Oracle® Unified Messaging

Developer's Guide

Release 2.1.2

January, 2001

Part No. A86093-02

ORACLE

Oracle Unified Messaging Developer's Guide, Release 2.1.2

Part No. A86093-02

Primary Author: Ginger Tabora

Contributors: Byung Choung, Varouzhan Ebrahimian, Duane Jensen, Tom Kraikit, Jae Lee, Sunnia Lin, Allen Liu, Louise Luo, Stefano Montero, Howard Narvaez, Ricardo Rivera

Contents

3 Customizing Unified Messaging APIs

Send Us Your Comments

Oracle Unified Messaging Developer's Guide, Release 2.1.2

Part No. A86093-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev@us.oracle.com
- FAX: (650) 506-7228 Attn: Oracle Unified Messaging Documentation Manager
- Postal service:
  Oracle Corporation
  Unified Messaging Documentation Manager
  500 Oracle Parkway, Mailstop 4OP12
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services representative.

Preface

Intended Audience

This manual is intended for Unified Messaging client developers. It provides an introduction to Unified Messaging and describes the management tasks you will perform as an Unified Messaging server administrator.

Unified Messaging Documentation

Unified Messaging documentation is available in HTML and PDF format on the CD-ROM and installs automatically during product installation. Use your Web browser to access **$ORACLE_HOME/um/doc/index.html** on your server. The following documents are available:

Oracle Unified Messaging Release Notes

Oracle Unified Messaging Installation Guide

Oracle Unified Messaging Developer's Guide

Notation Conventions

The following notational conventions appear in this manual:

Convention	Description
italic	Italicized type identifies document titles.
`Monospace`	Monospace type indicates commands.
bold	Boldface type indicates script names, directory names, path names, and file names (for example, the **root.sh** script).
UPPERCASE	Uppercase letters indicate parameters or environment variables (for example, ORACLE_HOME).
. . .	In code examples, vertical ellipsis points indicate that information not directly related to the example has been omitted.
. . .	In command syntax, horizontal ellipsis points indicate repetition of the preceding parameters. The following command example indicates that more than one input_file may be specified on the command line: `command [input_file ...]`
< >	In command syntax, angle brackets identify variables that the user must supply. You do not type the angle brackets. The following command example indicates that the user must enter a value for the variable input_file: `command <input_file>`
[ ]	In command syntax, brackets enclose optional clauses from which you can choose one or none. You do not type the brackets. The following command example indicates that the variable output_file is optional: `command <input_file> [output_file]`
{ }	In command syntax, curly brackets indicate that a choice of two or more items separated by a vertical bar or pipe (|). You do not type the curly brackets. The following command example indicates a choice of either a or b: `command {a | b}`
$	The dollar sign represents the shell prompt in UNIX.

1

Overview of the Unified Messaging SDK

This chapter introduces the Unified Messaging Server Developer Kit (SDK) to the HTML developer. This chapter contains the following topics:

- Introduction to Unified Messaging
- Working with the Unified Messaging SDK
- System Requirements

Introduction to Unified Messaging

Today's business professional receives and sends messages via multiple sources: e-mail, voice mail, facsimiles, and short messages displayed on pagers. The challenge is to keep up with all these messages and quickly recognize high priority items that require immediate action. Oracle provides the solution: Unified Messaging, which integrates messages from multiple sources into a single "inbox." Unified Messaging not only consolidates all messages into a single interface, it also frees the business professional to focus on making decisions, rather than on keeping track of multiple telephone numbers, passwords, and access codes.

Unified Messaging from the Developer's Perspective

Unified Messaging provides an application development environment that includes Java packages, classes, and methods. These classes allow both front-end customization of the GUI and back-end customization of Unified Messaging functions.

The Unified Messaging SDK provides resources for both HTML developers and Java developers:

- HTML developers can use a sample client by modifying HTML pages.
- Java developers can write Java programs to extend the Unified Messaging SDK classes.

Unified Messaging Architecture

The Unified Messaging system architecture includes the following three tiers.

- Tier 1: A thin client, such as a Web browser
- Tier 2: An application layer, consisting of the Unified Messaging SDK and its environment:
  - A Unified Messaging SDK that runs in a JSP environment.
  - A Web server with JSP support enabled.
  - (Optional) RealAudio that provides audio streaming capabilities
- Tier 3: The data store layer, including:
  - An Oracle database that stores Unified Messaging information
  - The Oracle internet Directory (OiD) that stores directory information
  - The Oracle eMail Server

- The voice mail server and message store connection
- The facsimile server and message store connection
- The Short Message Service (SMS) gateway
- The Operational Support Systems (OSS) interfaces
- (Optional) Other IMAP4-compliant e-mail message stores

Unified Messaging Components

The Oracle Unified Messaging provides components related to Tier 2 and Tier 3. Oracle requires only that the user have a standard browser (a Real Audio plugin is optional). The sample application is based upon Java Server Pages.

