the presenters by putting them into different sections. The presenters could add activities such as whiteboard, audio, and video to broadcast the event to the audience. Attributes could be used to manipulate different features or permissions in the event.

Description

The Places Service exposes the Places architecture and allows you to create places and add meeting activities to them. The Places architecture defines the concept of a virtual place where users can meet and collaborate. A place consists of sections, activities, and attributes.

Refer to Chapter 8 of the *Sametime Java Toolkit Tutorial* for a detailed description of the Places architecture.

Places Service

The Places Service provides two basic functions:

- **Tracks availability of the Places Service**

  Tracking the availability of the Places Service is achieved by calling the isServiceAvailable() method of PlacesService and by implementing the PlacesServiceListener and adding it to the Places Service. The listener provides the events serviceAvailable() and serviceUnavailable() according to the state of the Places Service on the server. If the service is unavailable, the Places Service Component automatically attempts to reconnect to the service and will notify you with the serviceAvailable() event when it has successfully connected to the service.

- **Creates new Place objects**

  You create new Place objects by calling the createPlace() method of the Places Service. Provide the following parameters to the createPlace() method:

  - **placeName** – The first of two parameter uniquely identifying a place (see placeType at the end of this list)
  - **displayName** – The suggested display name for the place. The display name of a place is determined by its creator.
  - **encLevel** – The encryption level of messages in the place.
  - **placeType** – The second of two parameters uniquely identifying a place.

Place Object

The createPlace() method returns a Place object. Note that a virtual place on the Places Service of the Sametime server has not yet been created. The virtual place is created on the Sametime server only when you request to enter the place.

Entering a Place

To actually create the virtual place on the Sametime server you need to enter the place by calling one of the two enter() methods. When you enter a place you can specify whether you want to create a new place, join an existing place, or enter the place without caring whether you are creating a new one or joining an existing one. You can also specify whether to be put on the stage or in one of the other sections once you have entered the place. You can also specify the password if required for the place.

To know if you successfully entered the place, you must first implement a PlaceListener object and add it to the Place object. You will receive one of the following two events:

- **entered()** – The place was entered successfully. Use the getPlace() method of the event object to find the Place object this event references.
- **enterFailed()** – The request to enter the place has failed. Use the getPlace() method of the event object to find the place object this event references. Use the getReason() method of the event object to get the reason code for the failure.

Operations on a Place

Once you have entered a place you can do the following:

- **Use the addAllowedUsers() method to limit access to the place to a specified list of users.** If you do not call this method, any user can enter the place as long as he has the place name, type and password (if applicable). If the Places Service does not accept this request, you will receive an addAllowedUsersFailed() event on the PlaceListener object.
- **Add an activity to the place by calling the addActivity() method.** You receive either an activityAdded() event if successful or an addActivityFailed() event if unsuccessful. See the following Activity Objects section for more information on activities.
- **Get information about the place.** Use the following methods:
  - **getActivitiesNum()** – Returns the total number of activities currently in the place. This method is useful if you want to know how many activityAdded() events to expect when you enter the place.
  - **getMembers()** – Returns an enumeration you can use to go over all the members of the place: users, sections, activities, etc.

    Note: Users in sections that you are not listening to, will not be returned.
  - **getMySection()** – Returns the section object of the section you are currently in.
  - **getServer()** – Returns the server object on which the place has actually been created. In some cases this server might not be the same server as the one you are currently logged in to.
  - **getMyselfInPlace()** – Gets the MyselfInPlace object that represents the current user (myself) in the place.

- **Invite Sametime 1.5 clients to the place.** For backward compatibility reasons, an invite15User() method has been added. It allows Sametime 1.5 clients (Sametime Connect or toolkit clients) to be invited to place meetings without their being aware that they are actually in a place. From their point of view, they are in an old-style version 1.5 conference.

Receiving Section and Activity Events

You receive section and activity events by implementing the PlaceListener and adding it to the place using the addPlaceListener() method. When you enter a place, you receive sectionAdded() and activityAdded() events for each section and activity that was already exists in the place. Once in the place you will receive updates on sections and activities being added and removed with the sectionAdded(), sectionRemoved(), activityAdded(), and activityRemoved() events.

Leaving a Place

When you want to leave a place, call the leave() method of the Place object. You will receive a left() event from the PlaceListener. You can enter and leave a place multiple times. When you have no more use for the Place object, call the close() method. After calling the close() method, the Place object should no longer be used.

PlaceMember Interface

Many different kinds of objects are in a place (Place, Section, Activity, UserInPlace, and MyselfInPlace). These objects are all place members and therefore, they all implement the PlaceMember interface. This interface allows you to communicate with each member of a place by sending text and data messages to the member, and that you can change or remove the attributes of each member. Note that place member attributes are limited to the scope of a specific place, and their lifetime is limited to the duration of the place. Place member attributes have nothing to do with online attributes used by the Awareness Service (whose scope is the entire community) and with Storage Service attributes that are stored persistently on the Sametime server.

Sending Text and Data to a Place Member

Use the sendText() and sendData() methods of the PlaceMember interface to send text and data to specific place members. Using these methods, you can send a message to a specific user, to all users in a section, or to all users in a place by calling it on different place member objects. You can also use these methods to send messages to activities in the place since they are place members themselves. If the message could not be sent, you receive a sendFailed() event in the appropriate listener.

Manipulating Attributes of Place Members

You can manipulate the attributes of a place member by calling the changeAttribute() or removeAttribute() methods. If you want to retrieve the value of a known or heavy attribute you can use the queryAttrContent() method. The PlaceMemberListener provides an attributeChanged() event when an attribute has changed or if an attribute was queried on a place member. An attributeRemoved() event will be generated if an attribute was removed from a place member.

All of the above operations on place members might not be authorized, and can therefore fail. Trying to perform an unauthorized operation yields a **XXXFailed()** event with a reason code of ST_ERROR_PLC_NOT_AUTHORIZED. For example, trying to send a message to the place with the sendText() method while not being in a stage section creates the sendFailed() event.

Section Object

The Section object represents a section in the place and implements the PlaceMember interface. When you enter the place and have added a PlaceListener, you receive a snapshot of all the sections of that place. You also receive continuous updates on sections being added or removed from the place (see the Receiving Section and Activity Events section above). In Sametime every place has three sections and users cannot add or remove sections in a place. To determine if a section is a stage section, use the isStage() method.

Restricting the Section to Allowed Users

Use the Section object to restrict the section to a specified list of allowed users by calling the addAllowedUsers() method. You receive the addAllowedUsersFailed() event of the SectionListener if the allowed users could not be set for any reason. You can determine the total number of possible users in the section by calling the getCapacity() method.

Receiving Notifications on Users Entering and Leaving Sections

Using a SectionListener you can receive notifications on users entering and leaving the section. You receive a usersEntered() event each time users enter the section. The first time you add a SectionListener to a section you will receive a usersEntered() event with all the users who are in that section. You will receive a userLeft() event for each user who leaves the section.

Activity Object

The Activity object represents an activity in a place and implements the PlaceMember interface. Activities are added to the place using the addActivity()method of the Place object. Notification about activities being added and removed are received in the PlaceListener by the activityAdded() and activityRemoved() events.

Use the Activity object to retrieve information on the activity (such as its type or data), and whether you or another user added the activity.

UserInPlace Object

This object represents a user in the place and implements the PlaceMember interface. UserInPlace also extends STUserInstance (which extends STUser) and can therefore be used easily with other toolkit services that expect a STUser object.

MyselfInPlace Object

Because MyselfInPlace extends the UserInPlace class, it is a place member itself. You can use the MyselfInPlace object that is returned by the Place method getMyselfInPlace() to change your section in the place and to receive message events sent to you.

Changing Your Section

To change your section:

1. Implement MySectionListener and add it to the MyselfInPlace object with the addMySectionListener() method.

2. Call the changeSection() method providing the Section object representing the section you want to change to.

3. You will receive the sectionChanged() event if you had changed sections successfully. You will receive the changeSectionFailed() event if you have failed to change sections. Use the getReason() method of the event object to get the reason code for the failure.

Receiving Place Text and Data Messages

To receive messages sent to you by other members of the place, you should:

1. Implement MyMsgListener and add it to the MyselfInPlace object with the addMyMsgListener() method.
2. You will receive a textReceived() event when any member of the place sends you a text message. You will receive a dataReceived() event when a member of the place sends you a binary message.

Package

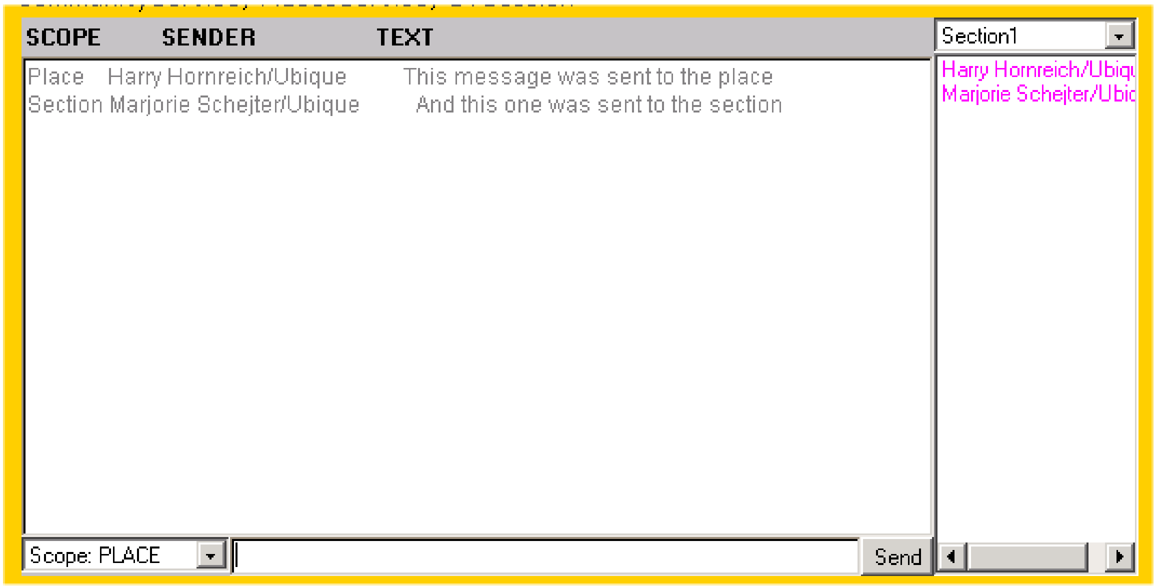
com.lotus.sametime.places

See Also

The Place Awareness List section of this guide

Example

This example demonstrates the concept of sections and scope in a place using the Places Service. This example logs you into the Sametime community and enters you into a predefined place. It allows you to navigate among sections in the place and to send messages to different scopes (user, section, and place).



```
Place    Harry Hornreich/Ubique      This message was sent to the place
Section  Marjorie Schejter/Ubique    And this one was sent to the section
```

Harry Hornreich/Ubiqu
Marjorie Schejter/Ubic

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Hashtable;
import com.lotus.sametime.places.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.constants.EncLevel;


/**
 * This applet allows chat over different sections of a place. It accepts 3
 * parameters from the HTML file: SERVER_NAME, LOGIN_NAME and password which
it
 * then uses to login.
 */
```

```
public class PlacesApplet extends Applet implements LoginListener,
    ActionListener, ItemListener, SectionListener, MyMsgListener,
MySectionListener
{
    /**
     * The place name.
     */
    private static final String PLACE_NAME = "SamplePlace";

    /**
     * The session object.
     */
    private STSession m_session;

    /**
     * The community service. (For login).
     */
    private CommunityService m_commService;

    /**
     * The places service.
     */
    private PlacesService m_placesService;

    /**
     * The place were going to deal with.
     */
    private Place m_place;

    /**
     * The current section wer'e in.
     */
    private Section m_mySection;

    /**
     * The list of sections in the place.
     */
    private Hashtable m_sections = new Hashtable();

    /**
     * The list of users in the current section we're in.
     */
    private Hashtable m_users = new Hashtable();

    // UI components.
    private Button    m_btnSend;
    private TextField m_tfSend;
    private TextArea  m_taTranscript;
    private Choice    m_chSections;
    private Choice    m_chScope;
    private List      m_peopleList;

    //
    // Applet life cycle.
    //
    /**
     * Initialize the sametime components.
     */
    public void init()
    {
        try
        {
            m_session = new STSession("Places Session" + this);
            m_session.loadSemanticComponents();
            m_session.start();
        }
        catch(DuplicateObjectException e)
        {
            e.printStackTrace();
        }

        // Get a reference to the components we are interested in.
        m_commService = (CommunityService)
                          m_session.getCompApi(CommunityService.COMP_NAME);
        m_commService.addLoginListener(this);

        m_placesService = (PlacesService)
                            m_session.getCompApi(PlacesService.COMP_NAME);

        initUI();
    }

    /**
     * Login to the community server.
     */
    public void start()
    {
        String serverName = getCodeBase().getHost().toString();
        String loginName  = getParameter("loginName");
        String password   = getParameter("password");

        m_commService.loginByPassword(serverName,
                                      loginName, password);


    }

    /**
     * Logout.
     */


    public void destroy()
    {
      m_commService.logout();
      m_session.stop();
      m_session.unloadSession();
    }

    /**
     * A notification that wer'e logged in.
     */
    public void loggedIn(LoginEvent event)
    {
        System.out.println("Sample: Logged in");

        // Create the place wer'e going to work with, and enter.
        m_place = m_placesService.createPlace(PLACE_NAME, PLACE_NAME,
                                              EncLevel.ENC_LEVEL_DONT_CARE, 0
);
        // Add an adapter to be the listener to this place.
        addPlaceListener();
        m_place.enter();
    }

    /**
     * Logged out of the server.
     */
    public void loggedOut(LoginEvent event)
    {
        System.out.println("Sample: Logged out");
    }
    /**
     * Add a PlaceAdapter to listen to the events we care about.
     */
    private void addPlaceListener()
    {
        m_place.addPlaceListener(new PlaceAdapter()
        {
            public void entered(PlaceEvent event)
            {
                PlacesApplet.this.entered(event);
            }

            public void enterFailed(PlaceEvent event)
            {
                PlacesApplet.this.enterFailed(event);
            }

            public void left(PlaceEvent event)
            {
                PlacesApplet.this.left(event);
            }

            public void sectionAdded(PlaceEvent event)
            {
                PlacesApplet.this.sectionAdded(event);
            }

            public void sendFailed(PlaceMemberEvent event)
            {
                PlacesApplet.this.sendFailed(event);
            }

        });
    }

    //
    // Place listener interface.
    //

    /**
     * Entered the place.
     */
    private void entered(PlaceEvent event)
    {
        // Listen to my section in order to get notification on its users.
        m_mySection = m_place.getMySection();
        m_mySection.addSectionListener(this);

        MyselfInPlace myself = m_place.getMyselfInPlace();

        // Listen for incoming messages.
        myself.addMyMsgListener(this);

        // Listen for section changes.
        myself.addMySectionListener(this);

        enableGuiItems(true);
    }

    /**
     * Failed to enter the place. Better luck next time.
     */
    private void enterFailed(PlaceEvent event)
    {
        System.out.println("Sample: Failed to enter to the place. Reason: " +
                           event.getReason());
    }

    /**
     * Left the place for some reason.
     */
    private void left(PlaceEvent event)
    {
        System.out.println("Sample: Left the place. Reason: " +
                           event.getReason());

        enableGuiItems(false);
    }

    /**
     * New section in the place.
     */
    public void sectionAdded(PlaceEvent event)
    {
        Section newSection = event.getSection();
        Integer sectionId = newSection.getMemberId();
        String sectionKey = "Section" + sectionId.toString();

        m_sections.put(sectionKey, newSection);
        m_chSections.add(sectionKey);
    }

    //
    // Section listener.
    //
    /**
     * New users have entered the section.
     */
    public void usersEntered(SectionEvent event)
    {
        String userName;
        UserInPlace[] newUsers = event.getUsers();
        for (int i = 0; i < newUsers.length; i++)
        {
            userName = newUsers[i].getDisplayName();
            m_users.put(userName, newUsers[i]);
            m_peopleList.add(userName);
        }
    }

    /**
     * A user has left the section.
     */
    public void userLeft(SectionEvent event)
    {
        String userName = event.getUser().getDisplayName();
        m_users.remove(userName);
        m_peopleList.remove(userName);
    }

    /**
     * A attempt to send a message was failed.
     */
    public void sendFailed(PlaceMemberEvent event)
    {
        m_taTranscript.append("\n ************Failed to send the message.
Reason: " +
                              Integer.toHexString(event.getReason()) +
                              "h ************ \n\n");
    }
    /**
     * AWT events.
     */
    public void actionPerformed(ActionEvent event)
    {
        if (event.getSource() == m_btnSend)
        {
            PlaceMember receiver;
            String text = m_tfSend.getText();
            String scope = m_chScope.getSelectedItem();

            // Get the receiver place member according to the selected scope.
            if (scope.equals("Scope: PLACE"))
            {
                receiver = m_place;
            }
            else if (scope.equals("Scope: SECTION"))
            {
                receiver = m_mySection;
            }
            else
            {
                String selectedUser = m_peopleList.getSelectedItem();
                receiver = (UserInPlace)m_users.get(selectedUser);
            }

            receiver.sendText(text);
            m_tfSend.setText("");
        }
    }

    /**
     *
     */
    public void itemStateChanged(ItemEvent event)
    {
        if (event.getSource() == m_chSections)
        {
            String sectionKey = (String)event.getItem();

            Section newSection = (Section)m_sections.get(sectionKey);
            MyselfInPlace myself = m_place.getMyselfInPlace();

            // Move to the selected section. See sectionChanged() below.
            myself.changeSection(newSection);
       }
    }

    //
    // My section listener.
    //

    /**
     * Navigated to a new section.
     */
    public void sectionChanged(MyselfEvent event)
    {
        System.out.println("Sample: Section changed");

        m_users.clear();
        m_peopleList.removeAll();

        m_mySection.removeSectionListener(this);
        m_mySection = event.getSection();
        m_mySection.addSectionListener(this);
    }
    /**
     * Unable to switch sections.
     */
    public void changeSectionFailed(MyselfEvent event)
    {
        System.out.println("Sample: Failed to change a section. Reason: " +
                           event.getReason());
    }

    //
    // My Msg Listener.
    //

    /**
     * A text message was received.
     */
    public void textReceived(MyselfEvent event)
    {
        PlaceMember sender = event.getSender();
        if (sender instanceof UserInPlace)
        {
            String senderName = ((UserInPlace)sender).getDisplayName();
            String scope;
            if (event.getScope() == PlacesConstants.SCOPE_PLACE)
            {
                scope = "Place    ";
            }
            else if (event.getScope() == PlacesConstants.SCOPE_SECTION)
            {
                scope = "Section ";
            }
            else
            {
                scope = "User     ";
            }

            m_taTranscript.append(scope + senderName + "          ");
            m_taTranscript.append(event.getText() + "\n");

            m_tfSend.requestFocus();
        }
    }

    public void dataReceived(MyselfEvent event)
    {}

    //
    // Helpers.
    //

    /**
     * Set up the ui components.
     */
    private void initUI()
    {
        setLayout(new BorderLayout());