- Tier 1: The graphical user interface of the Unified Messaging sample application, displayed in a Web browser. The HTML interface, generated dynamically, combining information from the database and information supplied in customized JSP templates.
- Tier 2: The Apache listener and JServ engine.
- Tier 3: Oracle eMail Server and an SMS gateway.

To complete the Unified Messaging system, Unified Messaging includes the interfaces used to connect all message sources:

- Alternate e-mail servers
- Voice mail servers
- Facsimile servers
- OSS system

Working with the Unified Messaging SDK

The Unified Messaging SDK runs in a programming environment that includes the following main components:

- The Unified Messaging Applications Programming Interface (Unified Messaging API)
- Apache and JServ
- Unified Messaging sample application

Unified Messaging API

The Unified Messaging API is a set of classes and methods, written in Java, used to implement the Unified Messaging system. Application developers use the standard component and object model to create their own custom messaging solution.

Sample Application

The sample application included, provides Web access to the Unified Messaging system through browsers

The sample application consists of a set of screens that give users access to standard Unified Messaging functions, including:

- E-mail access
- Directory access
- Audio and facsimile conversion (g726 to WAV or RealAudio, TIFF to GIF)
- SMS connections
- Administrative functions

The application can be accessed at the following locations after installation of the "UMSDK Application" component of the Unified Messaging SDK:

```
http://<hostname:port>/um/login/jsp
```

System Requirements

This section describes three types of requirements for using Unified Messaging:

- Environment requirements
- Requirements for customizing the Unified Messaging GUI
- Knowledge requirements for HTML developers

Environment Requirements

The following components are required for Unified Messaging to function on all platforms:

- Oracle database
- Web server with JSP support

- Oracle eMail Server
- Oracle Internet Directory
- A Web browser
- A development tool

You do not need a special development tool to develop HTML-based applications. Use a text editor or HTML editor or, if you prefer, a Web authoring tool to create and maintain the template files that make up your application. No such special development environments or tools are provided with the Unified Messaging SDK.

Requirements for Customizing the Unified Messaging GUI

To customize the sample applications included with the Unified Messaging SDK you must have access to the following:

- Oracle database
- Web server administration authority
- An account on an Oracle eMail Server

The sample applications run in the following Web browsers:

- Netscape Navigator 4.5x
- Internet Explorer 5.x

Knowledge Requirements

HTML application developers generally work with the provided Unified Messaging API to create applications. The simplest way to create applications is to modify the HTML templates provided in the Unified Messaging sample application.

This document assumes the HTML application developer has an understanding of HTML, including form syntax. It also assumes a knowledge of general HTTP concepts including the practical use of Web browsers, Web servers, and URLs. While knowledge of browser scripting languages like JavaScript is not required to develop simple HTML applications, such knowledge will assist in developing enterprise-quality HTML applications and in understanding the Unified Messaging GUI. Java programming knowledge is not required to program HTML applications, but an understanding of general object-oriented programming concepts and the ability to read Javadoc reference information is necessary.

2

Planning Your Development Strategy

This chapter describes planning your development strategy for the Unified Messaging server and includes the following topics:

- Choosing a Development Approach
- Development Tools
- General Design Considerations for HTML Development
- HTML Application Development Process Overview

Choosing a Development Approach

Web application developers typically take one of two main approaches in writing application to customize Unified Messaging, though additional approaches are possible. For complex tasks or tasks requiring special processing, you may also choose to extend the JavaBeans APIs in combination with either the HTML approach or the Java approach. The approach you choose, and whether you also choose to extend the JavaBeans APIs, depends on the complexity of the task, the target runtime environment or platform, and the areas of expertise of the developers. All approaches, however, access the Unified Messaging server through the JavaBeans APIs.

- HTML Approach: Using HTML application to access the JavaBeans APIs through the Java Server Pages.
- Java Approach: Using Java application to access the JavaBeans APIs directly.

HTML Approach

With the HTML approach, you develop templates that consist of a combination of standard HTML and URL syntax. The JSP engine passes templates and executes Java code held within. Variables of the calling URL, browser cookies, and browser environment variables passed in the HTTP request can be accessed using Java servlets. Once processed, sends this dynamically generated HTML is sent to the browser.

This approach offers the following advantages:

- HTML application use only standard ASCII syntax, allowing rapid prototyping and development without the need to compile code.
- You can use scripting languages such as JavaScript™, JScript™, or VBScript™ to enhance the functionality or interactivity of your application.
- You can add rich functionality without requiring Java programming expertise.
- It requires only a standard Web browser on the client machine, making it ideal for thin-client solutions.

Java Approach

With the Java approach, you develop custom Java application that pass instructions to the UM APIs directly. Java application can achieve all of the functionality of HTML application, by directly invoking API methods to access and manipulate

attributes. Although the Java approach is a development option, writing Java application is beyond the scope of this document.

Development Tools

You do not need a special development tool to develop HTML-based application for Unified Messaging. You can use a text or HTML editor or, if you prefer, a Web authoring tool to create and maintain the template files that make up your application.

To create Java applets or use any browser-compatible scripting, you can use development tools. No special development environments or tools are provided with the Unified Messaging SDK.

General Design Considerations for HTML Development

The visible part of most HTML application will be a collection of HTML templates. Templates are the building blocks of your application and include a variety of programming information.

You can build new HTML pages and use these as templates for your application or you can modify existing templates. Regardless of where you begin, there are several factors to consider before you write the application.

Consider How to Display Data

You should consider the number of runtime application objects—for example, message titles—that appear in the interface and how template tags are replaced by those objects. Also consider the formatting of objects included by template tags.

Consider How to Create Links

Links in an HTML application can be static URLs to other pages of the application or detailed functional instructions for the application's next operation.

When you build application URLs, you will specify the next template to use and the error template to use. Before you begin developing your Unified Messaging application, consider how you want to create links from text—like lists of message subjects—and links from forms. When planning a link from an application page, consider:

- What's the next HTML page to display after an action?

- What's the next HTML page to display after an error?

This diagram is a basic example of how various lists and forms may be linked in an application.

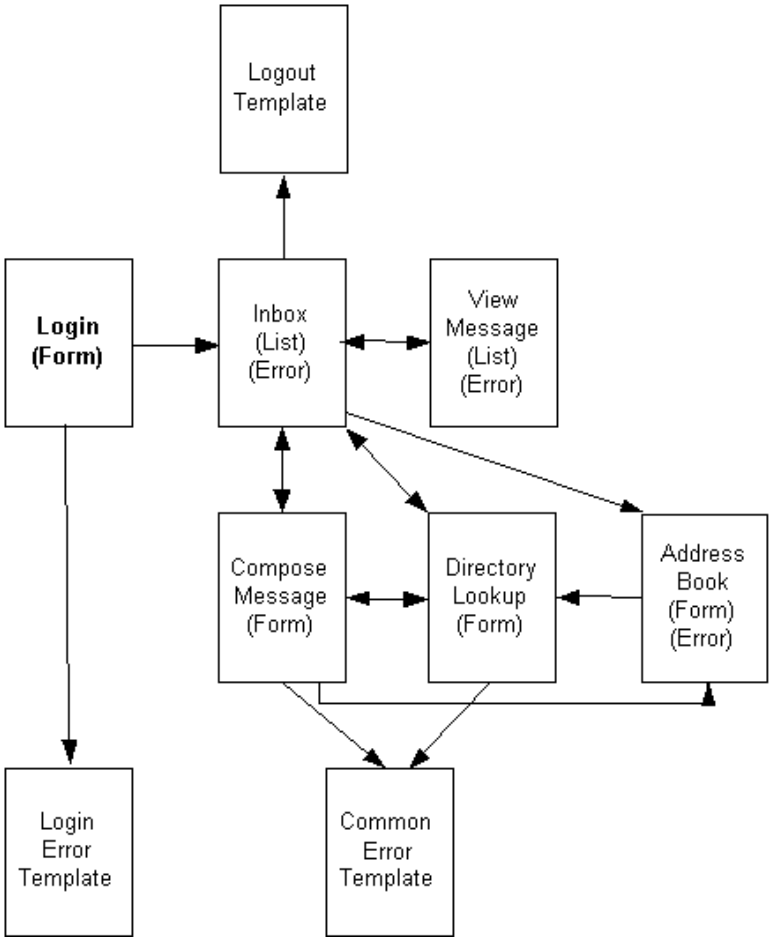

Figure 2-1

The diagram also illustrates how you can display error messages in the current HTML template (e.g., the Inbox, View Message, and Address Book templates), or you can build dedicated error message templates.

Consider How to Use and Create Forms

HTML forms allow application users to enter a variety of information using text fields, radio buttons, check boxes, and selection menus. You should plan to use forms to allow users to log into your application, to search for text, and to enter text for use in messages or as other attribute information.

Submitting a form constructs a URL. The content of the URL submitted by the form is composed of a series of name=value pairs that are taken from the names and values of elements in the form. The name=value pairs can be used to define, create, and manipulate application objects such as, messages using references to existing application objects and the objects in the JavaBeans APIs. The placement order of form elements determines the order of name=value pairs that are submitted in URLs.

Note: When constructing HTML forms, Oracle Corporation recommends using POST for the form method. POST will not display the URL "contents" in the location text field of the browser and allows for larger data transfer. GET is a valid form method; however, it is subject to URL length limitations.

Consider Your Network Bandwidth

Network bandwidth limitations may slow application performance and decrease user productivity. For example, if you are building a mail application for use over modem lines, it might be frustrating to the user to have to wait while a list of the subject lines of the 3000 messages in the user's inbox is created and sent out down the line. It is important to consider your network bandwidth prior to developing and deploying your application. For example, if you build a limit to the objects displayed, you must also build ways to show the remaining objects.