        Panel sendPanel = new Panel(new BorderLayout());
        sendPanel.add("East", m_btnSend = new Button("Send"));
        sendPanel.add("Center", m_tfSend = new TextField());
        sendPanel.add("West", m_chScope = new Choice());
        m_chScope.add("Scope: PLACE");
        m_chScope.add("Scope: SECTION");
        m_chScope.add("Scope: USER");
        Panel chatPanel = new Panel(new BorderLayout());
        chatPanel.add("South", sendPanel);
        chatPanel.add("Center", m_taTranscript = new TextArea());
        m_taTranscript.setFont(new Font("Courier New", Font.PLAIN, 12));
        m_taTranscript.setForeground(Color.blue);
        m_taTranscript.setEnabled(false);

        Label header = new Label();
        header.setFont(new Font("Dialog", Font.BOLD, 12));
        header.setText("SCOPE        SENDER                TEXT");
        chatPanel.add("North", header);

        Panel listPanel = new Panel(new BorderLayout());
        listPanel.add("North", m_chSections = new Choice());
        listPanel.add("Center", m_peopleList = new List());
        m_peopleList.setForeground(Color.magenta);

        m_btnSend.addActionListener(this);
        m_chSections.addItemListener(this);

        // Enable the gui items only after we enter to the place.
        enableGuiItems(false);

        add("East", listPanel);
        add("Center", chatPanel);
    }

    private void enableGuiItems(boolean enable)
    {
        m_btnSend.setEnabled(enable);
        m_tfSend.setEnabled(enable);
        m_peopleList.setEnabled(enable);
        m_chScope.setEnabled(enable);
        m_chSections.setEnabled(enable);

        m_tfSend.requestFocus();
    }

    public void addAllowedUsersFailed(SectionEvent event)
    {}

    public void removeAllowedUsersFailed(SectionEvent event)
    {}

    public void removeAttributeFailed(PlaceMemberEvent event)
    {}

    public void attributeChanged(PlaceMemberEvent event)
    {}

    public void attributeRemoved(PlaceMemberEvent event)
    {}

    public void queryAttrContentFailed(PlaceMemberEvent event)
    {}

    public void changeAttributeFailed(PlaceMemberEvent event)
    {}
}
```

Post Service

Overview

The Post Service allows the sending of one-time messages to many users at once, and allows these users to respond to the message they received. An example of a use for this service is a company secretary sending a meeting reminder to all the meeting participants. Another use is to send invitations to a new meeting type.

Description

Sending a Post Message

Most of the functionality of the Post Service is implemented in the Post object.
You create a post object by calling one of the two createPost() methods of the PostService. Once you have the Post object, you can set its title, message, and details; and you can add users to the list of users to whom the post message should be sent. Before sending the post message you should add a PostListener object. Call the send() method to send the prepared post message to all users. For each user that the post message could not be sent to, you receive the PostListener event sendToUserFailed(). For each user that responds to the post message, you receive the PostListener event userResponded().

You can use the same Post object to send more than one post message.

Receiving and Responding to a Post Message

To receive Post messages, you must perform two preparation steps:

1. Register the post types you are willing to receive
   by calling the registerPostType() method of PostService.

2. Add a PostServiceListener object by calling the addPostServiceListener() method of PostService.

You will now receive messages posted to you by the posted() event of the PostServiceListener. By using the getPost() method of the event object, you can retrieve the Post object to learn more about who posted you. You can respond to the postee by calling the respond() method of the Post object.

Note that only post messages having a type that you have registered will be received and the posted() event generated. The Post Service will automatically reject post messages of other types.

Package

com.lotus.sametime.post

See Also

The Instant Messaging Service section of this guide

Storage Service

Overview

The Storage Service stores user-related information as attributes directly on the Sametime server. The Storage Service allows you to access these attributes from wherever you log in to your Sametime server. For example, in Sametime your Sametime Connect contact list is stored on the Sametime server. You can log in using Sametime Connect from *any* machine and always get your latest contact list. Your Sametime applications can use this service to access standard Sametime attributes, such as contact list and Sametime Connect preferences, or add their own application-specific attributes.

Do not confuse the attributes used in this service with the online attributes of the Awareness or Places Services. The Storage Service persistently stores user-related attributes for a specific user, and only that user can retrieve the attribute values. The Awareness Service allows a user to set non-persistent online attributes that can be listened to by other users in the community. The Places Service has similar attributes, but they exist in the context of a place.

Description

Before beginning to use the Storage Service, you must:

1. Implement StorageServiceListener and add it to the Storage Service by calling the addStorageServiceListener() method.

2. Decide whether you want to enable request buffering. When buffering is enabled, the storage requests will be buffered until the Storage Service on the Sametime server is available. Use the enableBufferring() method of the Storage Service to set this option. By default, request buffering is disabled.

StorageServiceListener provides two events to let you know if the Storage Service is currently available or not: serviceAvailable() and serviceUnavailable(). If the service is unavailable for some reason, you will receive the serviceUnavailable() event, and the service component will automatically reconnect to the service when it is available. Once the service is available again, you will receive the serviceAvailable() event.

The Storage Service provides the basic store and query operations.

Storing Attributes

To store an attribute you must create or have an STAttribute object. Call either the storeAttr() or storeAttrList() method and record the returned request ID.

If you use the storeAttr() method, you receive an attrStored() event. Use the getRequestId() method of the event object to compare the event request ID with the request ID you recorded previously. Use the getRequestResult() method of the event object to find out if the store request was successful. The possible values are:

- **ST_OK** – All the attributes were stored successfully. Use the getAttrList() method of the event object to get the list of attributes that were stored successfully.

- **ST_FAIL** – None of the attributes was stored successfully. Use the getAttrList() method of the event object to get the list of attributes that failed to be stored.

Querying Stored Attributes

To query an attribute, you must know the attribute key or keys; and then call the queryAttr() or queryAttrList() method and record the returned request ID. As a response to a query request, you will receive an attrQueried()

event. Use the getRequestId() method of the event object to compare the event request ID with the request ID you recorded previously. Use the getRequestResult() method of the event object to find out if the query request was successful or not. The possible values are:

- **ST_OK** – All the requested attributes were retrieved successfully.
  Use the getAttrList() method of the event object to get the list of successfully retrieved attributes.

- **ST_ATTRS_NOT_EXIST** – Some of the requested attributes were not retrieved because they had not been stored on the Sametime server. Use the getAttrList() method of the event object to get the list of successfully retrieved attributes. Use the getFailedAttrKeys() method of the event object to get the list of attribute keys that were not retrieved.

- **ST_FAIL** – None of the attributes were retrieved. Use the getFailedAttrKeys() method of the event object to get the list of attribute keys that were not retrieved.

Receiving Attribute Updates

StorageServiceListener also has the attrUpdated() event. This event is fired if an attribute was changed by a different login of the same user. In other words, the user is running more than one client, and one of the clients stored some attribute using the Storage Service. Use the getUpdatedKeys() method of the event object to get the array of attribute keys that have changed. Note that these are only the attribute keys and not the actual values of the changed attributes. Use the queryAttr() method to retrieve the actual changed attribute values if needed.

Attributes are shared by all logins of the same user. The Storage Service stores only a single copy of each attribute on the Sametime server for each user.

Package

com.lotus.sametime.storage

Example

See the Live Names Sample in the Awareness List section of Appendix I of this guide for code that shows how to use the Storage Service to store and retrieve attributes.

Token Service

Overview

The Token Service provides the ability to generate a token that can be used by a logged-in user to log in to the Sametime server without being challenged to authenticate again. For example, a Sametime application could generate a token and pass it as a parameter to a second application. The launched application would then use this token to log in to the community. A token has a time limit, depending on adminstrator configuration.

Description

The Token Service returns a Token object. Use this object's getLoginName() and getTokenString() methods to get the login name and token string to pass to the Community Service method loginByToken().

To get a login token using the TokenService:

1. Add a TokenServiceListener object by calling the addTokenServiceListener() method of Token Service.

2. Call the generateToken() method of Token Service.

3. Receive the token in the tokenGenerated() event of the TokenServiceListener object.

Package

com.lotus.sametime.token

See Also

The Community Service section of this guide

XML Buddy List Service

Overview

In Sametime 7.5, a new, more flexible XML based Buddy List format was introduced. The XML Buddy List Service provides the ability to get this new XML version of the user's contact list from the user storage on the Sametime server, or to store it there. It also provides automatic synchronization between the new XML and old classic formats. These capabilities enable the developer to avoid dealing with the low-level protocols defined by the Storage Service.

Description

Before beginning to use the XML Buddy List Service, you must:

- Load the XmlBuddyListService component into the Sametime session object, like any other toolkit component.

- Implement XmlBuddyListListener, and add it to your XML Buddy List Service instance by calling the method:

```
addBuddyListListener ();
```

XmlBuddyListListener provides a single method which you can implement to handle all buddy list events:

- `void handleBuddyListEvent(XmlBuddyListEvent event);`

The handleBuddyListEvent() method will initially trigger with one of the following XmlBuddyListEvent IDs which indicate whether or not the XmlBuddyListService is currently available:

```
XmlBuddyListEvent. BUDDYLIST_SERVICE_AVAILABLE

XmlBuddyListEvent. BUDDYLIST_SERVICE_UNAVAILABLE
```

The XmlBuddyList Object

The XmlBuddyList object contains groups that are encapsulated by the XmlPrivateGroup and XmlPublicGroup classes implementing the XmlGroup interface.

Get the Contact List from the User Storage

The developer can get a contact list using this method, which is defined in the XmlBuddyListService:

```
void getBuddyList();
```

Note that this method does not return any value. The asynchronized result is retrieved by the event handler. When the buddy list has been successfully retrieved, the event ID of the XmlBuddyListEvent from the handleBuddyListEvent() handler method will be:

```
XmlBuddyListEvent.BUDDYLIST_RETRIEVAL_SUCCEEDED
```

The XmlBuddyList object can then be extracted from the XmlBuddyListEvent:

```
XmlBuddyList m_buddyList = evt.getBuddyList();
```

In case of failure, the event ID of the XmlBuddyListEvent from the handleBuddyListEvent() handler method will be:

```
XmlBuddyListEvent. BUDDYLIST_RETRIEVAL_FAILED
```

Set the Contact List to the User Storage

The following method (defined in the XmlBuddyListService) is used to write a new Contact list to user storage:

```
void saveBuddyList(XmlBuddyList newBuddyList)
```

If the set action succeeds, the user will get an XmlBuddyListEvent through the event handler, with an ID of:

```
XmlBuddyListEvent. BUDDYLIST_SAVE_REMOTELY_SUCCEEDED
```

In case of failure, the event ID will be:

```
XmlBuddyListEvent.BUDDYLIST_SAVE_REMOTELY_FAILED
```

BuddyList Updated

When the XML contact list has been updated by some other login, the user will get an XmlBuddyListEvent through the event handler with an ID of:

```
XmlBuddyListEvent. BUDDYLIST_UPDATED
```

NOTE: This event just indicates that the buddy list was updated on the server, it does not contain the actual updated buddy list.

Package

com.lotus.sametime.buddylist.xml

See Also

The Buddy List and Storage Service sections of this chapter

Appendix A. Enabling Your Domino Applications

This appendix describes the process for enabling a Domino database template with Sametime. The STJavaSample.nsf sample application contains all the agents and script libraries you need to enable your template.

The agents do the following:

1. Generate an access token for the user.

2. Locate a Sametime server from which the applet will be loaded.

Using Tokens

Token authentication allows a user who has logged in to your application to log in to the Sametime server without being challenged to authenticate again. The application generates a token and passes it to the Sametime applet. The applet logs in by token to the Sametime server.

The Sametime server includes two separate security features capable of generating the authentication token used by Sametime:

- **Domino Single Sign-On (SSO) Authentication** – This authentication method uses Lightweight Third Party Authentication (LTPA) tokens.

- **Secrets and Tokens authentication databases** – Releases of Sametime before 3.0 used only the Secrets and Tokens authentication databases to create authentication tokens. For this reason, the Sametime server must support both the Domino SSO feature and the Secrets and Tokens authentication system when Sametime operates in environments that include previous versions of Sametime.

The recommended authentication mode for applications that work with a Sametime server is LTPA. However, if the Domino server uses a DSAPI filter for authentication instead of LTPA, the application can authenticate with a Secrets and Tokens token. The sample application STJavaSample.nsf first looks for an LTPA token. If the LTPA token cannot be found, a Secrets & Tokens token is generated.

Locating the Sametime Server

Sametime-enabled applications running on a non-Sametime server must locate the user's home Sametime server, or a Sametime server in the user's LAN. When the application generates a Web page that contains a Sametime applet, it also needs to know the codebase of the applet – that is, a Sametime server that contains the applet.

You must create your application so that it can find the server information it needs. The previous versions of the Java toolkit recommended that you use a LotusScript® agent to look in the Domino Directory for the user's person record and the associated home server. This method currently works only with the Domino Directory. You cannot use it with LDAP directories or for anonymous access.

A more reliable, efficient, and flexible method is to deploy a special XML file, hostAddress.xml, on the application server. This file contains a nearby Sametime server location. This method requires some simple administrator intervention; but it uses less code, performs more efficient lookup, works with both Domino and LDAP directories, and works with both anonymous and authenticated access.

The format of this XML file is:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<sametime>
         <hostAddress>myhost.lotus.com</hostAddress>
         <httpPort>80</httpPort>
</sametime>
```

The Sametime server installation process generates this XML file, inserting the address of the Sametime server on which the toolkits are installed. It then stores the XML file in the "sametime" directory under the HTTP root. If you are using a Sametime 2.5 server, you can create and store the file manually.

You must deploy this XML file to the same location (the “sametime” directory under the HTTP root) on all of your application servers. The application can then use HTTP to read the address of the Sametime server from this XML file and load the applet from the server.

The Sametime server whose address appears in the XML file is not necessarily the user's home server. However, it will redirect the user to the appropriate home server.

A Sametime server updates the generated XML file automatically if the host address changes. If your application uses the file, you must then re-deploy the updated file to your application servers.

The agent SametimePopulateHostFields in the STJavaSample.nsf sample application works with Sametime servers and the XML file to locate the Sametime server. You can use this agent in your own applications. See the "Using the Sample Domino Application" section that follows.

Using the Sample Domino Application

The sample application STJavaSample.nsf is located in the Java Toolkit code directory. It is a Notes and Web application database – it can be viewed both in a browser and in the Notes client.

To use this sample, copy it to the Domino Data directory. Open the database using either a browser or the Notes client. If login is successful, you should see your name turn green.

The sample uses a sample applet, TokenLoginApplet, that demonstrates login by token. There is a link to the Java source code for this applet in the toolkit Samples Web page.

Enabling Your Template

The following steps are required to enable an application or template with Sametime:

1. Prepare the template.

2. Enable a form.

3. Test the template.

4. Distribute the template.

The Sametime script libraries and agents are required to enable your template with Community and Meeting Services. The processes are slightly different for Web templates and Notes templates; these differences are noted below.

Step 1: Prepare the template

For the template to be enabled with Sametime, it must contain the Sametime script libraries and agents.

Copy the Sametime script libraries to your template

Copy the following script libraries from the STJavaSample.nsf database in the Script Libraries view, and paste them into the same view in your template:

- SametimeHost
- SametimeAgent
- SametimeClient
- SametimeStrings

Copy the Sametime agents to your template

Copy the following agents from the Sametime Discussion or Teamroom® template in the Agents view and paste them into the same view in the template that you want to enable:

- **Web templates** – SametimePopulateTokenLTPA, SametimePrePopulateTokenST, SametimePopulateTokenST and SametimePopulateHostField agents
- **Notes templates** – SametimeJavaGetLtpaToken and SametimeCreateTokenNotes agents

Sign the agents

Because of Sametime's security features, the agent must be signed by an authorized developer to run on a Sametime or Domino server. Your server administrator should grant you the required server access controls. Once you are granted proper access, sign the Sametime agents that you copied to your template.

1. Ask the server administrator to add your name to the "Run unrestricted LotusScript/Java Agents" list in the Agent Manager section of the server document for the Sametime or Domino server where you want to put the application.
2. Choose agents in the enabled database's client menu. Double-click the agent that you want to sign.
3. Save the agent. The agent is now signed.

To verify that the agent was signed successfully:

1. Right-click the agent and choose Properties.
2. Click the Fields tab and check that the $UpdatedBy field contains your name.

You might need to repeat the agent-signing procedure for each new server on which the template will be installed.

Step 2: Enable a form

Enabling a Web form

To enable a form in a Web template:

1. Add three hidden computed-for-display text fields. Name them HTTP_Cookie, Server_name, and Server_Port, after the CGI variables that are used by the agents. Specify an empty string as the value formula for each field. Domino will place the CGI variable values in the fields automatically.

2. Add the following lines to the WebQueryOpen event of the form:

   ```
   @Command([ToolsRunMacro]; "SametimePopulateHostField");
   @Command([ToolsRunMacro]; "SametimePopulateTokenLTPA");
   @If((SAMETIME_TOKEN="");
       @Do(@Command([ToolsRunMacro]; "SametimePrePopulateTokenST");
           @Command([ToolsRunMacro]; "SametimePopulateTokenST"));"")
   ```

If you are going to deploy your application on a Sametime server from a previous Sametime release (2.5 or earlier), remove the command that runs the agent SametimePopulateTokenField; LTPA tokens are not supported in previous releases.

3. Use Pass-Thru HTML to add a Sametime applet to the form. Mark the whole paragraph starting with the “<applet>” tag and ending with the “</applet>” tag as Pass-Thru HTML.

   For the CODEBASE property, use:

   ```
   CODEBASE=http://[SAMETIME_HOST]/sametimeapplets/
   ```

   The [SAMETIME_HOST] is a computed-for-display text field. The name of the field should be SAMETIME_HOST. Specify an empty string as the value formula. The actual value is computed by the SametimePopulateHostField agent.

The codebase is set to the directory called “sametimeapplets”on the user’s home Sametime server. Put your applet class and archive files in that directory. If you want to put your applet files in a different directory, change the CODEBASE property accordingly.

   If your applet uses token-based authentication, it should be given access to the user name and Sametime token by providing the applet with the following parameters:

   ```
   <PARAM NAME="loginName" VALUE="[loginName]">
   ```

   The loginName is a computed-for-display text field. The value formula is “@UserName.”

   ```
   <PARAM NAME="token" VALUE="[SAMETIME_TOKEN]">
   ```

   The [SAMETIME_TOKEN] is a computed-for-display text field. The field name should be SAMETIME_TOKEN. Specify an empty string as the value formula. The actual value is computed by the SametimePopulateTokenField agent.

Make sure that your applet gets these parameters and uses them to log on to the Sametime server.

Enabling a Notes Client form

To enable a form in a template for the Notes client:

1. In the (Options) script of the form object, write:

   ```
   Use "SametimeClient"
   Use "SametimeHost"
   ```

2. In the PostOpen event of the form, write:

   ```
   Dim token As String
   Dim host As String

   ' Get the Authentication token
   token = SametimeGetToken()
   Call gSametimeSession.SetEnvironmentVar(ENV_ST_TOKEN, token)

   ' Get the host name of the sametime server
   host = SametimeGetHostNotes()
   Call gSametimeSession.SetEnvironmentVar(ENV_ST_HOST, host)
   ```

3. In the QueryClose event of the form, write: the following:

```
Call gSametimeSession.SetEnvironmentVar(ENV_ST_TOKEN, "")
Call gSametimeSession.SetEnvironmentVar(ENV_ST_HOST, "")
```

4. Add a Sametime applet to the form. You can either import the applet, or link to the applet on the Sametime server.

5. Add a parameter named "codebase" to the applet. The parameter formula should be "http://" + @Environment("SThost") + "/sametimeapplets".