You should also consider bandwidth when you are planning sophisticated graphics, multiple frames, or any other HTML structures that create processing overhead.

Consider Browser Limitations

Make sure to consider the limitations of the Web browsers that will support your application. For example, if you plan to use scripting for HTML form validation, your choice of a scripting language may depend on the browser being used to run the application. In addition, since scripting is processed by the Web browser, heavy use of scripting in your application increases the load on the browsers running the application and may decrease performance.

Make sure you consider the browser versions you intend to support, and the size of the typical machine that the browser will run on when planning your application.

Consider Using Existing Application Designs and Templates

Another approach to application development is to use HTML application templates provided with the Unified Messaging SDK as a starting point for a new application or to add Unified Messaging capabilities to existing Web pages.

For example, if you are developing an application with the messaging JavaBeans APIs, you can begin your application design by looking at the included admin and user application. The sample application is a HTML template with calls to the Unified Messaging APIs.

You can copy and modify any or all parts of these application to build a comparable application that meets your needs. To view the source code of any of the HTML application templates for the sample application, go to the application files in the following directory:

```
$ORACLE_HOME/um/templates/um
```

HTML Application Development Process Overview

As you write HTML templates, you may find that your application development is easier if you take the following steps:

1. Design and create regular HTML files as static templates.

   Start by creating a set of industry-standard HTML files that define the graphical user interface and flow of the application. At this point, data is hard-coded, not retrieved from the Unified Messaging Server, and clicking a link or an icon displays another HTML file.

2. View your static templates through personal Web server software.

   So that the links to images and other templates work as they will in the running application, view the static templates through a personal Web server, rather than viewing local files through a Web browser.

   During this step, you can experiment with the look and feel of the application, as well as the execution flow. You can also work with end users, managers, Web designers, or other developers to refine the application before you start building live Unified Messaging functionality.

3. Create the directory structure.

   Create a directory for your application in `$ORACLE_HOME/um/templates`. In your application directory, create separate directories for templates and images, like this:

   ```
   $ORACLE_HOME/um/templates/um

   $ORACLE_HOME/um/templates/images
   ```

4. Change the static URLs to application URLs.

   Start with the first template, which by default has the same name as the application. Change the static URLs, which call other HTML files, into application URLs, which call methods from the JavaBeans APIS and specify the next template that will be displayed.

5. Replace static data with template tags that call the JavaBeans APIs.

   You can then transform the static HTML files into dynamic HTML templates by replacing static data with template tags that call the JavaBeans APIs. For example, in developing an e-mail application, you might begin by displaying a hard coded list of messages and replace this list with template tags that display actual messages stored in the user's inbox on the Unified Messaging server. You can handle repeating data sets or conditional processing with template tags.

6. View your finished application through a Web browser.

   Once you have modified all of the templates, the application is complete. You do not need to compile or link anything.

3

Customizing Unified Messaging APIs

This chapter discusses how to use Unified Messaging administration facilities and how the Unified Messaging API can be used to develop applications. The following topics are included:

Unified Messaging API Overview

Typical Client Application Development

Searching the Directory

The SDK Package

The MS Package

Unified Messaging API Overview

The Unified Messaging SDK provides an Application Programming Interface (API) that provides access to the Unified Messaging server data and functionality. This API is exposed as a set of JavaBeans. This allows developers to use a standard component and object model when developing new applications. It also provides an opportunity for application developers to use other JavaBean components in conjunction with the Unified Messaging JavaBeans.

The Unified Messaging Functional Class Hierarchy

This class hierarchy provides a high-level view of the Unified Messaging classes likely to be most useful to web application developers. It shows the package or sub-package to which each class belongs (oracle.um.sdk or oracle.um.ms). This hierarchy groups the Unified Messaging classes according to functional areas, and does not imply inheritance.

- sdk.Session

- sdk.Audio
- sdk.AdministratorList
- sdk.Settings
- sdk.CustomerNotes
- sdk.Note
- sdk.Directory
- sdk.Address
- sdk.GSMDir
- sdk.GSMAddress
- sdk.MsgStores
- ms.Store
- ms.Folder
- ms.Message
- ms.UMStore
- ms.UMRoot
- ms.UMInbox
- ms.Message
- ms.Folder
- ms.Message
- ms.SearchFolder
- ms.Message
- sdk.PagerDevice
- sdk.NotificationRule
- sdk.SMS
- sdk.SMSMessage
- sdk.Registration
- sdk.Trace

Keep in mind that this is only a partial list. Additional classes exist within the Unified Messaging packages. For a complete description of the Unified Messaging classes, refer to the Unified Messaging Javadoc.

The Unified Messaging API contains one major package: oracle.um.sdk. This package supports both user and administrator functions, including:

Session management, including access to user profiles

- Directory searching
- E-mail manipulation
- SMS management
- Synchronization and notification

This main package has two sub-packages performing specific functions:

- oracle.um.ms, which provides access to the IMAP4 message stores, resolves addresses, and sends and receives e-mail.
- oracle.um.services.ldap, which provides access to the LDAP directory.

The packages are built in a hierarchy and installed at the following location:

```
$ORACLE_HOME/um/lib/um.jar
```

For detailed information about the complete syntax of the Unified Messaging SDK packages, classes, methods, and properties. The Javadoc is available at the following location:

```
$ORACLE_HOME/um/doc/javadoc/index.html
```

The Unified Messaging Java Class Hierarchy

The Java class hierarchy describes how one class inherits from another. This information is important because the inheriting class inherits the methods and properties of the parent class. The following lists describe the class hierarchy of the two Unified Messaging packages: oracle.um.sdk and oracle.um.ms. Most of the classes that a web application developer will use are included in the oracle.um.sdk package. Other supporting packages and sub-packages provide classes that deal with specific, less frequently used tasks. This list describes the class hierarchy of the oracle.um.sdk package:

- Address
- Audio

- Directory
- Fax
- GSMAddress
- GSMDir
- List
- AdministratorList
- CustomerNotes
- MsgStores
- PagerDevice
- SMS
- Note
- NotificationRule
- Registration
- Session
- Settings

This list describes the class hierarchy of the oracle.um.ms package:

- BodyPart
- Folder
- SearchFolder
- UMInbox
- UMRoot
- Message
- Migration
- MultiPart
- Store
- UMStore
- Transport

These hierarchies provide only a partial list. Some classes within the oracle.um.ms package are used by other classes for internal processing. For a complete listing of the class hierarchy, refer to the Unified Messaging Javadoc.

Typical Client Application Development

Developing or customizing a Unified Messaging application will proceed more efficiently for the developer who understands the following:

- The functions of a messaging system that need to be provided
- The specific Unified Messaging classes to use for each task

Some general messaging system functions include:

- Creating and managing a login session
- Retrieving Message Stores
- Retrieving and displaying a folder (e.g. Inbox)
- Retrieving and displaying a message
- Creating and sending a new message
- Searching for message using specified search criteria
- Managing and organizing messages using folders
- Updating various items related to the user, including the personal address book, user profile settings, and notification rules.

These tasks are explained with examples that are derived from the Unified Messaging client. These templates can be viewed, in their entirety, at the following location:

```
$ORACLE_HOME/um/templates/um
```

Before you start customizing or developing your application, be sure you have completed all the post-installation tasks related to the Unified Messaging Applications component.

Creating and managing a login session

An instance of the Session class must be created when a user logs into the system. This is done with the username and password provided by the user. A login page is usually the entry point for the end user in most session-based applications. Here is

the login page [login.jsp] that the Unified Messaging client uses to gather this information:

```
<html>
<head>
 <title>Unified Messaging Login</title>
 <link rel=stylesheet type="text/css" href="umstyle.css">
</head>

<body bgcolor="#FFFFFF">
<br>

<form action=<%= response.encodeUrl("checkStatus.jsp") %>
method=post>
 <table border=0 cellpadding=0 cellspacing=0
        align=center width=640 background="/images/umlogo.gif">
  <tr>
   <td><img src="/images/blank.gif" height=350 width=1></td>
   <td>
    <table border=0 cellspacing=0 cellpadding=0
           align=right width=45% background="">
     <tr><td height=30> </td></tr>
     <tr><td>
      <font face="Arial,Helvetica" size=2><b>Username or Phone
Id</b></font>
     </td></tr>
     <tr><td><input name=username type=text size=16
maxlength=20></td></tr>
     <tr><td>
      <font face="Arial, Helvetica" size=2><b>Password</b></font>
     </td></tr>
     <tr><td><input name=password type=password size=16
maxlength=20></td></tr>
     <tr><td> </td></tr>
     <tr><td>
      <input type=submit value=Login>
      <input type=reset value=Reset>
     </td></tr>
    </table>
   </td>
  </tr>
 </table>
</form>
<center>
 <a href="http://www.javasoft.com/products/jsp/index.html">
```

```
  <font face="Arial, Helvetica" size=2>On Java Server Pages
1.0</font></a>
</center>
<%
 String errorMsg = request.getParameter("errorMsg");
 if (errorMsg != null)
  out.println("<p><center class=errMsg>" + errorMsg + "</center>");
%>
</body>
</html>
```

You can see that this page is essentially a form with two fields for the username and password. When a user logs into the system, there may be other preparations required before letting them proceed with their login session. Aside from validating their information, special steps may want to be taken based on things like the amount of quota the user has left, or whether this user is logging in for the very first time. For this reason, the form action in the Unified Messaging client login page will call the "checkStatus.jsp" template with the results of this form.