   The codebase parameter is set to the directory called "sametimeapplets "on the Sametime server. The applet will be loaded from the codebase parameter, so you must put the applet's class and archive files in that directory. Although you might have imported the applet or linked to another location, the applet will actually be loaded from the computed codebase parameter.

6. If your applet uses token-based authentication, it should be given access to the user name and Sametime token, as follows:
   - Add a parameter named "loginName" to the applet. The parameter formula should be "@UserName".
   - Add a parameter named "token" to the applet. The parameter formula should be @Environment("STtoken").

   Your applet should get these parameters and use them to log on to the Sametime server.

To enable a form for both Web and Notes access, follow the instructions for enabling a Web form and for enabling a Notes form. You will place two copies of the applet on the form, one for Web access and one for Notes access. Use the settings "Hide paragraph from Notes R4.6 or later" and "Hide paragraph from Web browsers" to hide the applets.

Step 3: Test the template

The last step before deploying the Sametime-enabled application on the Sametime server is to test the application for awareness and conversation capabilities. Typically, this testing means having several users online and having them see each other online and communicate by sending real-time messages.

Step 4: Distribute the template

Once the template has been tested, it can be deployed across other Sametime and Domino servers.

You might have to repeat the agent-signing procedure for each new server where the template will be installed.

Appendix B. Component Dependencies for Loading and Packaging

The first step in using the Sametime Java Toolkit is to create a Sametime session and load the components into it. This appendix describes component dependencies and restrictions, the various ways to load components, and how to load components selectively.

Loading a selected group of components allows you to use some of the Sametime capabilities, while removing others. One example is to load only the service components and to implement your own user interface (UI), replacing the standard Sametime UI with your own. Removing unneeded components also allows you to create clients that are more compact.

Methods of Loading Components

The standard way to load components is by calling STSession.loadAllComponents(). Currently this method is limited to loading only the community components. Alternative ways of loading components are also provided by the toolkit and are described below.

Using STSession.loadComponents

The STSession.loadComponents() method accepts an array of component names and loads them into the session. The names of the components should be fully qualified Java class names of the classes that implement the component interfaces. (See the “Available Components” section below for a list of these classes.)

This method will skip components it cannot load. Note that the meeting components cannot be loaded in this manner since they have unique constructors.

An example implementation is:

```
STSession session = new STSession("A name");
String[] components = {
    "com.lotus.sametime.community.STBase",
    "com.lotus.sametime.im.ImComp"
    };
session.loadComponents(components);
IMService im = session.getCompApi(IMService.COMP_NAME);
```

Creating Components Directly

An alternative way of loading components is to instantiate the component objects yourself, using the “`new`” operator with the component implementation class. To the component constructor, pass a reference to the containing Sametime session. The component will register itself to that session. You can also keep a pointer to the component while loading it and avoid using the STSession.getCompApi() method.

An example implementation is:

```
STSession session = new STSession("A name");
new STBase(session);
// Keep the reference to the IM component,
// instead of calling 'getCompApi'.
IMService im = new ImComp(session);
```

Using STSession.loadAllComponents

It is possible to load components selectively even if you use the STSession.loadAllComponents() method. This method tries to load all community components. If a component is not found, it will be skipped. Therefore, you can remove unwanted components when packaging your application and those components will not be loaded.

Restrictions

Keep the following restrictions in mind when loading components:

- The meeting components must be loaded explicitly using the `new` operator.
- Dependencies exist between components. When you load a component selectively, you need to make sure that you also load all the components it depends upon. See the section below, "Component Dependencies," for more details.
- The UI dialogs and panels provided with Sametime need to have all the components available. If you use the Sametime UI, you must load all the components.
- An STSession cannot hold more than one instance of a component of a given type. Each type of component can only be loaded once. If you try to load a second instance of a component into a session, a DuplicateObjectException will occur.
- The Community Services components must be loaded before the session is started.
- If you are creating an applet that uses the AwarenessList AWT component, and you want to support file transfer (through the Send File menu option that appears when a user right-clicks an online name in the Awareness List), you must sign the applet, to allow local file access. If you want to use the AwarenessList AWT component in an unsigned applet, file transfer will not be supported. In this case, load the components your applet requires separately (instead of using STSession.loadAllComponents), and omit the FileTransferUI component. Otherwise, the menu option will appear but will not work.

Available Components

The following tables list the available Sametime components, the names of their interfaces, and the implementation class. The implementation class can be passed to the STSession.loadComponents() method or created directly.

Table listing available component name, interface, and implementation class.

Service Components		
Name	**Component Interface**	**Implementation Class**
Announcement	AnnouncementService	com.lotus.sametime.announcement.AnnouncementComp
Awareness	AwarenessService	com.lotus.sametime.awareness.AwarenessComp
Community	CommunityService	com.lotus.sametime.community.STBase
Directory	DirectoryService	com.lotus.sametime.directory.DirectoryComp
File Transfer	FileTransferService	com.lotus.sametime.filetransfer.FileTransferComp
Instant messaging	InstantMessaging Service	com.lotus.sametime.im.ImComp
Lookup	LookupService	com.lotus.sametime.lookup.LookupComp
Names	NamesService	com.lotus.sametime.names.NamesComp
Places	PlacesService	com.lotus.sametime.places.PlacesComp
Post	PostService	com.lotus.sametime.post.PostComp
Storage	StorageService	com.lotus.sametime.storage.StorageComp
Token	TokenService	com.lotus.sametime.token.TokenComp

Component Dependencies

Components are interdependent. When loading a component selectively, make sure that all the components it depends on are also loaded. The table below lists the component dependencies.

Table listing available component dependencies.

Service Components	
Component Name	**Depends on**
Announcement	Community
Awareness	Community
Buddy List	Community, Storage
Community	-
Directory	Community
FileTransfer	Community
Instant messaging	Community
Lookup	Community
Names	-
Places	Community
Post	Community, Im
Storage	Community
Token	Community

Appendix C. Sametime Identifiers

Sametime services use identifiers for many purposes. Among them:

- Online users' and servers' attribute keys
- Instant Message types, data message types and sub-types
- Persistent user storage attribute keys
- Post types

It is important that Sametime-enabled applications do not conflict with each other by using the same identifier for different purposes. That can happen, for example, if two separate companies produce Sametime-enabled products that use the same identifier for two different purposes. That is why the following conventions have been established for all Sametime identifier types:

- The range 0 – 100,000 is reserved for internal Sametime/Lotus/IBM use

  Any third-party developers who want to allocate an identifier range for their Sametime-enabled applications should e-mail Sametime_ID_Request@lotus.com.

This Appendix lists the identifiers that might be useful to developers using the Sametime Toolkits. The identifiers are already in use by Sametime applications such as Sametime Connect and Meeting Room Client. The identifiers are grouped by service. Please see the relevant chapter in this *Developer's Guide* for more information on how these identifiers are used.

Awareness Service

Table listing identifiers of awareness service.

Attribute Key ID	Attribute Name	User/ Server	Attribute Type	Value Format	Comments
1	Have I set my A/V Preferences?	User	Existential		Identifies whether the attributes 0002-0004 have been set by the user.
2	Do I have a Mic?	User	Existential		Identifies whether the user has a microphone.
3	Do I have Speakers?	User	Existential		Identifies whether the user has speakers.
4	Do I have a Video Camera?	User	Existential		Identifies whether the user has a video camera.
6	Do I Support File Transfer	User	Existential		Set by the toolkit UI or application level. Used to let other clients know whether a user supports file transfer, prior to sending a file.
9001	Are meeting services available?	Server	Existential		Will not be set for customers that do not have meeting services (like IBM etc.).
9002	Is Audio enabled?	Server	Existential		Requires the A/V Add-on to be installed and Audio enabled by the admin.
9003	Is Video	Server	Existential		Requires the A/V Add-on to be installed

	Enabled?				and Video enabled by the admin.
9004	Can we browse the directory?	Server	Existential		Will not be set in LDAP environment in Watson.
9005	Automatic Nicknames delimiter	Server		String	Usually "/"
9006	AppShare Can View Setting	Server		Byte 0 or 1	Identifies whether users connecting to this server can view appshare content.
9007	AppShare Can Host Setting	Server		Byte 0 or 1	Identifies whether users connecting to this server can host appshare content.
9008	AppShare Can Drive Setting	Server		Byte 0 or 1	Identifies whether users connecting to this server can drive shared applications
9009	Is File Transfer Enabled and max file size allowed?	Server	Value	Integer (4 bytes)	Set if the administrator allows File Transfer in the community. Value is the max allowed file size in K. 0 or a negative value means that the file size is unlimited.
9015	Is there a SIP Gateway in the community?	Server	Existential		Set by the SIP Gateway to let the clients know that user can add, be aware of, and interact with external SIMPLE users.

Instant Messaging Service

Table listing identifiers of instant messaging service.

IM Type	Used for	Data Type	Data Sub-Type	Data	Comments
1	Chat IM	1	0	Empty	Respond Start
			1	Empty	Respond Stop
25	Used as post type for invitations in Post Service.	N/A	N/A	N/A	N/A

Storage Service

Table listing identifiers of storage service.

Attribute Key ID	Value Types	Comments
0	String	The user's contact list. Contains a list of users and groups as they appear in Sametime Connect.
1	Boolean	Indicates whether to blink the Sametime Connect icon when a person becomes active.
2	Boolean	Indicates whether to play a sound when a person becomes active.
3	Boolean	Indicates whether the IM window should blink on a new incoming message.
4	Boolean	Indicates whether the IM window should come to front on a new incoming message.

5	Boolean	Indicates whether to display a sound for a new incoming message.
6	String	Default text for invite to chat.
7	Boolean	Indicates whether to show the away message for editing when the user selects to change his status to away. If true, a dialog will be opened prior to setting the user status.
8	Boolean	Indicates whether to show the DND message for editing when the user selects to change his status to DND. If true, a dialog will be opened prior to setting the user status.
10	Boolean	Indicates whether the contact list in Sametime Connect is displayed in show all mode (true) or show online only mode (false). In show online only mode, users that are currently offline are not displayed in the view.
11	Boolean	Indicates whether to show the active message for editing when the user selects to change his status to active. If true a dialog will be opened prior to setting the user status.
12	Boolean	Indicates whether to display an informational dialog box after adding a new user/group to the contact list.
13	Boolean	Indicates whether to ask the user before sending unsecured messages.
14	String	Default text for invite to a meeting.
15	Boolean	Indicates whether the contact list view in connect is displayed sorted in alphabetical order (true) or at its original order (false).
16	Boolean	Indicates whether the names in the contact list are displayed fully (false) or truncated to short names (true) using a specified delimiter.
17	Boolean	Indicates whether to start secure messages or not.
19	Boolean	Indicates whether to show the “Invitation has been sent” dialog when invite other people to join to a meeting.
30	Boolean	Indicates whether the search for users in the community is in exhaustive mode (true) on not (false).
31	Boolean	Indicates whether to show a dialog when a user who is already in the contact list is added a second time. True for displaying the dialog, false for not showing the dialog.
70	String	The sound file path to play when a person become active.
71	String	The sound file path to play when an invitation or new message arrives.
80	String	A list of the user’s away messages. The list contains the last five messages that were used by the user. The messages are separated using a semicolon. The first message in the list is the default message.
90	String	A list of the user’s DND messages. The list contains the last five messages that were used by the user. The messages are separated using a semicolon. The first message in the list is the default message.
100	String	A list of the user’s active messages. The list contains the last five messages that were used by the user. The messages are separated using a semicolon. The first message in the list is the default message.
110	Integer	Indicates the number of minutes to wait before changing automatically, the user status from Active to Away.
111	Boolean	Indicates whether to automatically change the user status from Active

		to Away when the user is not using the mouse or keyboard for a while.
112	Boolean	Indicates whether to change the user status from Away to Active when the user uses the mouse or keyboard.
8193	String	The XML version of the user's contact list. Contains a list of users and groups as they appear in Sametime Connect. Sametime Connect began using this new XML format in Sametime 7.5.
8194	String	XML containing all alerts the user has set. NOTE: This is for internal Sametime use only.
8195	String	XML containing all auto status settings the user has set. NOTE: This is for internal Sametime use only.

User In Place Attributes

Table listing identifiers of user in place attributes.

Attribute Key	Comments
8	Used for start/stop responding

Place Activity Types

Table listing identifiers of place activity types.

Attribute ID	Name
37121	Whiteboard
37122	Appshare
37123	Audio
37124	Video
37125	Reserved
37126	Chat
37127	Web Collaboration
37128	Reserved
37129	URL Push
37130	Question and Answer
37131	Shared Objects
37132	Moderation (user roles)

Post Service

Table listing identifiers of post service.

Post Type ID	Name
25	Invitations for meeting and n-way chat

Appendix D. Representing External Users

Overview

An external user is a user from another community.

Using the Java Toolkit, you can get these services involving an external user:

- Awareness services – You can add an external user to your WatchList and get updates on his status.
- Privacy services – You can add an external user to your privacy list.
- Instant messaging service – You can send and receive an instant message.

The STUser object is used for representing external users in the API. Certain functionality in the STUser definition handles external users:

- boolean isExternalUser()
  This method indicates whether or not this is an external user.
- void setExternalUser()
  This method is used to designate this user as an external user.

When to use the isExternalUser() method

If you are getting an ImReceived notification from the API and you want to know if the user who started the IM with you is from another community, you should run the following code:

```
/**
 * Adds a new External User to the buddyList
 * @param String externalId
 */
void addExternalUser(String externalId)
{
STId stId = new STId(externalId,"");
STUser user = new STUser(stId,externalId,"");
user.setExternalUser();
m_whatList.addItem(user);
}
```

When to use the setExternalUser() method

Whenever you create an STUser object for passing in the API and you need to indicate that this user is an External user, you should use the method.

For example, if you would like to add an external user to your watch list, you need to call the method WatchList.addItem(STObject stObject). The object should be an STUser.

You need to create the STUser with the user ID and name that you have. (Remember that there is no resolving service for an external user; so you should know an external user's ID in advance.)

Then you should call the method STUser.setExternalUser(). This tells the Awareness Service that it is an external user, and you will get his status. If you do not call this method, the Awareness Service will think that it is a Sametime internal user, and his status will always be offline.

Appendix E. Sametime Connectivity

Overview

In general, the two ways to connect to the Sametime server are:

- **Sametime Protocol connection** – The client connects by opening a TCP/IP socket to the server. Different kinds of proxies (socks4, socks5, and https) are supported using this method.

- **HTTP Tunneling connection** – The client opens what seems like an ordinary HTTP connection, but “tunnels” the Sametime protocol inside the HTTP protocol. This type of connection enables connecting through firewalls that only allow HTTP connections.

HTTP tunneling connectivity must address limits imposed by the browser on the number of connections that a Java applet can open using URLConnection. When using its native JVM, Internet Explorer limits the number to two connections per Internet Explorer process. In many situations, Sametime connectivity demands will quickly exceed these browser-imposed limits. The Sun Java Plugin does not limit the number of connections.

To address the Internet Explorer connection limitation, Sametime uses a tunneling mechanism called *hybrid polling*, which requires only a single HTTP connection. Hybrid polling is provided by the HybridPollingConnection class.

Sametime also supports a last-resort tunneling mechanism called full polling, in which HTTP connections are opened and closed intermittently. It is also available as a solution for unsigned applets that cannot use the Hybrid Polling Connection's 'use WinInet' option, because the framework requires access to local machine resources (specifically to a native DLL). However, full polling creates a heavy load on the server and degrades the user’s experience.

The HybridPollingConnection class

As its name implies, the HybridPollingConnection class implements the hybrid polling strategy described above. The Hybrid PollingConnection class can be used in two modes – it can either use the standard Java URLConnection class to perform the connection, or use a DLL to connect through Window's WinInet library. The second option is only available on Windows platform; to use it, you must install the file sturlcon10.dll. Code running on Internet Explorer with a native JVM must use this option, to bypass the two-connection limit of the browser.

Using the WinInet option may sometimes be advisable even when using the Java Plugin, because it makes the HybridPollingConnection's behavior a bit more like that of Internet Explorer. (For example, it makes the connection use the browser's certificate store instead of the Plugin's store for authentication.) The other mode, the URLConnection mode, also has its advantages: It is pure Java, it does not require DLL installation, and it can be used by unsigned applets, since it does not require access to native code. It is up to the developer to decide which connection mode to use.

Note also that code that uses WinInet on Windows will still run on non-Windows platforms (though it will use the Java URLConnection class instead). Deciding to use the WinInet option on Windows does not mean restricting your code to that platform.

The connectivity agent framework

To address Internet Explorer connectivity limitations, Sametime 3.x introduced a connectivity agent framework that multiplexed multiple Sametime connections over a single HTTP connection. The connectivity features in the current version of Sametime make this solution unnecessary and therefore this framework has been removed from the Java Toolkit starting with Sametime 8.0. Previous versions of the Java Toolkit can be acquired if you require the connectivity agent framework, or reference documentation about its usage. Previous versions can be downloaded from the IBM developerWorks Lotus downloads page:

http://www-128.ibm.com/developerworks/lotus/downloads/toolkits.html

Navigate to the IBM Sametime section and click on either the Sametime Software Development Kit (SDK) link for 7.5+ toolkits, or the Sametime Java toolkit link for 7.0 toolkits and prior.

Appendix F. Core Types

Overview

This section describes the core types used in the Java Toolkit. Each core type represents a Sametime entity that is used and shared by several or all of the Sametime services provided in the toolkit.

You might never work directly with the Sametime core types. However, if you debug your programs and inspect your Sametime objects, you will encounter core type objects.

Sametime ID Types

The Sametime identification classes are mainly for internal use; they are not exposed at the API level. The Sametime API uses more high-level entity definitions that encapsulate the Sametime ID.

STId

STId represents a Sametime entity identification. Every Sametime entity has a unique persistent ID within the Sametime community. The format and content of the ID can vary between communities. It is dependent on the company's directory that is integrated with Sametime (Domino or LDAP).

STLoginId

STLoginId represents a Sametime entity's runtime session identification. While every Sametime entity has its STId, some entities will also have session IDs assigned to them during runtime.

For example, a user has his own fixed ID, such as John Smith/IBM. When the user logs in to Sametime, an STLoginId will be assigned to him. (Multiple concurrent logins will result in different login IDs.) This information can be used to access a specific login of the user. It is valid only for the duration of the login.

Please see the *Sametime Java Toolkit Tutorial* or the *Sametime C++ Toolkit Tutorial* for more information on the Sametime user model.

Sametime Object Types

STUser

STUser represents a Sametime user in the community. Since Sametime provides user-to-user interaction, this structure is the most commonly used one.

In order to interact with other users in the community, the application needs to obtain a valid STUser object in one of two ways:

- Start with a user name and resolve it to the matching STUser object, using the Lookup Service to find the object.