Before we discuss this other template, we should also note that at the bottom of this template there is also some JSP code that checks to see if an error parameter was passed in the URL to this page. This general mechanism can be used to allow a template to react differently based on URL parameters such as error cases or to even allow a template to work as a "traffic cop" and direct traffic to other templates. In this case, we merely print out the value of the error message parameter and use stylesheets to make it appear in an appropriate manner. The code for checkStatus.jsp has no HTML inside it.

```
<%
String nextURL = response.encodeUrl("main.jsp");
try {

 String uname = request.getParameter("username");
 String pword = request.getParameter("password");

 oracle.um.sdk.Session umSession = new

 oracle.um.sdk.Session(uname, pword, "", "");

 session.putValue("umSession", umSession);

 oracle.um.sdk.Settings umSettings = umSession.getSettings();

 String regStatus = (String) umSettings.get("status");
```

```
 if (regStatus.equals("I")) {
   nextURL = response.encodeUrl("inactive.jsp"); // user is not activated
yet.
 }
 if (umSettings.isQuotaUsed()) {
   nextURL = response.encodeUrl("quotaExceeded.jsp"); // quota
exceeded
 } else if (umSettings.isQuotaCheckpointExceded()) {
   nextURL = response.encodeUrl("quotaWarning.jsp"); // warning
quota is over 80 percent.
 }

} catch (Exception e) {
 nextUrl = "login.jsp?errorMsg=" +
java.net.UrlEncoder.encode(e.toString());
}

response.sendRedirect(response.encodeRedirectUrl(nextUrl));

%>
```

This template will always redirect to another template. In a typical login scenario, this template would redirect to main.jsp. If there are any special cases, we may go to a different template based on each case. Almost the entire template is coded inside a try-catch statement, so if there are any exceptions, the template we would be redirected to is login.jsp with the error string passed as a parameter to the URL.

The main purpose of this template is to create an instance of a Session object by passing in the username and password to the constructor. The Session class serves a special function in the Unified Messaging API, acting as a factory class that provides access to other Unified Messaging classes. You can think of the Session class as the highest entity in the Unified Messaging functional hierarchy, because the other classes often require some information that has been processed and stored by the Session class.

If a session is created successfully, the Unified Messaging client will 'put' the Unified Messaging session into the JSP session for later reference and then gets the settings for this user's session. This information will include things like the status of the user's account. The value of user status can be one of the following:

- Administrator

  Administrators require a more powerful set of functions than other users, such as the ability to create and delete accounts. These functions are specified in the

Session object. Administrative privileges are discussed in further detail in the "Administering Unified Messaging" section.

- First-time user (inactive user)

  First-time users, also known as "inactive users," may need to go through an activation process including both authentication (username and password) and customization (user profile). The Registration class provides authentication by sending and verifying a registration code to the user's pager. The user can then specify a more appropriate username and password to replace the ones generated by the system.

- Active user

  An active user is a validated Unified Messaging user who has successfully activated a Unified Messaging account by selecting a personalized account name and password.

What happens next depends on whether the user is active or inactive. For inactive users, the client is redirected to the inactive.jsp template which performs the following steps:

1. Display the welcome message.
2. Retrieve the registration code.
3. Verify the registration code.
4. Set up the custom password and e-mail alias.
5. Create a messaging account.

If the user is active:

1. Check quota usage.
2. If the user is approaching the quota limit for messages, redirect the client to an notification page before allowing them to continue.

After this point, the user will proceed into the application where the Unified Messaging client will open message stores and provide an initial display.

Retrieving Message Stores

In order to get access to Unified Messaging messages the client must open message stores. A message store contains folders such as an inbox or wastebasket. The Unified Messaging system supports three types of message stores:

- Email server

- Voice mail server
- Facsimile server

A user can even have more than one message store for each type of message. For example, a user could have one e-mail message store, two voice mail message stores, and two facsimile message stores. In this case, each of the five message stores corresponds to its own e-mail, voice mail, or facsimile address. Each message store has its own inbox, where all incoming messages for that message store are initially placed. Multiple message stores may also be aggregated into one Unified Messaging inbox, to provide a single, unified view of messages from the multiple message stores. You should use the Store class to represent each message store in your Unified Messaging system.

In this code snippet from main.jsp, the Unified Messaging client retrieves the list of message stores and retrieves the first message store from the list (which is always the aggregated Unified Messaging inbox).

```
oracle.um.sdk.MsgStores umMsgStores =
umSession.getUMMsgStores();
oracle.um.ms.UMStore umStore = (oracle.um.ms.UMStore)
umMsgStores.getElement(0);
```

Retrieving and Displaying Folders (e.g. Inbox)

Folder objects are contained within a Store object, as shown in the following hierarchy:

```
oracle.ms.Store
oracle.ms.Folder
oracle.ms.Message
```

A folder can contain other folders and messages.