- Create an STUser object from scratch.

  This option is advanced and requires knowledge of the specific ID allocated for the user. This information can vary between different Sametime installation environments, and therefore should only be used by advanced developers who have intimate knowledge of the directory used by Sametime.

Once the STUser object is available, the application can perform user-to-user interactions such as watching the user's online status and sending instant messages to the user.

STServer

STServer represents a Sametime server. Once a user logs in to the community, he is connected to a specific Sametime server.

Using the Java toolkit, you can get the STServer object once you are logged in to the server. To do this, call the Login.GetServer() function.

STUserInstance

STUserInstance represents a single login of a Sametime user. Once logged on to Sametime, a user is assigned an STLoginId that is valid for the duration of the login.

The STLoginId is contained by the STUserInstance object. An application can choose to use this information to contact a specific login of the user, instead of relying on the server to choose the appropriate login. However, we recommend that the developer not use STLoginId or rely on it, because it is transient. By contrast, the STUser object holds persistent information.

Appendix G. Multiple Language Support

By default, Sametime Java Toolkit UI components use the system-specified encoding to display standard text. This can mean that a Java Toolkit application displays its text in a different language from that used for UI text. For example, a custom window title might appear in German, while standard field labels and help text appear in English.

To make the UI components use the same language encoding as the program, modify the program as follows:

1. Add the following import statement:

   ```
   import com.lotus.sametime.resourceloader.*;
   ```

2. Get a reference to the resource loader service:

   ```
   ResourceLoaderService resourceSvc = (ResourceLoaderService)
   m_session.getCompApi(ResourceLoaderService.COMP_NAME);
   ```

3. Set the locale used by the resource loader. For example:

   ```
   resourceSvc.setLocale(Locale.GERMANY);
   ```

Appendix H. Signing Applets

When you use the Sametime Java toolkit to create an applet, you must sign the applet in certain circumstances.

As with any applet, a Sametime applet must be signed to allow out of sandbox operations like clipboard operations, connections to servers other than the one from which the applet is downloaded, access to local file systems, and so on. However, you must also sign the applet in the following Sametime-specific situations:

- If your applet provides support for file transfer operations. For more information about adding file transfer support, see the descriptions of the FileTransfer UI component in Chapter 3 and the FileTransfer Service in Chapter 4. You may also want to read about the AwarenessList AWT component in Appendix I. The AwarenessList component lets users initiate file transfers to online names.

- If you use the connectivity agent framework. For more information about the framework and when to use it, see Appendix G.

To view an IBM Technote about signing the Sametime Java applets, visit http://www.ibm.com/support/ and search for “1201620”.

Appendix I. Deprecated AWT and Community UI Components

Previous versions of the Sametime Java Toolkit shipped with components that provided Sametime-specific UI behavior by extending Java AWT components. These Sametime-enabled AWT components have been deprecated starting with Sametime 7.5. Support for these will continue for two major releases at which point the components will be removed. User Interface technologies are continually evolving – the core Java Toolkit components can be used to achieve Sametime-enabled user interfaces on all of them including AWT, Swing, SWT, etc., so focus has been moved to the core components themselves (which the deprecated AWT components were built on). For more information on the core components, see the Community Services section above.

Introduction

These components extend a Java AWT component and provide Sametime-specific UI behavior. They can be created and destroyed at any time independent of the Sametime session object lifecycle. The components that inherit from the Java AWT `Component` class can be embedded like any other component inside any AWT `Container`.

The available Sametime-Enabled AWT components are:

- Add Dialog
- Awareness List
- Directory Panel
- Place Awareness List
- Privacy Panel
- Resolve Panel

The available Community UI Components are:

- Announcement UI Component
- Chat UI Component
- Community UI Component
- FileTransfer UI Component

The Java Toolkit supports keystroke access to Community UI components. For details, see the "Mnemonics" section of the properties file for each component (for example, chatui.properties in CommRes.jar).

Add Dialog

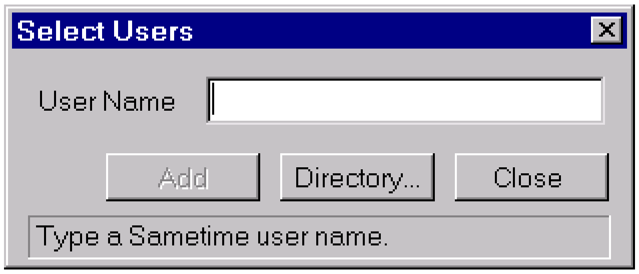
Overview

The Add dialog component provides the ability to retrieve user and group details.

When opened, the dialog window is in search mode, expecting a user name to be typed in. The window is extended to display a list of matches in the event that more than one match is found for a user name. The user can also browse the Sametime directory by clicking the "Directory" button, which uses a directory-browsing Add dialog instead of a search Add dialog.

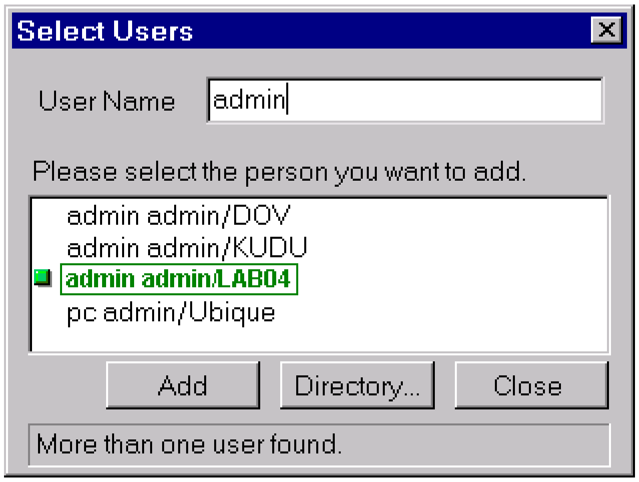
Description

When the Add dialog opens in search mode, it appears as follows:



Type a Sametime user name.

The dialog extends when more than one matching name is found (also known as resolve conflict). In the following dialog, more than one occurrence of "admin" was found:

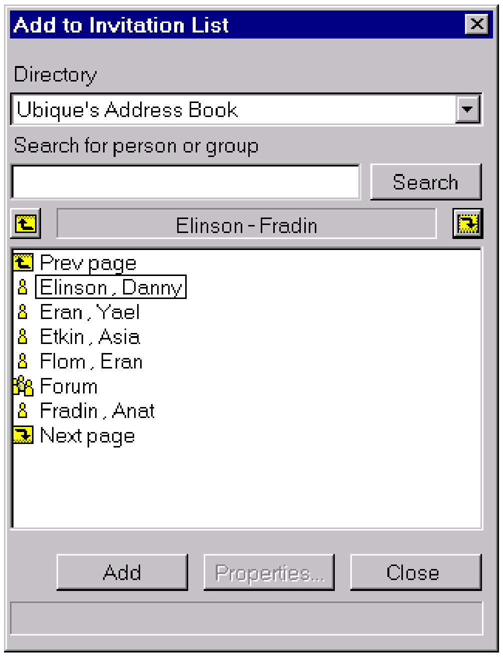


Please select the person you want to add.

admin admin/DOV
admin admin/KUDU
admin admin/LAB04
pc admin/Ubique

More than one user found.

Click the "Directory" button to launch the Directory dialog. The Add to Invitation List dialog window displays (as shown below). The list of people available depends on the chosen directory.



The Add dialog is a Sametime-enabled UI component that uses both the Lookup Service and the Directory Service when performing search and browse operations. To use the Add dialog, follow these steps:

1. Create and display the Add dialog. For example:

```
AddDialog addDialog =
   new AddDialog((Frame)getParent(),
                           m_session,
                           "Select Users");
addDialog.setVisible(true);
```

2. Listen to events that occur in the dialog by calling the `addResolveViewListener()` method and passing an object that implements `ResolveViewListener` or extends `ResolveViewAdapter`.

3. After the listener is registered, you can receive the following event notifications:

   - `Resolved()` – Notification that a unique user was found and selected by the end user. You can receive this event after a successful search (also known as "resolve") operation or after browsing the directory and selecting a user. You will also receive this event for each user inside a group if a group is selected when browsing the directory.

   - `ResolveFailed()` – Notification of a failed search operation. No match was found for the user name entered by the end user.

Package

`com.lotus.sametime.commui`

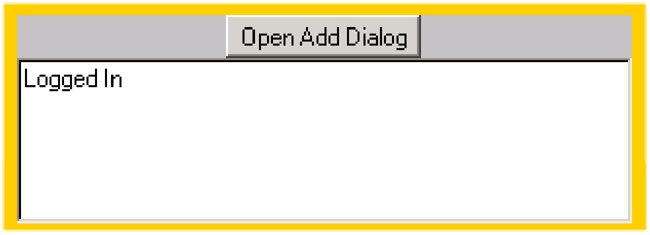
See Also

com.lotus.sametime.commui.CommUI in the *Sametime Java Toolkit Reference*.

The Resolve Panel section of this guide

Example

The following example contains an applet that logs on to the Sametime community. The applet contains a button that launches the Add dialog. Event notifications from the dialog are recorded in the text area displayed in the applet's area. You can run the Add Dialog sample from the Java Toolkit Samples page.



Logged In

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.commui.*;

/**
 * Add Dialog Sample.
 */
public class AddDialogApplet extends Applet
  implements LoginListener, ResolveViewListener, ActionListener
{
  /**
   * Our session.
   */
  private STSession m_session;

  /**
   * Text area for displaying messages.
   */
  private TextArea m_textArea;

  /**
   * String message displayed within the text area.
   */
  private StringBuffer m_msg = new StringBuffer();

  /**
   * The entry point for the applet.
   */
  public void init()
  {
    try
    {
      // generate a new session with a unique name
      m_session = new STSession("AddDialog Applet " + this);

      // Call the session to load all available components
      m_session.loadAllComponents();
      m_session.start();

      setLayout(new BorderLayout());
      Panel btnsPanel = new Panel(new GridBagLayout());

      Button openDialog = new Button("Open Add Dialog");
      openDialog.addActionListener(this);
      btnsPanel.add(openDialog);
      add(btnsPanel, BorderLayout.NORTH);

      m_textArea = new TextArea();
      add(m_textArea, BorderLayout.CENTER);

      // login to the community
      login();
    }
     catch(DuplicateObjectException e)
    {
      // This exception is thrown if an STSession with the
      // same name has already been created.
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {

    // get a reference to the community service. We use the session
    // object which contains a reference to all the components that
    // were loaded to get a reference to the community service.
    CommunityService comm = (CommunityService)
        m_session.getCompApi(CommunityService.COMP_NAME);

    // register a listener to the login/logout events.
    comm.addLoginListener(this);

    // login to the community
    comm.loginByPassword(getCodeBase().getHost(),
                         getParameter("loginName"),
                         getParameter("password"));
  }

  /**
   * Login event.
   */
  public void loggedIn(LoginEvent event)
  {
    m_msg.append("Logged In");
    m_textArea.setText(m_msg.toString());
  }

  /**
   * Logout event
   */
  public void loggedOut(LoginEvent event)
  {
    m_msg.append("Logged Out");
    m_textArea.setText(m_msg.toString());
  }

  /**
   * Notification of a successful resolve operation.
   */
  public void resolved(ResolveViewEvent event)
  {
    m_msg.append("**** Resolved ****\n");
    m_msg.append(event.getResolvedName() + " : "
               + event.getUser().getName() + "\n");
    m_textArea.setText(m_msg.toString());
  }

  /**
   * Notification of a failed resolve operation.
   */
  public void resolveFailed(ResolveViewEvent event)
  {
    m_msg.append("**** Resolved Failed****\n");
    m_msg.append(event.getResolvedName() + ", Reason:" +
                 Integer.toHexString(event.getReason()) + "\n");
    m_textArea.setText(m_msg.toString());
  }

  /**
   * Action Performed. The button was clicked open the dialog.
   */
  public void actionPerformed(ActionEvent e)
  {
    AddDialog d  = new AddDialog((Frame) getParent(),
                               m_session, "Select Users");
    d.addResolveViewListener(this);
    d.setVisible(true);
  }
}
```

Awareness List

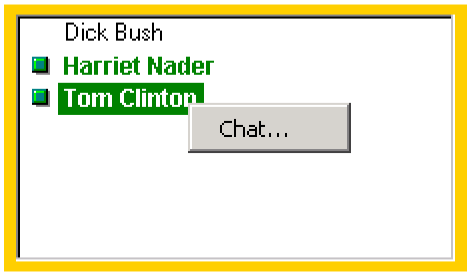
Overview

The Awareness List is an embeddable Sametime-enabled Abstract Windowing Toolkit (AWT) component that allows you to display a list of users in the Sametime community with their online status and attributes. The list reflects the user's current status. It is updated whenever a change in the user's status or attribute occurs.

Awareness Lists display the names of online users in green text. Instant messages and instant meetings can be started directly from the Awareness List by double-clicking on the online user's name, or by right-clicking on a selection of online user names and choosing any of the available meeting types. File transfers and announcements can also be initiated by right-clicking. However, note that if you want to support file transfer in an applet, you must sign the applet, to allow local file access. For more information, see "Restrictions" in Appendix B.

Description

An Awareness List with a list of three user names appears as follows:



The Awareness List extends java.awt.Panel and can therefore be embedded inside any Java AWT Container. Follow these steps to use the Awareness List:

1. Create the Awareness List:

```
    m_awarenessList = new AwarenessList(m_session,
                          AwarenessList.CHECKED_BORDER);
```

2. Add the list to any AWT Container:

```
  add(m_awarenessList, BorderLayout.CENTER);
```

3. Add users to be watched by calling the `addUser()` method:

```
  m_awarenessList.addUser(stUser);
```

   `The Awareness List also provides a simple API to remove a user from a list, get the currently selected items, or get all items in the list.`

4. `Listen to events that occur in the list by calling the addAwarenessViewListener() method and passing a class that implements AwarenessViewListener or extends AwarenessViewAdapter.`

5. After the listener is registered, you can receive the following event notifications:

   - **StatusChanged()** – Indicates a change in a user's status

- **SelectionChanged()** – Indicates that the user modified his selection in the list (mainly used for enabling/disabling menu items)
- **UsersAdded()** – Indicates that users were added to the list
- **UsersRemoved()** – Indicates that users were removed from the list
- **AddUserFailed()** – Indicates that a user failed to be added to the list
- **ServiceUnavailable()** – Indicates that the Awareness Service is currently unavailable (all users appear offline)
- **ServiceAvailable()** – Indicates that the Awareness Service is currently available

6. (Optional) Set right-click behavior of the list.

   By default, the lists' right-click menu allows you to initiate only one type of meeting (Chat) with one or more online users. You can modify this behavior to allow any kind of Sametime meeting by replacing the list's controller and implementing some listeners of the Chat UI component. For more information, see the Chat UI Component section in Chapter 3.

   To change the list's controller:

```
AVController avController = new
  AVController(m_awarenessList.getModel());
m_awarenessList.setController(avController);
```

Package

```
com.lotus.sametime.awarenessui.list
```

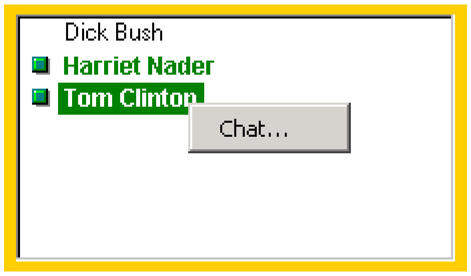
See Also

The Place Awareness List section of this guide

The Live Names and Extended Live Names samples (Chapters 5 and 6) in the *Sametime Java Toolkit Tutorial*

Example

The following example is the Live Names sample from the Sametime Java Tutorial that shows how to use the Awareness List. You can run the Live Names sample from the Java Toolkit Samples page.



```
import java.awt.*;
import java.applet.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.STUser;
import com.lotus.sametime.community.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.awarenessui.list.AwarenessList;

/**
 * Sample applet that displays a list of live user names.
 */
public class LiveNamesApplet extends Applet
    implements LoginListener, ResolveListener
{
  private STSession m_session;
  private CommunityService m_comm;
  private AwarenessList m_awarenessList;

  /**
   * Applet initalized. Create the session, load all components,
   * start the session and then login.
   */
  public void init()
  {
    try
    {
      m_session = new STSession("LiveNamesApplet " + this);
      m_session.loadAllComponents();
      m_session.start();

      setLayout(new BorderLayout());
      m_awarenessList = new AwarenessList(m_session, true);
      add(m_awarenessList, BorderLayout.CENTER);

      login();
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    m_comm = (CommunityService)
        m_session.getCompApi(CommunityService.COMP_NAME);
    m_comm.addLoginListener(this);
    m_comm.loginByPassword(getCodeBase().getHost(),
                           getParameter("loginName"),
                           getParameter("password"));
  }

  /**
   * Logged in event. Print logged in msg to console.
   * Resolve the users to add to the awareness list.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Logged In");

    LookupService lookup = (LookupService)
      m_session.getCompApi(LookupService.COMP_NAME);

    Resolver resolver =
      lookup.createResolver(false,  // Return all matches.
                            false,  // Non-exhaustive lookup.
                            true,   // Return resolved users.
                            false); // Do not return resolved groups.

    resolver.addResolveListener(this);

    String[] userNames = getUserNames();
    resolver.resolve(userNames);
  }

  /**
   * Helper method to read a list of user names from the html
   * parameter 'watchedNames'.
   */
  String[] getUserNames()
  {
    String users = getParameter("watchedNames");

    StringTokenizer tokenizer = new StringTokenizer(users, ",");
    String[] userNames = new String[tokenizer.countTokens()];

    int i = 0;
    while(tokenizer.hasMoreTokens())
    {
      userNames[i++] = tokenizer.nextToken();
    }
    return userNames;
  }

  /**
   * Users resolved succesfuly event. An event will be generated for
   * each resolved user. Add the resolved user to the awareness list.
   */
  public void resolved(ResolveEvent event)
  {
    m_awarenessList.addUser((STUser) event.getResolved());
  }

  /**
   * Handle a resolve conflict event. Will be received in the case
   * that more then one match was found for a specified user name.
   * Add the users to the awareness list anyway.
   */
  public void resolveConflict(ResolveEvent event)
  {
    STUser[] users = (STUser[]) event.getResolvedList();
    m_awarenessList.addUsers(users);
  }

  /**
   * Resolve failed. No users are available to add to the list.
   */
  public void resolveFailed(ResolveEvent event)
  {
  }
  /**
   * Logged out event. Print logged out msg to console. Leave default
   * behavior which will display a dialog box.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");
  }

  /**
```

```
     * Applet destroyed. Logout, stop and unload the session.
     */
    public void destroy()
    {
      m_comm.logout();
      m_session.stop();
      m_session.unloadSession();
    }
  }
```

Directory Panel

Overview

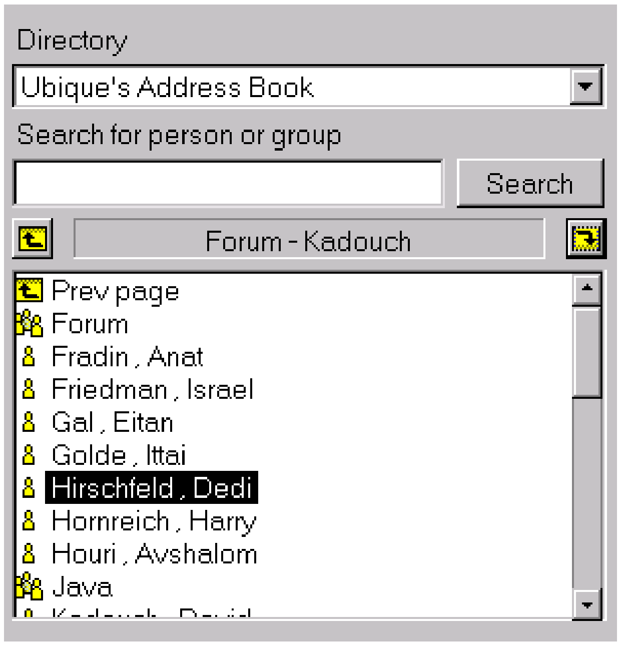
The Directory Panel is an embeddable Sametime-enabled AWT component that allows you to display the list of available directories in the Sametime community and the directory contents. This panel enables basic directory browse and search capabilities for each of the available directories. An applet or application that needs to integrate directory access capabilities can quickly embed this panel and receive notifications on selections of users and groups within the directory. Information obtained from the directory can be used to interact with other users within the community.

The Directory browsing service provides a list of all available directories and allows you to query for chunks of entries in each directory. Depending on your Sametime server configuration, directory browsing might not be available. See the Directory Service section in this manual for instructions on how to determine whether directory browsing is available on your Sametime server.

A simple Directory dialog containing the Directory Panel is also provided in the toolkit. See the *Sametime Java Toolkit Reference* for more information on this dialog.

Description

The Directory Panel extends java.awt.Panel and can therefore be embedded inside any Java AWT Container. While the user is logged on to the system, the panel will display the available directories and enable the user to browse and search within each directory.



Follow these steps to use the Directory Panel:

1. Create the Directory Panel:

   ```
   DirectoryPanel dirPanel = new DirectoryPanel(m_session);
   ```

2. Add the panel to any AWT Container:

   ```
   add(dirPanel, BorderLayout.CENTER);
   ```

3. Listen to Directory Panel events by calling the addDirectoryListViewListener() method and passing a class that implements the DirectoryListViewListener or extends DirectoryListViewAdapter.

4. After the listener is registered, you can receive the following event notifications:

   - **NodeDoubleClicked()** – Indicates that the user double-clicked a node in the list. A node is either a user or group in the Directory list.

   - **SelectionChanged()** – Indicates that the user modified his selection in the list. You can use the Directory Panel method getSelectedEntries() to get a list of the currently selected entries at any time.

Package

```
com.lotus.sametime.directoryui
```

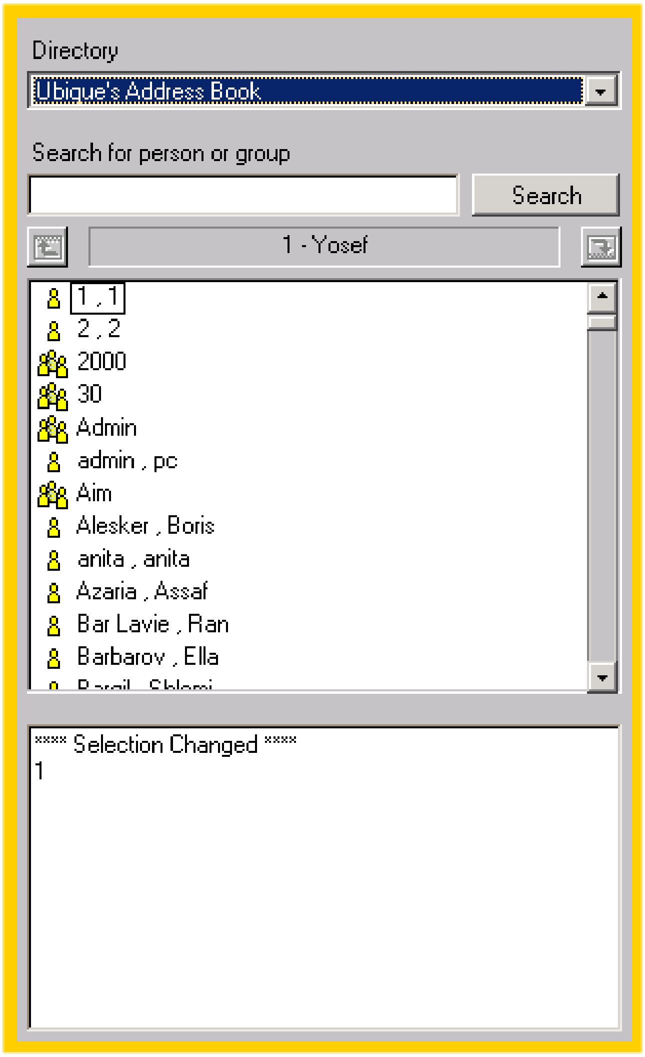
See Also

com.lotus.sametime.directoryui.DirectoryDialog of the *Sametime Java Toolkit Reference*

The Directory Service section of this guide

Example

The following example contains an applet that logs on to the Sametime community and displays the Directory Panel within the applet's area. Any change in selection or a double click is recorded in the text area displayed below the panel. You can run the Directory Panel sample from the Java Toolkit Samples page.



```
**** Selection Changed ****
1
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.directoryui.*;

/**
 * Directory Sample using the directory panel.
 */
public class Directory extends Applet
  implements LoginListener, DirectoryListViewListener
{
  /**
   * Our session.
   */
   private STSession m_session;

  /**
   * Text area for displaying messages.
   */
  private TextArea m_textArea;
  /**
   * The entry point for the applet.
   */
  public void init()
  {
    try
    {
      // generate a new session with a unique name
      m_session = new STSession("Directory Applet " + this);

      // Call the session to load all available components
      m_session.loadAllComponents();
      m_session.start();

      setLayout(new BorderLayout());
      DirectoryPanel dirPanel = new DirectoryPanel(m_session);
      add(dirPanel, BorderLayout.CENTER);

      //register a listener to receive selection change and
      //double click events.
      dirPanel.addDirectoryListViewListener(this);

      m_textArea = new TextArea();
      add(m_textArea, BorderLayout.SOUTH);

      // login to the community
      login();
    }
    catch(DuplicateObjectException e)
    {
      // This exception is thrown if an STSession with the same
      // name has already been created.
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    // get a reference to the community service. We use the session
    // object which contains a reference to all the components that
    // were loaded to get a reference to the community service.
    CommunityService comm = (CommunityService)
                    m_session.getCompApi(CommunityService.COMP_NAME);

    // register a listener to the login/logout events.
    comm.addLoginListener(this);

    // login to the community
    comm.loginByPassword(getCodeBase().getHost(),
                         getParameter("loginName"),
                         getParameter("password"));
    // Wait for the loggedin() event to enter the place
  }

  /**
   * Login event.
   */
  public void loggedIn(LoginEvent event)
  {
    String msg  = "Logged In";
    m_textArea.setText(msg);
  }

  /**
   * Logout event
   */
  public void loggedOut(LoginEvent event)
  {
    String msg  = "Logged Out";
    m_textArea.setText(msg);
  }

  /**
   * Returns the applet's insets. Creates 5 pixel margin around
   * the applet.
   */
  public Insets getInsets()
  {
    return new Insets(5, 5, 5, 5);
  }

  /**
   * Selection changed in the directory list.
   */
  public void selectionChanged(DirectoryListViewEvent event)
  {
    String msg = "**** Selection Changed ****\n";

    Vector v = event.getSelectedNodes();
    Enumeration e = v.elements();
    while(e.hasMoreElements())
    {
      msg += ((STObject) e.nextElement()).getName() + "\n";
    }

    m_textArea.setText(msg);
  }

  /**
   * User or Group double clicked in the directory's list.
   */
  public void nodeDoubleClicked(DirectoryListViewEvent event)
  {
    String msg = "**** Node Double Clicked ****\n";
    msg += event.getDoubleClickedNode().getName();
    m_textArea.setText(msg);
  }

  /**
   * Applet destroyed. Logout of Sametime.
   */
  public void destroy()
  {
    CommunityService comm = (CommunityService)
           m_session.getCompApi(CommunityService.COMP_NAME);
    comm.logout();

    m_session.stop();
    m_session.unloadSession();
  }
}
```

Place Awareness List

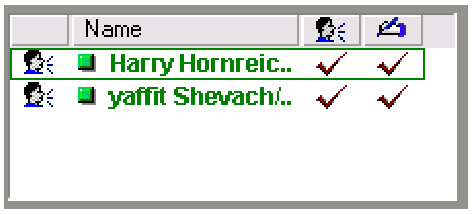
Overview

The Place Awareness List is an embeddable Sametime-enabled AWT component that displays the list of users in a specific place. The users are shown with their online status and place attributes. This list is updated whenever a change in a user's status or attribute occurs.

The Place Awareness List and the Awareness List provide similar functionality. The major difference between them is the source of the list of names.

Description

This screen capture of the Place Awareness List shows two users in a place:



	Name		
	Harry Hornreic..	✓	✓
	yaffit Shevach/..	✓	✓

The Place Awareness List extends java.awt.Panel and can therefore be embedded inside any Java AWT Container. Follow these steps to use the Place Awareness List:

1. Create the Place Awareness List:

```
m_placeAwarenessList =
     new PlaceAwarenessList(m_session,true);
```

2. Add the list to any AWT Container:

```
add(m_placeAwarenessList, BorderLayout.CENTER);
```

3. Bind the list to a specific place. The following example code creates a place object and then binds the Place Awareness List to it:

```
m_place = placesService.createPlace("Tom's Place",
                          "My Place",
                          "",
                          EncLevel.ENC_LEVEL_RC2_40,
                          0);
m_placeAwarenessList.bindPlace(m_place);
```

4. `Listen to events that occur in the list by calling the addAwarenessViewListener() method and passing a class that implements AwarenessViewListener or extends AwarenessViewAdapter.`

   After the listener is registered, you can receive the following event notifications.

   - **StatusChanged()** – Indicates a change in a user's status
   - **SelectionChanged()** – Indicates that the user modified his selection in the list (mainly used for enabling/disabling menu items)
   - **UsersAdded()** – Indicates that users were added to the list

- **UsersRemoved()** – Indicates that users were removed from the list
- **AddUserFailed()** – Indicates that a user failed to be added to the list
- **ServiceUnavailable()** – Indicates that the Awareness Service is currently unavailable (all users appear offline)
- **ServiceAvailable()** – Indicates that the Awareness Service is currently available

5. (Optional) Set right-click behavior of the list.

   By default, the list's right-click menu allows you to initiate only one type of meeting (Chat) with one or more online users. You can modify this behavior to initiate any kind of Sametime meeting by replacing the list's controller and implementing some listeners of the Chat UI component. See the Chat UI section in Chapter 3 for more details.

   To change the lists' controller:

```
AVController avController = new
  AVController(m_placeAwarenessList.getModel());
m_placeAwarenessList.setController(avController);
```

Package

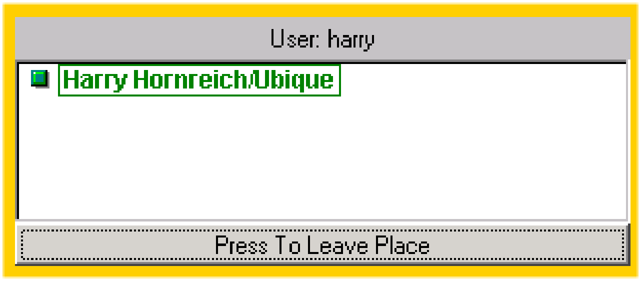
`com.lotus.sametime.awarenessui.placelist`

See Also

The Awareness List section in this guide

Example

The following example contains an applet that embeds a Place Awareness List. Click the "Press to Enter Place" button to enter the place. Select online users and right-click to start a meeting or see available meeting tools. You can run the Place Awareness List sample from the Java Toolkit Samples page.



```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.awarenessui.placelist.PlaceAwarenessList;
import com.lotus.sametime.awarenessui.av.AVController;
import com.lotus.sametime.community.*;
import com.lotus.sametime.places.*;
import com.lotus.sametime.core.constants.EncLevel;

/**
 * Place Awareness Sample using the Place Awareness List.
 */
public class PlaceAwareness extends Applet implements LoginListener,
                                                       ActionListener
{
  /**
   * Our session.
   */
  private STSession m_session;

  /**
   * The place awareness list.
   */
  private PlaceAwarenessList m_placeAwarenessList;

  /**
   * Enter/leave place button
   */
  private Button m_button;

        /**
   * The place being watched.
   */
  private Place m_place;


  /**
   * The entry point for the applet.
   */
  public void init()
  {
    try
    {
      // generate a new session with a unique name
      m_session = new STSession("Place Awareness List " + this);

      // Call the session to load all available components
      m_session.loadAllComponents();
      m_session.start();

      setLayout(new BorderLayout());

      //add a label with the user name at top of the list
      Label label = new Label("User: " + getParameter("loginName"));
      label.setAlignment(Label.CENTER);
      add(label, BorderLayout.NORTH);

      // create the new list view
      m_placeAwarenessList = new PlaceAwarenessList(m_session, true);
      add(m_placeAwarenessList, BorderLayout.CENTER);

      //add the enter/leave button at the bottom
      m_button = new Button("Press To Enter Place");
      add(m_button, BorderLayout.SOUTH);
      m_button.setEnabled(false);
      m_button.addActionListener(this);

      // The default right-click menu for the awareness list does not
      // give audio & video as options. We can change that by
      // changing the controller to a different one.
     AVController avController =
          new AVController(m_placeAwarenessList.getModel());
      m_placeAwarenessList.setController(avController);
      // login to the community
      login();
    }
    catch(DuplicateObjectException e)
    {
      // This exception is thrown if an STSession with the same
      // name has already been created.
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    // get a reference to the community service. We use the session
    // object which contains a reference to all the components that
    // were loaded to get a reference to the community service.
    CommunityService comm = (CommunityService)
              m_session.getCompApi(CommunityService.COMP_NAME);

    // register a listener to the login/logout events.
    comm.addLoginListener(this);

    // login to the community
    comm.loginByPassword(getCodeBase().getHost(),
                         getParameter("loginName"),
                         getParameter("password"));

    // Wait for the loggedin() event to enter the place
  }

  /**
   * Login event.
   */
  public void loggedIn(LoginEvent event)
  {
    //Get refernce to the place service
    PlacesService placesService = (PlacesService)
               m_session.getCompApi(PlacesService.COMP_NAME);

    //Create a new place object
    m_place = placesService.createPlace("Tom's Place",
                                        "My Place",
                                        EncLevel.ENC_LEVEL_RC2_40,
                                        0);

    //bind the list to the specified place.
    m_placeAwarenessList.bindPlace(m_place);

    //enable enter/leave button
    m_button.setEnabled(true);
  }

  /**
   * Logout event
   */
  public void loggedOut(LoginEvent event)
  {
    //disable enter/leave button
    m_button.setEnabled(false);
  }
```

```
    /**
     * Action event. Leave/Enter button clicked enter or leave
     * place accordingly.
     */
    public void actionPerformed(ActionEvent evt)
    {
      if(evt.getActionCommand().equals("Leave"))
      {

        //enter the place
        m_place.leave(0);
        m_button.setActionCommand("Enter");
        m_button.setLabel("Press To Enter Place");
      }
      else
      {
        //enter the place
        m_place.enter();
        m_button.setActionCommand("Leave");
        m_button.setLabel("Press To Leave Place");
      }
    }

    /**
     * Applet destroyed. Logout of Sametime.
     */
    public void destroy()
    {
      CommunityService comm = (CommunityService)
            m_session.getCompApi(CommunityService.COMP_NAME);
      comm.logout();

      m_session.stop();
      m_session.unloadSession();
    }
  }
```

Privacy Panel

Overview

The Privacy Panel list is an embeddable Sametime-enabled AWT component that allows a user to view and modify his privacy settings. Privacy settings determine a user's online status visibility within the Sametime community.

The user can select one of three options:

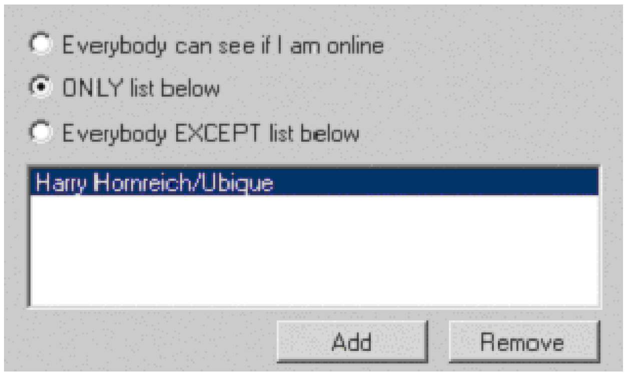
- **Everybody can see me** – The user is visible to all users in the community.
- **Only list below** – Only the specified users in the list will receive online notifications about the user.
- **Everybody except list below** – The user is visible to all users except for the users specified in the list.

Remember that Sametime privacy is symmetric; you cannot see other users if you choose not to let them see you.

A simple Privacy dialog containing the Privacy Panel is also provided in the toolkit. See the *Sametime Java Toolkit Reference* for more information on this dialog.

Description

In the screen capture of the Privacy Panel shown below, only one person will be able to see when the user is online:


Everybody can see if I am online [✓ ONLY list below] Everybody EXCEPT list below

Harry Hornreich/Ubique

The Privacy Panel extends java.awt.Panel and can therefore be embedded inside any Java AWT Container. Follow these steps to use the Privacy Panel:

1. Create the Privacy Panel:

   ```
   PrivacyPanel privacyPanel = new PrivacyPanel(m_session)
   ```

2. Add the panel to any AWT Container:

   ```
   add(privacyPanel, BorderLayout.CENTER);
   ```

3. The Privacy Panel has no event listeners to listen to.

4. The panel allows modifications of the user privacy settings. Call the `submit()` method of the panel to submit the end user's changes.

Package

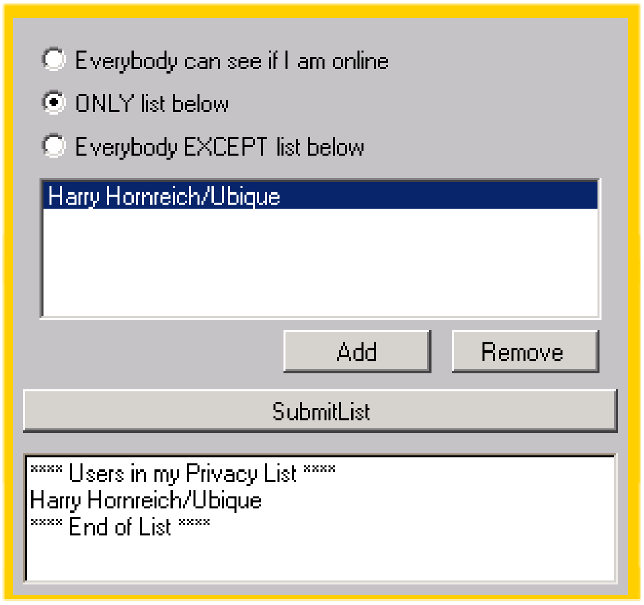
`com.lotus.sametime.commui`

See Also

com.lotus.sametime.commui.PrivacyDialog of the *Sametime Java Toolkit Reference*

Example

This example code contains an applet that logs on to the Sametime community and displays the Privacy Panel within the applet's area. Any change in the privacy settings is recorded in the text area displayed below the panel. You can run the Privacy Panel sample from the Java Toolkit Samples page.