To demonstrate a retrieval of a folder from a message store, here is some code from msg_list_con.jsp:

```
String newFolder = request.getParameter("newFolder");
oracle.um.ms.Folder f = null;
if (newFolder != null)
{
 oracle.um.ms.UMStore umStore = (oracle.um.ms.UMStore)
session.getValue("umStore");
 f = umStore.getFolder(newFolder);
 f.preFetch();
```

```
 session.putValue("currFolder", f);
}
```

The code gets the value of the parameter "newFolder" from the requesting URL. If the value is not null, it retrieves the umStore object from the session, and gets the folder from the store. It then prefetches some information about the first messages in the folder and the folder is "put" into the session for later reference.

Once the folder is available, displaying the list of contents can range from fairly simple to fairly complex based on what information you would like to display, and how you would like the user to interact with the content. Basic to any display would be the list of messages and their attributes. To retrieve information about these messages, you would call a method very similar to the one to retrieve a folder from a store:

```
m = f.getMessage(i);
```

Where f is the folder and I is the index of the message in the folder. With the message in hand, retrieving the attributes is relatively straightforward. Placing this code in a loop makes the list of messages easy to come by. Here is an example of the basic aspects to displaying a folder list taken from parts of the msg_list_con.jsp template:

```
.
.
.
<form name=msgList>
<%
 oracle.um.ms.Message m = null;
 String msgType = null;
 int msgSize = 0;
 boolean bg = false;
 for (int i=msgNumOffset; i<lastMsgNum; i++)
 {
  m = f.getMessage(i);
  msgType = m.getFolder().getStore().getMsgStoreType();
  msgSize = m.getSize() / 1024;
...
%>
.
.
.
lots of interesting formatting and javascript mixing JSP and HTML
.
```

```
.
.
<% } %>
</form>
```

Retrieving and Displaying Messages

Like folders, Message objects contain other objects, as shown in the following hierarchy:

```
oracle.ms.Message
oracle.ms.MultiPart
oracle.ms.BodyPart
```

A Message object can contain the following objects:

- At most, one MultiPart object (simple messages may not have a MultiPart object)
- One or more BodyPart objects, which can represent any of the following:
  - The email text message
  - An attached message
  - An attached file, such as an audio file

In addition to retrieving the content of the e-mail itself, you may want to retrieve various parts of the message header, which includes information such as:

- Sender
- Recipient
- Date
- Subject
- Message status

These parts of the message header are represented as properties of the message, so to retrieve them, use the appropriate getXXX method on the Message object, as shown in the following example:

```
String subj = msg.getSubject();
```

A simple message object may contain only the message body; that is, the e-mail text itself. It may not contain a multipart object at all. In this case the body of the

message can be retrieved using the getBody method, as shown in the following example:

```
String content = msg.getBody();
```

In more complex messages, a message may contain one MultiPart object. This MultiPart object is a container for one or more BodyPart objects that may represent different components of a complete message such as:

- Email message text
- Attached files
- Attached messages

A complex message may thus contain several BodyPart objects, representing the e-mail text and attachments. In this case, you will need to use the MultiPart and BodyPart classes to access the message text and other attachments of a message.

MIME

E-mail was originally designed to transmit ASCII text. MIME, or Multipurpose Internet Mail Extensions, defines a series of file types that allow mail systems to transmit non-ASCII files, such as formatted text files, image files, and sound files. MIME adds a header to the file that specifies the type of data contained. Sample MIME types include:

text/HTML, text/plain, application/pdf, image/gif, image/tiff, and audio/wav. You need to know the MIME type of a BodyPart or Message object to determine how to handle its content. To ascertain the type of the you can call the isMimeType method as shown in the following example:

```
bp = mp.getBodyPart(i);
if (bp.isMimeType("text/html"))
{…}
```

If a Message object has a MIME type of multipart/* it contains a MultiPart object. If the content of this message is not a MultiPart object, the MIME type of its contents will be a type other than multipart/*, such as text/*. In this case, the is MimeType("multipart/*") method would return the value false.

A Complex Message Example

Here's an example of a more complex message. This is the "containing" relationship structure:

- Message object
- MultiPart object
- Body Part #1 - The actual e-mail text file, MIME type text/plain.
- Body Part #2 - An attached document file, MIME type application/pdf.
- Body Part #3 - An attached photograph file, MIME type image/gif.

Note that a BodyPart object can be an entire e-mail message. For example, assume the following scenario:

- Manager Magee sends Smith a message with the subject line, "Departmental Meeting."
- Smith then forwards Magee's message to Jones with the subject line, "Important Meeting Scheduled."

To display the forwarded message to Jones, you would find that his message object is multipart and that one of the BodyParts was of MIME type message/*. Opening the content of that BodyPart would get you a message object that contained a BodyPart that had the actual e-mail message from Magee.

Thus, a BodyPart object can contain another whole Message object. This Message object, in turn, can be a multipart Message object that contains more BodyPart objects. This series of objects containing objects can create a recursive structure that could be many levels deep.

Displaying Messages

Be sure to consider the recursive nature of e-mail messages when you plan how your web application will display e-mail messages. In most cases, you will probably not want to display the content of all the BodyPart objects in a single window. This is particularly true for those BodyParts that represent attachments. Providing an access point to these items in the form of a link is usually best. In general displaying all the BodyParts could make for a complicated and difficult-to-read display, because you cannot predict in advance the number of levels of BodyPart objects a message may contain.