```
**** Users in my Privacy List ****
Harry Hornreich/Ubique
**** End of List ****
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.commui.*;

/**
 * Privacy Sample using the privacy panel.
 */
public class PrivacyApplet extends Applet
  implements LoginListener, ActionListener, MyPrivacyListener
{
  /**
   * Our session.
   */
  private STSession m_session;

  /**
   * Text area for displaying messages.
```

```
     */
    private TextArea m_textArea;


    /**
     * The Privacy Panel.
     */
    private PrivacyPanel m_privacyPanel;
          /**
     * The entry point for the applet.
     */
    public void init()
    {
      try
      {
        // generate a new session with a unique name
        m_session = new STSession("Privacy Applet " + this);

        // Call the session to load all available components
        m_session.loadAllComponents();
        m_session.start();

        //Create the UI components which include the privacy panel,
        //submit button and the text area for displaying privacy
        //information.
        setLayout(new BorderLayout(0, 10));
        Panel upperPanel = new Panel(new BorderLayout());
        m_privacyPanel = new PrivacyPanel(m_session);
        upperPanel.add(m_privacyPanel, BorderLayout.CENTER);

        Panel btnsPanel = new Panel(new BorderLayout(0, 10));
        Button submitList = new Button("SubmitList");
        submitList.addActionListener(this);

        btnsPanel.add(submitList, BorderLayout.CENTER);
        upperPanel.add(btnsPanel, BorderLayout.SOUTH);
        add(upperPanel, BorderLayout.CENTER);

        m_textArea = new TextArea();
        add(m_textArea, BorderLayout.SOUTH);

        // login to the community
        login();
      }
      catch(DuplicateObjectException e)
      {
        // This exception is thrown if an STSession with the same
        // name has already been created.
        e.printStackTrace();
      }
    }

    /**
     * Login to the community using the user name and password
     * parameters from the html.
     */
    private void login()
    {
      // get a reference to the community service. We use the session
      // object which contains a reference to all the components that
      // were loaded to get a reference to the community service.
      CommunityService comm = (CommunityService)
                m_session.getCompApi(CommunityService.COMP_NAME);

      // register a listener to the login/logout events.
      comm.addLoginListener(this);
      // login to the community
      comm.loginByPassword(getCodeBase().getHost(),
                            getParameter("loginName"),
                            getParameter("password"));
    }

    /**
     * Login event.
     */
    public void loggedIn(LoginEvent event)
    {
      // register a listener to privacy change evnets.
      event.getLogin().addMyPrivacyListener(this);

      String msg  = "Logged In";
      m_textArea.setText(msg);
    }

    /**
     * Logout event
     */
    public void loggedOut(LoginEvent event)
    {
      String msg  = "Logged Out";
      m_textArea.setText(msg);
    }

    /**
     * Action Performed. The button was clicked open the dialog.
     */
    public void actionPerformed(ActionEvent p1)
    {
      m_privacyPanel.submit();
    }

    /**
     * Returns the applet's insets. Creates 5 pixel margin around
     * the applet.
     */
    public Insets getInsets()
    {
      return new Insets(5, 5, 5, 5);
    }

    /**
     * My privacy settings changed.
     */
    public void myPrivacyChanged(MyPrivacyEvent event)
    {
      String msg  = "**** Users in my Privacy List ****\n";
      Enumeration e =  event.getPrivacyList().elements();

      while(e.hasMoreElements())
      {
        msg += ((STUser) e.nextElement()).getName() + "\n";
      }

      msg += "**** End of List ****\n";

      m_textArea.setText(msg);
    }

    /**
     * Change to my privacy settings were denied.
     */
    public void changeMyPrivacyDenied(MyPrivacyEvent event)
    {
      String msg  = "**** Privacy Settings Denied ****\n";
      m_textArea.setText(msg);
    }

    /**
     * Applet destroyed. Logout of Sametime.
     */
    public void destroy()
    {
      CommunityService comm = (CommunityService)
               m_session.getCompApi(CommunityService.COMP_NAME);
      comm.logout();

      m_session.stop();
      m_session.unloadSession();
    }
  }
```

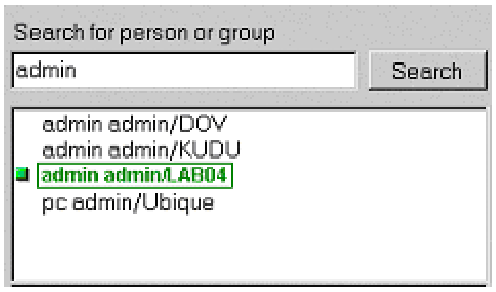
Resolve Panel

Overview

The Resolve Panel is an embeddable Sametime-enabled AWT component that provides a text field for a user to enter a name to resolve and performs the resolve operation. The result is returned by a listener to the embedding application. If more than one match is found, a list is displayed from which the user can select the appropriate name. If no matches are found, an error message is displayed.

Description

Following is a screen capture of the Resolve Panel:



The top part of the panel contains the text field for the user name to be searched and the “Search” button. The lower part contains an Awareness List that is populated with matching user names when more than a single match is found for the searched name.

The Resolve Panel extends java.awt.Panel and can therefore be embedded inside any Java AWT Container. Follow these steps to use the Resolve Panel:

1. Create the Resolve Panel:
```
ResolvePanel resolvePanel = new
          ResolvePanel(m_session);
```
2. Add the panel to any AWT Container:
```
add(resolvePanel, BorderLayout.CENTER);
```
3. Listen to events that occur in the panel by calling the addResolveViewListener() method and passing a call that implements ResolveViewListener or extends ResolveViewAdapter.

4. After the listener is registered, you can receive the following event notifications:

- **`Resolved()`** – Notification of a successful resolve operation.
- **ResolveFailed()** – Notification of a failed resolve operation.

Package

`com.lotus.sametime.commui`

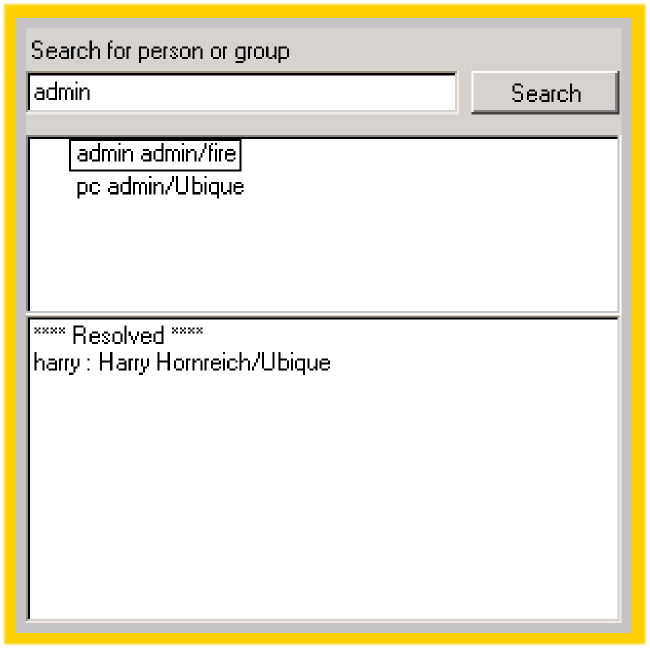
See Also

com.lotus.sametime.commui .CommUI of the *Sametime Java Toolkit Reference*

The Add Dialog section in this guide

Example

This example contains an applet that logs on to the Sametime community and displays the Resolve Panel within the applet's area. Event notifications concerning the success and failure of the resolve operation are recorded in the text area displayed below the panel. You can run the Resolve Panel sample from the Java Toolkit Samples page.



```
admin admin/fire
pc admin/Ubique
```

```
**** Resolved ****
harry : Harry Hornreich/Ubique
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.commui.*;

/**
 * Resolve Sample using the resolve panel.
 */
public class ResolveApplet extends Applet
  implements LoginListener, ResolveViewListener
{
  /**
   * Our session.
   */
  private STSession m_session;

  /**
   * Text area for displaying messages.
   */
  private TextArea m_textArea;

  /**
   * The entry point for the applet.
```

```
     */
    public void init()
    {
      try
      {
        // generate a new session with a unique name
        m_session = new STSession("Resolve Applet " + this);

        // Call the session to load all available components
        m_session.loadAllComponents();
        m_session.start();

        setLayout(new BorderLayout());
        ResolvePanel resolvePanel = new ResolvePanel(m_session);
        add(resolvePanel, BorderLayout.CENTER);

        //register a listener to receive notifications about user
        //resolved events.
        resolvePanel.addResolveViewListener(this);

        m_textArea = new TextArea();
        add(m_textArea, BorderLayout.SOUTH);

        // login to the community
        login();
      }
      catch(DuplicateObjectException e)
      {
        // This exception is thrown if an STSession with the same
        // name has already been created.
        e.printStackTrace();
      }
    }

    /**
     * Login to the community using the user name and password
     * parameters from the html.
     */
    private void login()
    {
      // get a reference to the community service. We use the session
      // object which contains a reference to all the components that
      // were loaded to get a reference to the community service.
      CommunityService comm = (CommunityService)
                   m_session.getCompApi(CommunityService.COMP_NAME);

      // register a listener to the login/logout events.
      comm.addLoginListener(this);

      // login to the community
      comm.loginByPassword(getCodeBase().getHost(),
                           getParameter("loginName"),
                           getParameter("password"));
    }

    /**
     * Login event.
     */
    public void loggedIn(LoginEvent event)
    {
      String msg  = "Logged In";
      m_textArea.setText(msg);
    }

    /**
     * Logout event
     */
    public void loggedOut(LoginEvent event)
    {
      String msg  = "Logged Out";
      m_textArea.setText(msg);
    }

    /**
     * Returns the applet's insets. Creates 5 pixel margin around
     * the applet.
     */
    public Insets getInsets()
    {
      return new Insets(5, 5, 5, 5);
    }

    /**
     * Notification of a successful resolve operation.
     */
    public void resolved(ResolveViewEvent event)
    {
      String msg = "**** Resolved ****\n";
      msg += event.getResolvedName() + " : " +
             event.getUser().getName();
      m_textArea.setText(msg);
    }

    /**
     * Notification of a failed resolve operation.
     */
    public void resolveFailed(ResolveViewEvent event)
    {
      String msg = "**** Resolved Failed****\n";
      msg += event.getResolvedName() + ", Reason:" +
             Integer.toHexString(event.getReason());
      m_textArea.setText(msg);
    }

    /**
     * Applet destroyed. Logout of Sametime.
     */
    public void destroy()
    {
      CommunityService comm = (CommunityService)
               m_session.getCompApi(CommunityService.COMP_NAME);
      comm.logout();

      m_session.stop();
      m_session.unloadSession();
    }

}
```

Announcement UI Component

Overview

The Announcement UI component provides the ability to send and receive announcements without having to deal with the low-level protocols involved. The component includes the SendAnnouncement dialog used for writing and sending an announcement, and the ReceiveAnnouncement dialog for receiving an announcement.

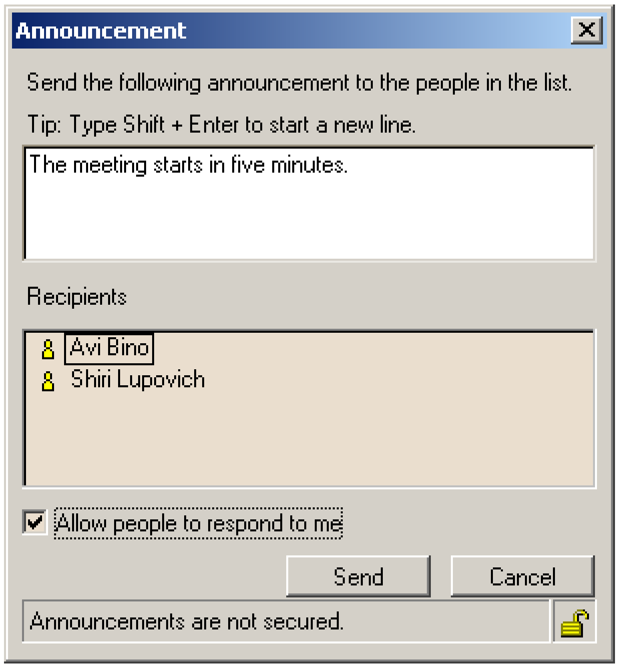
Description

You must load the Announcement UI component into the Sametime session object, like any other toolkit component. The Announcement UI component listens to events from the Announcement Service and notifies the user of incoming announcements. It can also be used to easily send announcements.

Sending Announcements

To send an announcement, call the sendAnnouncement() method of the Announcement Service. This method takes a list of STObject objects as a parameter. Each object in the list can be an STUser or an STGroup (a public group). The announcement will be sent to all users and public groups in the list.

A SendAnnouncement dialog opens, letting the user write his announcement.


Announcement
Send the following announcement to the people in the list.
Tip: Type Shift + Enter to start a new line.
Recipients
Allow people to respond to me
Send
Cancel

The meeting starts in five minutes.

Avi Bino
Shiri Lupovich

☑ Allow people to respond to me

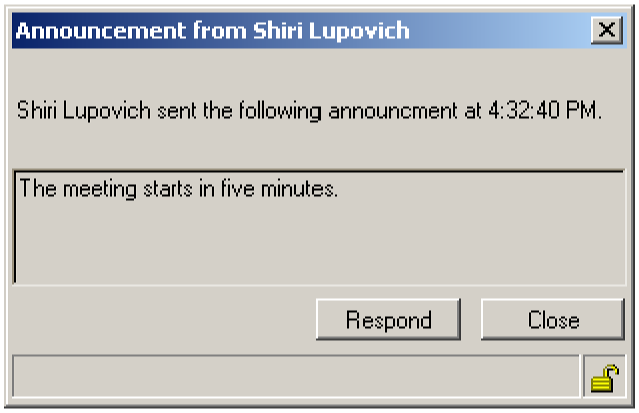
Announcements are not secured.

The user can decide whether to let the recipients respond to the announcement by using the “Allow people to respond to me” checkbox.

The user clicks Send to send the announcement.

Receiving Announcements

To receive announcements, load the Announcement UI component into the Sametime session object. The Announcement UI component listens to events from the Announcement Service, and opens a ReceiveAnnouncement dialog as soon as an announcement is received.



Shiri Lupovich sent the following announcment at 4:32:40 PM.

The meeting starts in five minutes.

If the sender allowed responses, a Respond button appears. This button, when clicked, initiates an IM message to the sender.

Package

com.lotus.sametime.announcementui

See Also

The Announcement Service section of this guide

Chat UI Component

Overview

The Chat UI component provides the ability to create Sametime meetings and participate in them without having to deal with the low-level protocols that are involved. It includes the Invite dialog used for inviting users to meetings, the Join dialog for incoming invitations, and the chat windows. The Chat UI component can also launch the Meeting Room Client for complex Sametime meetings.

Description

You must load the Chat UI component into the Sametime session object, like any other toolkit component. The Chat UI component listens to events from the Instant Messaging and Post Services, and notifies the user of incoming instant messages (IMs) and meeting invitations. It can also be used to easily initiate IMs and meetings.

Since the Chat UI component is not a Sametime-enabled AWT component, it cannot be embedded, and it displays more than one kind of UI dialog.

Initiating Instant Messages (IMs)

You can create an IM with another user in one of two ways. The table below identifies the two methods.

Table listing methods of creating instant messages.

If you have….	You will use…
The user name only	The create1On1Chat(String username) method. Chat UI tries to resolve the given string and start an IM session with the user. If the user cannot be resolved, an appropriate message will appear. If there is more than one registered user that matches the given name, a Resolve dialog will display the different matches from which the end user can select.
An STUser object	The create1On1ChatById(STUser user) method. Chat UI attempts to open an IM session with the user and displays a message if unsuccessful.

Initiating Meetings

There is only one method for creating a new Sametime meeting. When called, the Chat UI component will create the meeting, invite the requested users, and launch the appropriate window for the meeting:

```
        createMeeting(MeetingTypes meetingType, String meetingName,
             String inviteText, boolean showInviteDlg, STUser[] users)
```

The various meeting types are defined in the MeetingTypes class. The table below lists and describes the meeting types.

Table listing meeting types.

Meeting Type	Used For
MeetingTypes.ST_CHAT_MEETING	A chat meeting
MeetingTypes.ST_AUDIO_MEETING	An audio meeting
MeetingTypes.ST_VIDEO_MEETING	A video meeting
MeetingTypes.ST_SHARE_MEETING	An application-sharing meeting
MeetingTypes.ST_COLLABORATION_MEETING	A collaboration meeting

Setting the showInviteDlg flag to true will launch an optional invitation dialog before the meeting invitations are sent. This dialog enables the end user to select the participants of the meeting using the dialog UI.

Inviting to an existing meeting

A user usually invites another user to an existing meeting using the meeting UI. At any time a meeting participant can select "Invite others" from a menu to open the Invitation dialog to invite other users to the meeting. The application itself can also invite users using the following method:

```
  inviteToMeeting(MeetingInfo info, Place place, String inviteText,
        STUser[] invitees, boolean showInviteDlg, boolean autoJoin);
```

The meeting is described by a MeetingInfo object (described below). Also, the `place` object in which the meeting is held is required.

Receiving Invitations to Meetings

The Chat UI component enables you to receive instant messages (IMs) and invitations to meetings from other users in the community. When a user initiates an IM with you, a message window will pop up automatically. For meetings, a Join dialog will pop up with the meeting description, which allows you to decide if you want to join the meeting. By default, Chat UI only accepts invitations to chat meetings.

Meetings that include IP Audio, IP Video, or Application Sharing capabilities are launched in a separate browser window using the Meeting Room Client (MRC). Launching a browser is dependent on the toolkit's environment and therefore the toolkit does not include code to do so. Therefore, in order for an application to be able to participate in such meetings it should register itself as a MeetingListener by calling addMeetingListener(). Whenever Chat UI wants to launch the MRC, the launchMeeting() event will be generated with a MeetingInfo object describing the meeting and a URL object with the URL that the browser needs to be pointed to. A typical implementation of this event handler for an applet is:

```
Public void launchMeeting(MeetingInfo info, URL url)
{
  applet.getAppletContext().showDocument(URL);
}
```

Handling the UrlClicked() Event

By default, text in a URL format will be shown underlined inside a chat transcript window of an IM or chat meeting. When a user clicks on the URL, an event is fired. An application should register as a UrlClickListener by calling the addUrlClickListener() method to handle this case. A typical implementation of this event handler is:

```
public void urlClicked(UrlClickEvent event)
{
  String urlString = event.getURL();
  URL url;

  try
  {
    url = new URL(urlString);
  }
  catch (MalformedURLException e)
  {
    return;
  }

  applet.getAppletContext().showDocument(url);
}
```

Customizing Chat UI behavior

You can alert users when a new one-on-one chat message arrives. The following table lists the types of alerts and how to set them.

Table listing alert types.

Type of Alert	How to Set
Audible beep	setBeepOnMessage(Boolean beep);
Blink on the frame title	set/getBlinkOnMessage(Boolean blink);
Message pops up on top of other windows	set/getToFrontOnMessage(Boolean toFront);

If more than one IM window is opened, the windows will be cascaded. The application can set the cascading direction. The windows can either open from the top right corner and cascade to the left, or open from the top left corner and cascade to the right. Use:

```
SetCascadingDirection(in direction);
```

where the possible directions are ChatUI.CASCADE_LEFT and ChatUI.CASCADE_RIGHT.

Package

```
com.lotus.sametime.chatui
```

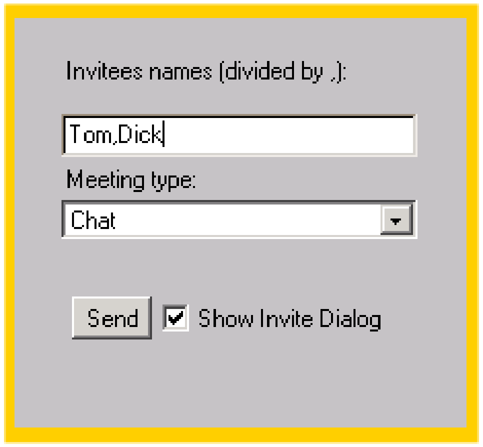
See Also

The Instant Messaging Service section of this guide

The Places Service section of this guide

Example

The following example contains an applet that logs on to the Sametime community and allows you to initiate (and receive) instant messages and all the possible types of Sametime meetings. Enter the user names, select the meeting type, select whether you want to display the Invite dialog before you send the invitation, and click the "Send" button. You can run this Chat UI Component sample from the Java Toolkit Samples page.


Invitees names (divided by ,):
Tom,Dick
Meeting type:
Chat

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.Vector;
import java.util.StringTokenizer;

import com.lotus.sametime.chatui.*;
import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.core.types.STUser;
import com.lotus.sametime.community.*;
import com.lotus.sametime.commui.*;
import com.lotus.sametime.core.constants.MeetingTypes;

/**
 * A sample of the chat ui component.
 */
public class ChatUIApplet extends Applet implements LoginListener,
                                                    MeetingListener,
                                                    UrlClickListener,
                                                    ActionListener,
                                                    CommUIListener
{
```

```
/**
 * The session.
 */
private STSession m_session;

/**
 * The chat ui component.
 */
private ChatUI m_chatUI;

/**
 * The community ui component.
 */
private CommUI m_commUI;

/**
 * The number of names that are waiting to be resolved.
 */
private int m_namesToResolve;

/**
 * The list of resolved meeting users.
 */
private Vector m_meetingUsers = new Vector();

/**
 * The 'meeting type' choice.
 */
private Choice m_chType;

/**
 * The user names text field.
 */
private TextField m_tfInvitees;

/**
 * The 'send' button.
 */
private Button m_btnSend;

/**
 * The 'Show invite dialog' check box.
 */
private Checkbox m_cbInviteDlg;


/**
 * Applet life cycle
 */
public void init()
{
  // Create a session of components.
  try
  {
    m_session = new STSession("ChatUIApplet");
    m_session.loadAllComponents();
    m_session.start();
  }
  catch(DuplicateObjectException e)
  {
    e.printStackTrace();
  }

  initializeLayout();

  // Login to the community.
```

```
    String serverName = getCodeBase().getHost().toString();
    String loginName  = getParameter("loginName");
    String password   = getParameter("password");


    CommunityService comm = (CommunityService)m_session.getCompApi
      (CommunityService.COMP_NAME);
    comm.addLoginListener(this);
    comm.loginByPassword(serverName, loginName, password);

    m_chatUI = (ChatUI)m_session.getCompApi(ChatUI.COMP_NAME);
    m_chatUI.addUrlClickListener(this);
    m_chatUI.addMeetingListener(this);

    m_commUI = (CommUI)m_session.getCompApi(CommUI.COMP_NAME);
    m_commUI.addCommUIListener(this);
  }

  /**
   * Applet destroyed. Stop and unload the session.
   */
  public void destroy()
  {
    CommunityService comm = (CommunityService)
            m_session.getCompApi(CommunityService.COMP_NAME);

    comm.logout();
    m_session.stop();
    m_session.unloadSession();
  }

  //
  // Login Listener.
  //

  /**
   * Logged in to the community.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Applet: LOGGED IN");
    m_btnSend.setEnabled(true);
  }

  /**
   * Logged out of the community.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("SAMPLE: LOGGED OUT. REASON=" +
                       event.getReason());
    m_btnSend.setEnabled(false);
  }

  //
  // Action Listener.
  //

  /**
   * Send button was clicked.
   */
  public void actionPerformed(ActionEvent p1)
  {
    String[] userNames = breakString(m_tfInvitees.getText(), ",");
    if (userNames == null || userNames.length == 0)
    {
      return;
    }

    // Disable the send button until wer'e finished.
    m_btnSend.setEnabled(false);

    // Try to resolve the names.
    m_namesToResolve = userNames.length;
    for (int i=0; i < m_namesToResolve; i++)
    {
      m_commUI.resolve(userNames[i]);
    }
  }

  //
  // CommuiListener.
  //
  /**
   * Resolve request succeded.
   */
  public void resolved(CommUIEvent event)
  {
    STUser resolved = event.getUser();
    m_meetingUsers.addElement(resolved);

    if (--m_namesToResolve == 0)
    {
      createMeeting();
    }
  }

  /**
   * Resolve request failed.
   */
  public void resolveFailed(CommUIEvent event)
  {
    if (--m_namesToResolve == 0)
    {
      createMeeting();
    }
  }

  /**
   * All names resolved. Create the meeting.
   */
  private void createMeeting()
  {
    m_btnSend.setEnabled(true);

    String meetingType = m_chType.getSelectedItem();

    // Check if it is an 1on1 instant message.
    if ((meetingType == "Chat") && m_meetingUsers.size() == 1)
    {
      STUser imPartner = (STUser)m_meetingUsers.firstElement();
      m_chatUI.create1On1ChatById(imPartner);

      m_meetingUsers.removeAllElements();
      return;
    }

    // Create the meeting type accoding to the selected string.
    MeetingTypes type = null;
    if (meetingType == "Chat")
    {
      type = MeetingTypes.ST_AUDIO_MEETING;
    }
    else if (meetingType == "Audio")
    {
      type = MeetingTypes.ST_AUDIO_MEETING;
    }
    else if (meetingType == "Video")
    {
      type = MeetingTypes.ST_VIDEO_MEETING;
    }
    else if (meetingType == "Share")
    {
      type = MeetingTypes.ST_SHARE_MEETING;
    }
    else if (meetingType == "Collaboration")
    {
      type = MeetingTypes.ST_COLLABORATION_MEETING;
    }

    STUser[] invitees = new STUser[m_meetingUsers.size()];
    m_meetingUsers.copyInto(invitees);

    boolean showInviteDlg = m_cbInviteDlg.getState();

    m_chatUI.createMeeting(type, "", "", showInviteDlg, invitees);
    m_meetingUsers.removeAllElements();
  }

  //
  // Meeting Listener.
  //

  /**
   * Open the meeting room client to participate in a meeting.
   * This event is received as a result of this user initiating
   * a meeting or being invited to join a meeting by a different
   * user.
   */
  public void launchMeeting(MeetingInfo meetingInfo, URL url)
  {
    getAppletContext().showDocument(url,
                   meetingInfo.getDisplayName());
  }


  /**
   * An error has occurred during a meeting creation.
   */
  public void meetingCreationFailed(MeetingInfo meetingInfo,
                                    int reason)
  {
    System.err.println("Failed to create the meeting: " + reason);
  }

  //
  // Url Click Listener.
  //

  /**
   * Url was clicked. Open it.
   */
  public void urlClicked(UrlClickEvent event)
  {
    URL url;
    String urlString = event.getURL();
    try
    {
      url = new URL(urlString);
    }
    catch(MalformedURLException e)
    {
      System.err.println("URL Clicked: MALFORMED URL");
      return;
    }

    getAppletContext().showDocument(url);
  }

  //
  // Helpers.
  //

  /**
   * Set up the ui components.
   */
  void initializeLayout()
  {
    m_chType = new Choice();
    m_chType.add("Chat");
    m_chType.add("Audio");
    m_chType.add("Video");
    m_chType.add("Share");
    m_chType.add("Collaboration");

    Panel p = new Panel(new GridLayout(3,1));

    Panel p1 = new Panel();
    p1.add(m_btnSend = new Button("Send"));
    p1.add(m_cbInviteDlg = new Checkbox("Show Invite Dialog", true));

    Panel p2 = new Panel(new GridLayout(3,1));
    p2.add(new Label("Meeting type:"));
    p2.add(m_chType);
    p2.add(p1);

    Panel p3 = new Panel (new BorderLayout());
    p3.add(new Label("Invitees names (divided by ,):"),
           BorderLayout.CENTER);
    p3.add(m_tfInvitees = new TextField(),BorderLayout.SOUTH );

    p.add(p3);
    p.add(p2);
    p.add(p1);

    add(p);

    m_btnSend.addActionListener(this);

    // Enable sending only when we are logged in.
    m_btnSend.setEnabled(false);
  }

  /**
   * Break a string into an array of smaller strings, based on a given
   * separator char.
   */
  protected String[] breakString(String toBreak, String separator)
  {
    String[] result = null;
    if (toBreak != null && !toBreak.equalsIgnoreCase("")) {
      if (separator == null || separator.equalsIgnoreCase("")) {
        result = new String[1];
        result[0] = toBreak;
        }
        else {
          StringTokenizer stk = new StringTokenizer(toBreak,
                                                    separator);
          result = new String[stk.countTokens()];
          for (int i = 0; i < result.length && stk.hasMoreTokens();
               i++)
            result[i] = stk.nextToken();
      }
      return result;
    }
  }
```

Customizing the Chat UI Component

Overview

Customizing the Chat UI component provides the ability to customize the chat window without having to design a new window.

You can customize the chat window by:

- Adding user panels to three fixed positions
- Adding menus and menu items to the window menu bar
- Customizing the text before it is sent to the chat session

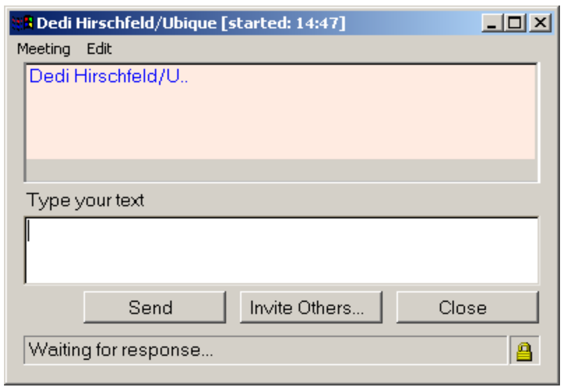
Customizing the Chat UI component allows the addition of more features to the chat window to suit the user's specific needs.

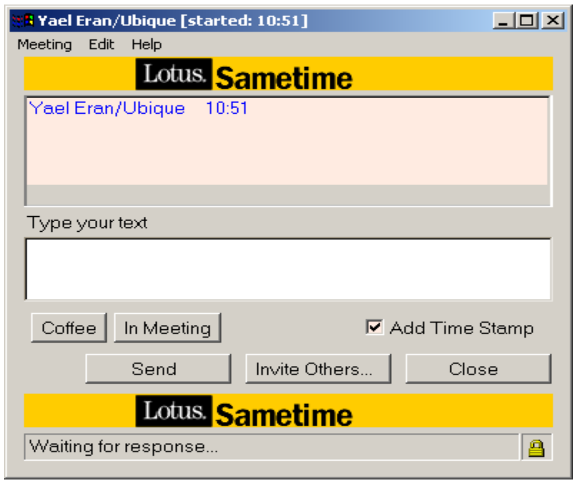
These changes appear in both the IM and conference windows.

Description

To customize the Chat UI, you will need to create a new class that extends the DefaultChatFactroy and overrides it with the needed functions. The DefaultChatFactroy, which implements the ChatFactory interface, responds to calls from the ChatUI components by generating the appropriate UI component for the request.

Below are examples of the regular and customized chat windows:





Adding user panels

This section describes:

- Adding a Logo Panel
- Adding Buttons Panel

You can add up to three panels by overriding this method:

```
public Panel getCustomizedPanels(int panelPosition, ChatFrame frame)
```

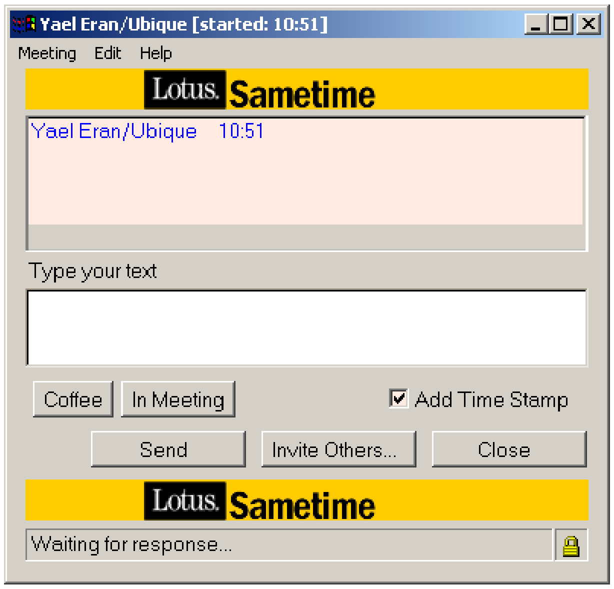
The first parameter is panelPosition.

The table below describes the position of the constants on the panel:

Table listing position constants.

Constants	Position
DefaultChatFactory. TOP_PANEL	On top on the user panel under the menu.
DefaultChatFactory. CENTER_PANEL	In the center on the user panel above of the buttons.
DefaultChatFactory. BOTTOM_PANEL	At the bottom on the user panel above the status bar.

Below is an example of the possible positions:



The second parameter is ChatFrame, the container for the customized panels. The method getCustomizedPanels returns the panel associated with the requested position.

A typical implementation of this method is:

```
public Panel getCustomizedPanels(int panelPosition,
                                 ChatFrame frame)
    {
…
        if (panelPosition == TOP_PANEL)
            return createLogoPanel("logo.gif");
        if (panelPosition == CENTER_PANEL)
            return ShortcutButtons(frame);
        if (panelPosition == BOTTOM_PANEL)
            return createLogoPanel("logo.gif");
        return null;
    }
```

The panel can contain image buttons, banners, links, and more. The ChatFrame will insert the user panel into the specified position.

Adding a Logo Panel

In the given sample, the top and bottom panels each contain a logo. The logo panels are created by the following method:

```
private Panel CreateLogoPanel(String logo)
{
  Panel  p = new ImagePanel(getLogoURL(logo));
  p.setBackground(new Color(0xFFcc00));
  return p;
}
```

The getLogoURL(logo) returns the URL of the image (where the samples are). The ImagePanel creates a panel with the image that was given in the constructor. The complete code is available for download from the Java Toolkit section on the Sametime server.

Adding Buttons Panel

On the center panel there are two buttons that send pre-prepared messages and a check box to add a time stamp to the sent text. (See the Modify Written Text section of this chapter for more information.)

The center panel is created by the following method:

```
public Panel ShortcutButtons(ChatFrame frame)
```

The complete code is available for download from the Java Toolkit section on the Sametime server.

Adding a Menu and Menu Items

Another way to customize the chat window is to add a menu to the menu bar by overriding the method:

```
public void getCustomizedMenu(MenuBar mb)
```

You now have access to the frames MenuBar and you are able to add menus and menu items.
In the following sample code, a typical implementation of this method is used to add a help menu with an "About" menu item.

```
    public void getCustomizedMenu(MenuBar mb)
    {
        Menu helpMenu = new Menu("Help");
        MenuItem about = new MenuItem("About");
        about.addActionListener(new ActionListener()
            {
               public void actionPerformed(ActionEvent event)
                {
                   try
                   {
m_applet.getAppletContext().showDocument(newURL("http://www.lotus.com/home.nsf
/welcome/sametime"),"target");
                   }
                   catch (MalformedURLException e)
                   {
                       e.printStackTrace();
                   }
                }
            });

        helpMenu.add(about);
        mb.add(helpMenu);
    }
```

Note The existing menus (Meeting and Edit) cannot be modified.

Modify Written Text

The ability to intercept text before it is sent enables you to modify the original text into a customized version. This ability is useful, to add a check spelling service to the chat for example; or as in our sample, to add a time stamp to the written text.

In order to intercept text messages, a developer needs to perform the following steps.

A TextModifier listener has to be attached to the ChatFrame. This is done by implementing the TextModifier interface that includes only the method shown below:

```
public String TextSubmitted(String text)
```

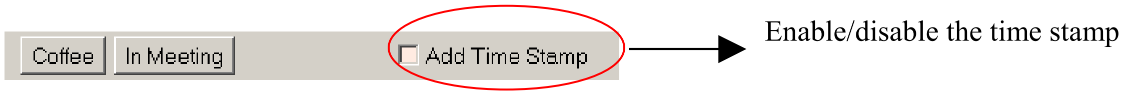
This function is called every time the text is submitted. Prior to being sent, the method gets the typed text, modifies it, and returns the modified text to be sent.

To return the object that implements the TextModifier interface, you must override the method. Overriding the DefaultChatFactory's method:

```
public TextModifier getTextModifier(ChatFrame frame)
```

This method is called by each ChatFrame, upon its initialization, to get the appropriate TextModifier that will be associated with this frame. If it is not implemented, this method will return the default TextModifier that is set to null. In our sample, the newChatFactory implements the TextModifier interface.

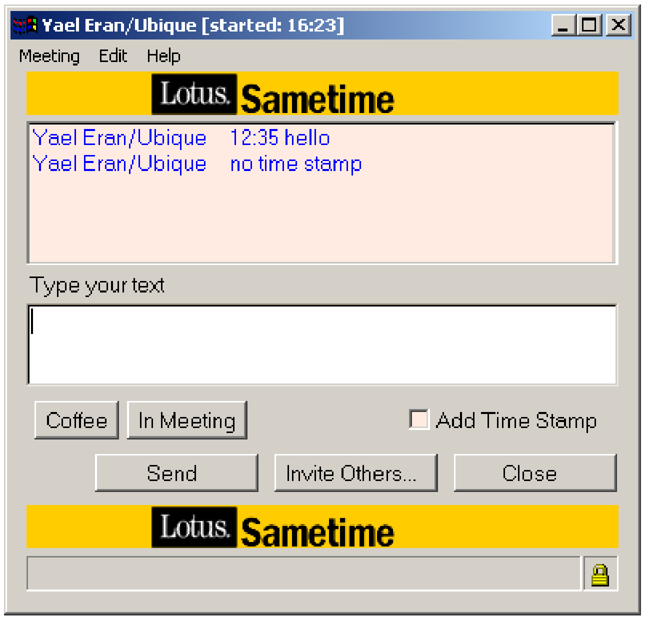
In the center panel is a check box that adds the time to the chat message that was written. Changing the status of the check box enables/disables adding the time stamp.



The following is the implementation of this method:

```
public String TextSubmitted(String text)
{
          if(m_addTime.getState())
          {
             text =getTimeStamp ()+text;
          }
return text;
}
```

```
In the sample, the current time is taken from the system and attached to the
beginning of the sent message.
```



Yael Eran/Ubique 12:35 hello
Yael Eran/Ubique no time stamp

Example

The sample shown on the server (and shown in the above screen shot) opens a customized chat window that contains two logo panels, one at the top and one at the bottom. The sample also contains a center panel that contains buttons that send pre-prepared messages, and a check box that adds a time stamp to the written text. A help menu was added to the modified chat window.

Note To view the complete code, you need to download the sample from the server.

The sample contains three classes:

- **CustomizeChatUI** – Extends an Applet and sets the NewChatFactory as the ChatFactory for the ChatUI component
- **CustomizeChatFactory** – Extends the DefaultChatFactory and provides customized UI

- **ImagePanel** – Creates a panel with an image

Community UI Component

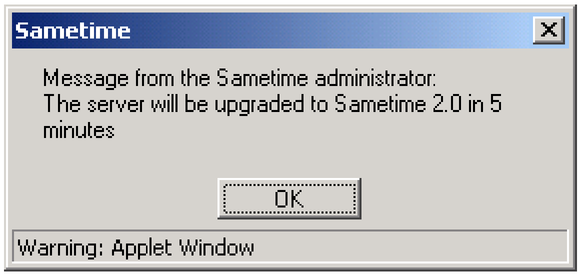
Overview

The Community UI component provides the following capabilities:

- Display of messages when the user is disconnected from the community. The message includes a short text description of the specific reason.
- Display of administrator messages that are sent to the entire community.
- Resolving user names in the community. When more than one user match is found, a dialog allowing the end user to select the appropriate match is displayed.

Description

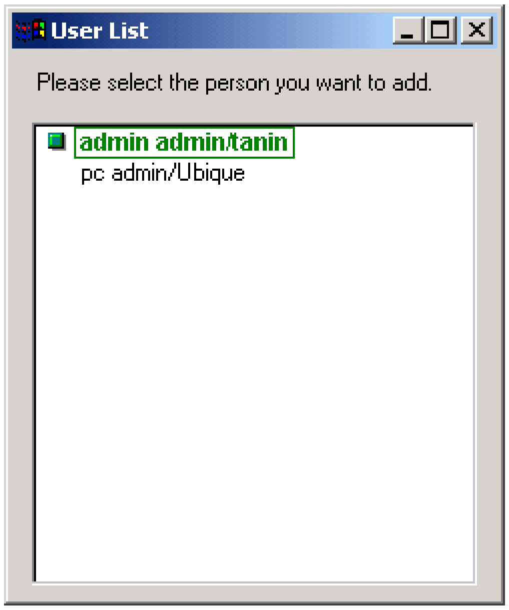
The following is a dialog of an administrator message:



Message from the Sametime administrator:
The server will be upgraded to Sametime 2.0 in 5 minutes

Warning: Applet Window

When you resolve user names and more than one match is found, the following dialog is displayed:



Please select the person you want to add.

admin admin/tanin
pc admin/Ubique

To resolve a user name using the Community UI:

1. Get a reference to the Community UI component:

```
        CommUI commUI = (CommUI)
           m_session.getCompApi(CommUI.COMP_NAME);
```

2. Listen to community UI events by calling the addCommUIListener() method and passing a class that implements CommUIListener or extends CommUIAdapter.
3. Call the resolve method with the required user name:

```
    commUI.resolve("admin");
```

4. You can receive the following event notifications as a response to the resolve request:
   - **resolved()** – the user name was resolved successfully
   - **resolveFailed()** – the user name was not resolved
5. If the more then one match is found for the user name, a dialog will be launched (see above) and the end user can then make a selection. After the users makes a selection, a resolved() event will be fired with the selected user details.

Package

`com.lotus.sametime.commui`

See Also

The Community Services section of this guide

FileTransfer UI Component

Overview

The FileTransfer UI component provides the ability to send and receive files without having to deal with the low-level protocols that are involved. It includes the SendFile dialog, used for choosing the file to send and initiating the file transfer; the ReceiveFile dialog for accepting or declining the file transfer; and the FileTransferStatus dialog that indicates the progress of the transfer on both sides.

Note that if you are creating an applet that supports file transfer, you must sign the applet, to allow local file access.

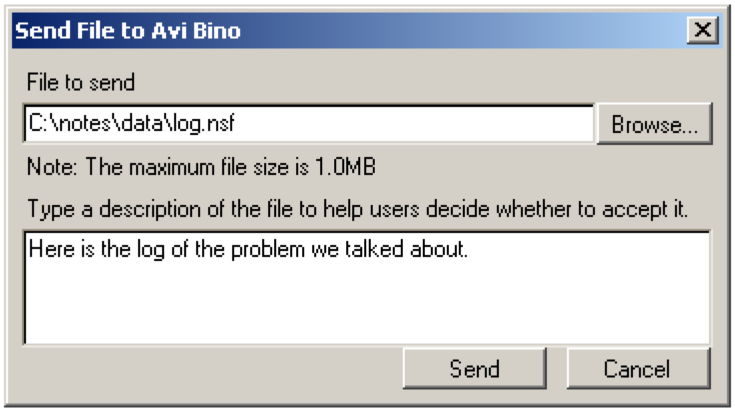
Description

You must load the FileTransfer UI component into the Sametime session object, like any other toolkit component. The FileTransfer UI component listens to events from the FileTransfer Service and notifies the user of incoming file transfers. It can also be used to easily initiate file transfers.

Sending Files

To send a file, call the sendFile() method of the FileTransferUI Service. This method takes the remote user's STUser object as a parameter.

A SendFile dialog opens, letting the user choose a file to send and add a description.


Send File to Avi Bino

File to send

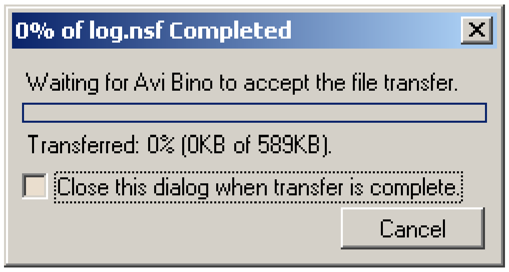
C:\notes\data\log.nsf

Note: The maximum file size is 1.0MB

Type a description of the file to help users decide whether to accept it.

Here is the log of the problem we talked about.

When the user clicks Send, a FileTransfer object is created. The transfer starts as soon as the other side accepts it. Then the FileTransferStatus dialog opens, indicating the progress of the transfer.


0% of log.nsf Completed

Waiting for Avi Bino to accept the file transfer.

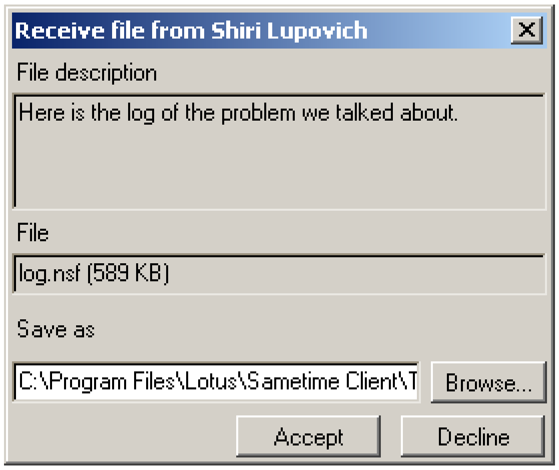
Transferred: 0% (0KB of 589KB).

Close this dialog when transfer is complete.

If you want to get notifications when the file transfer is completed or stopped, implement the FileTransferUIListener interface and use the addFileTransferUIListener() method of the FileTransfer Service to register the listener object, before you call the sendFile() method.

Receiving Files

To receive file transfers, you need only load the FileTransfer UI component into the Sametime session object. The FileTransfer UI component listens to events from the FileTransfer Service, and opens a ReceiveFile dialog as soon as a file transfer request is received.


Receive file from Shiri Lupovich

File description

Here is the log of the problem we talked about.

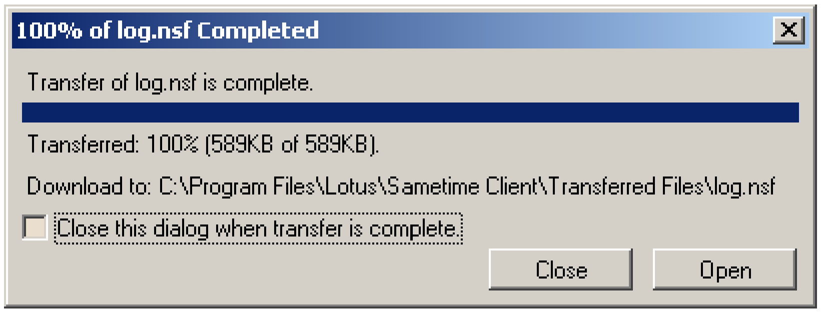
File

log.nsf (589 KB)

Save as

C:\Program Files\Lotus\Sametime Client\T

The user can either accept or decline the transfer. If the user accepts, a FileTransferStatus dialog opens, indicating the progress of the transfer. After successful completion, the user can open the file directly from the FileTransferStatus dialog.


100% of log.nsf Completed

Transfer of log.nsf is complete.

Transferred: 100% (589KB of 589KB).

Download to: C:\Program Files\Lotus\Sametime Client\Transferred Files\log.nsf

If you want to get notifications when the file transfer is completed or stopped, implement the FileTransferUIListener interface and use the addFileTransferUIListener() method of the FileTransferUI Service to register the listener object, before you call the accept() method.

Package

com.lotus.sametime.filetransferui

See Also

The FileTransfer Service section of this guide

Example

This example consists of two classes: UIFileSenderSample and UIFileReceiverSample.

UIFileSenderSample displays a 'Send a File' button. After clicking this button, the user sees the SendFile dialog of the toolkit, and can select the file to send. The transfer starts when the user clicks Send. When the transfer is completed, the user can send another file by clicking 'Send a File' again.

The receiver logs in and then waits for a file transfer. Upon receiving a file transfer, the toolkit creates a ReceiveFile dialog and the user can accept the transfer or decline it. After the transfer is completed, the receiver waits for another transfer.

To run this example, simply copy the code and change the constants defined at the beginning of each class to real values from your community. Be sure to run the receiver before you click the Send button, because the receiver must be already logged in when the sender tries to send the file.

```
import java.util.Date;
import java.awt.*;
import java.awt.event.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.core.util.Debug;
import com.lotus.sametime.community.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.filetransferui.*;

public class UIFileSenderSample extends Frame implements LoginListener,
FileTransferUIListener, ResolveListener, ActionListener
{
         // replace all these constants with real values from your
   // community
   private final static String SERVER_NAME = "server_name";
   private final static String SENDER_NAME = "sender_user_name";
   private final static String PASSWORD = "sender_password";
   private final static String RECEIVER_NAME    = "receiver_user_name";
   private STSession m_session;
   private CommunityService m_comm;
   private String m_remoteUserName;
   private FileTransferUI m_fileTransferUI;
   private Button m_sendBtn;
   private STUser m_receiver;

   public UIFileSenderSample(String host, String userName, String password, String
         remoteUserName)
   {
      try
      {
         m_session = new STSession("UIFileSenderSample " + this);
         m_session.loadAllComponents();

         m_session.start();

         Frame frame = findParentFrame();
         System.err.println("Mainframe = " + frame);
         m_session.setSessionProperty("mainFrame", frame);

         m_comm = (CommunityService)
            m_session.getCompApi(CommunityService.COMP_NAME);
         m_comm.addLoginListener(this);

         m_remoteUserName = remoteUserName;

         m_sendBtn = new Button("Send A File");
         m_sendBtn.setEnabled(false);
         m_sendBtn.addActionListener(this);
         add(m_sendBtn,BorderLayout.CENTER);

         pack();

         setVisible(true);

         FileTransferUI fileTransferUISvc = (FileTransferUI)
            m_session.getCompApi(FileTransferUI.COMP_NAME);
fileTransferUISvc.addFileTransferUIListener(this);
         m_comm.loginByPassword(host, userName, password);
      }
      catch(DuplicateObjectException e)
      {
         e.printStackTrace();
      }
   }

   /**
   * Main
   */
   public static void main(String[] args)
   {
      UIFileSenderSample uIFileSenderSample1 =
         new UIFileSenderSample(SERVER_NAME ,SENDER_NAME, PASSWORD,
                              RECEIVER_NAME);
   }

   //
   // LoginListener
   //

   /**
   * Logged in event.
   */
   public void loggedIn(LoginEvent event)
   {
      System.out.println("Logged In");
      m_fileTransferUI = (FileTransferUI)
         m_session.getCompApi(FileTransferUI.COMP_NAME);

      LookupService lookupSvc = (LookupService)
         m_session.getCompApi(LookupService.COMP_NAME);

      Resolver resolver = lookupSvc.createResolver(true, true,
                                                   true, false);
      resolver.addResolveListener(this);
      resolver.resolve(m_remoteUserName);
   }

   /**
   * Logged out event.
   */
   public void loggedOut(LoginEvent event)
   {
      System.out.println("Logged Out");
      m_session.stop();
   }

   //
   // ResolveListener
   //

   /**
   * A resolve request has been performed, and a match was found.
   *
   * @param            event The event object.
   * @see              ResolveEvent#getName
   * @see              ResolveEvent#getResolved
   */
   public void resolved(ResolveEvent event)
   {
      m_receiver = (STUser)event.getResolved();

      Frame frame = new Frame("the frame");
      m_sendBtn.setEnabled(true);
   }

   /**
   * A resolve request has been performed, and multiple matches have
   * been found.
   *
   * @param            event The event object.
   * @see              ResolveEvent#getName
   * @see              ResolveEvent#getResolvedList
   */
   public void resolveConflict(ResolveEvent event)
   {
      System.out.println("FileSender Resolve conflict");
      m_comm.logout();
   }

   /**
   * A resolve request failed.
   *
   * @param            event The event object.
   * @see              ResolveEvent#getReason
   * @see              ResolveEvent#getFailedNames
   */
   public void resolveFailed(ResolveEvent event)
   {
      System.out.println("FileSender Resolve Failed");
      m_comm.logout();
   }

   //
   // ActionListener
   //

   public void actionPerformed(ActionEvent event)
   {
      //send the file
      m_fileTransferUI.sendFile(m_receiver);
   }

   //
   // FileTransferUIListener
   //

   /**
   * Notification sent when a file transfer session has been completed
   * successfully.
   */
   public void fileTransferCompleted(FileTransferUIEvent event)
   {
      Date date = new Date();
      System.out.println("File Transfer Completed: " +
         event.getFileName() +
         " Time: " + date.toString());
   }

   /**
   * Notification sent when a session has been stopped/terminated.
   */
   public void fileTransferFailed(FileTransferUIEvent event)
   {
   System.out.println("File Transfer Failed reason: " +
      event.getReason() +
      "File: " +
      event.getFileName());
   }

   //
   // Helpers
   //

   /**
   * Find the parent frame of this applet.
   */
   public Frame findParentFrame()
   {
      Frame frame = null;
      Container c = this.getParent();
      while(c != null)
      {
         if(c instanceof Frame)
         {
            frame = (Frame)c;
            break;
         }

      c = c.getParent();
      }
      return frame;
   }
}
```

```
import java.util.Date;
import java.awt.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.community.*;
import com.lotus.sametime.core.util.Debug;
import com.lotus.sametime.filetransferui.*;

public class UIFileReceiverSample extends Frame implements LoginListener,
FileTransferUIListener
{

   // replace all these constants with real values from your
   // community
   private final static String SERVER_NAME      = "server_name";
   private final static String RECEIVER_NAME    = "receiver_user_name";
   private final static String PASSWORD="receiver_password";

   private STSession m_session;
   private CommunityService m_comm;

   public UIFileReceiverSample(String host, String userName,
                               String password)
   {
      try
      {
         m_session = new STSession("UIFileReceiverSample " + this);
         m_session.loadAllComponents();

         m_session.start();

         Frame frame = findParentFrame();
         System.err.println("Mainframe = " + frame);
         m_session.setSessionProperty("mainFrame", frame);

         m_comm = (CommunityService)
            m_session.getCompApi(CommunityService.COMP_NAME);
         m_comm.addLoginListener(this);

         FileTransferUI fileTransferUISvc = (FileTransferUI)
            m_session.getCompApi(FileTransferUI.COMP_NAME);
fileTransferUISvc.addFileTransferUIListener(this);
         m_comm.loginByPassword(host, userName, password);
      }
      catch(DuplicateObjectException e)
      {
         e.printStackTrace();
      }
   }


   /**
   * Main.
   */
   public static void main(String[] args)
   {
      UIFileReceiverSample uiFileReceiverSample1 =
         new UIFileReceiverSample(SERVER_NAME ,RECEIVER_NAME,
                                  PASSWORD);
   }

   //
   // LoginListener
   //
```

```
/**
* Logged in event.
*/
public void loggedIn(LoginEvent event)
{
   System.out.println("Logged In");
}

/**
* Logged out event.
*/
public void loggedOut(LoginEvent event)
{
   System.out.println("Logged Out");
   m_session.stop();
}

//
// FileTransferUIListener
//

/**
* Notification sent when a file transfer session has been completed
* successfully.
*/
public void fileTransferCompleted(FileTransferUIEvent event)
{
   Date date = new Date();
   System.out.println("File Transfer Completed: " +
      event.getFileName() +
      " Time: " + date.toString());
}

/**
* Notification sent when a session has been stopped/terminated.
*/
public void fileTransferFailed(FileTransferUIEvent event)
{
   System.out.println("File Transfer Failed reason: " +
      event.getReason() +
      "File: " +
      event.getFileName());
}

//
// Helpers
//

/**
* Find the parent frame of this applet.
*/
public Frame findParentFrame()
{
   Frame frame = null;
   Container c = this.getParent();
   while(c != null)
   {
      if(c instanceof Frame)
      {
         frame = (Frame)c;
         break;
      }
      c = c.getParent();
   }
   return frame;
```

```
    }
}
```

Appendix J. Deprecated Meeting Services Java APIs

Previous versions of the Sametime Java Toolkit shipped with Meeting Services Java APIs which allow client Java applets and applications to create and join multi-participant meetings. These Meeting Services Java APIs have been deprecated starting with Sametime 7.5. Support for these APIs will continue for two major releases.

The APIs are used to share and annotate documents, share and control applications, and communicate using live audio and video between meeting participants. The developer can choose the tools to use in each meeting and control the visual layout and containment of various UI components. The Meeting Services include Application Sharing, Shared Whiteboard, and Interactive Audio and Video and provides a highly scalable and lightweight Broadcast Receiver. The Broadcast Receiver receives and interprets low bandwidth audio, video, and data streams generated by the different Meeting Services, and displays these without actively participating in the meeting.

The Meeting Service API libraries along with full documentation and samples are included in the Sametime 7.0 Java Toolkit which can be downloaded from the IBM developerWorks Lotus downloads page:

http://www-128.ibm.com/developerworks/lotus/downloads/toolkits.html

Navigate to the IBM Sametime section, click on Sametime Java toolkit, then select IBM Sametime 7.0 Java Toolkit to download the 7.0 version which includes all the Meeting Services materials.

Notices

Trademarks