The Unified Messaging client displays attachment icons in a separate browser frame from the content. This allows the list of attachments to always be visible. This also requires an "original message" icon to take the user back to the content of the message. This code is in message_view_header.jsp and a snippet follows:

```
<!-- list attachment and the original msg icons for emails -->
<%
```

```
 if (msg.isMimeType("multipart/mixed"))
 {
  oracle.um.ms.MultiPart mp = (oracle.um.ms.MultiPart)
msg.getContent();
  int numbp = mp.getCount();
  if (numbp > 1)
  {
   out.println("<table border=0 cellspacing=0 cellpadding=0>");
   out.println("<tr><td width=15><img src='/images/blank.gif' width=15
height=5 border=0></td><td class=attach><a
href='javascript:load_content();'><img src=\"/images/MessRead.gif\"
alt=\"Message Content\" border=0></a></td>\n<td class=attach
width=5><img src='/images/blank.gif' width=5 height=5
border=0></td>");
   out.println("<td class=attach width=100% >");
   for (int i=1; i<numbp; i++)
   {
    oracle.um.ms.BodyPart bp = mp.getBodyPart(i);
    if (bp.isMimeType("message/rfc822"))
    {
     oracle.um.ms.Message bpcon = (oracle.um.ms.Message)
bp.getContent();
     String subj = bpcon.getSubject();
     if (subj.trim().equals(""))
      subj = "No Subject";

     out.println("<a href='javascript:open_att(["+i+"],true);'><img
src=\"/images/Attch_w_Merss.gif\" alt=\"Open Attachment\" border=0>"
+subj+ "</a><img src='/images/blank.gif' width=5 height=5 border=0>");
    }
    else
    {
     String filename = bp.getFileName();
     if (filename.trim().equals(""))
      filename = "Anonymous";
     out.println("<a href='javascript:open_att(["+i+"],false);'><img
src=\"/images/Attch.gif\" alt=\"Open Attachment\" border=0>"
+filename+ "</a><img src='/images/blank.gif' width=5 height=5
border=0>");
    }
   }
   out.println("</td><td class=attach nowrap><img src='/images/blank.gif'
width=5 height=5 border=0></td></tr>");
   out.println("<tr><td width=100% colspan=5><hr></td></tr></table>");
  }
```

```
}
```

Converting Audio and Facsimile Files

The Unified Messaging SDK provides audio conversion from g726 format to both WAV and RealAudio formats. This conversion is needed because voice mail systems use g726 format and facsimiles use the TIFF format. Neither of these formats is compatible with standard web browsers. Web browsers will work with WAV and GIF formats by default and now usually come prepackaged with a RealAudio plugin. Both audio and facsimile conversions require a three-step process:

1. First create an instance of the Audio and Fax objects for the user's session (during login time usually).
2. When you need to handle a voice mail or facsimile, call prepareAudio() or prepareFax() to ask the server to perform the necessary conversion.
3. Finally, display the converted audio file or fax images.

After the customer logs in successfully it is a good idea to create an instance of the Audio and Fax objects. You can reuse a single instance of these objects throughout an entire session. Here's an example of creating a Fax object:

```
oracle.um.sdk.Fax fax = umSession.getFax();
```

Once you have these object instances available, you need to call their prepareXXX method before giving the end user the data. Here are the function parameters for the prepareAudio and prepareFax methods of the Audio and Fax classes found in the oracle.um.sdk package:

```
public void prepareAudio(BodyPart part, String title, String author, String
copyright)
public void prepareFax(BodyPart part)
```

Parameter	Description
part	The body part that contains the voice data.
title	The title of the voice data.
author	The author of the voice data.
copyright	The copyright information regarding the voice data.

Once prepared, the data can be made available to the end user. In the case of audio, you can choose either RealAudio or WAV format by using either the method

getRealcontent() or getWavecontent() on the Audio object. For Fax data, you can use the getGifcontent() method on the Fax object.

Creating Messages

Creating a new message via a compose window is very simple. The JSP template for creating a new message is essentially a form with various attributes about the new message. Here are the more relevant parts of the code that is used in the Unified Messaging client compose window:

```
<form name=composeEmailForm enctype="multipart/form-data"
        action="<%= response.encodeUrl("msg_comp_email_send.jsp")
%>"
        target=msgRes method=post>
…

  <td class=fieldData colspan=3 nowrap><input type=text name=to
size=35 value="<%= to %>">
…
  <td class=fieldData colspan=3 nowrap><input type=text name=cc
size=35 value="<%= cc %>"></td>
  <td class=fieldData colspan=3 nowrap><input type=text name=subject
size=35 value="<%= subject %>"></td>
…
<td class=fieldData nowrap><select name=fcc>
<%
try {
 oracle.um.ms.UMStore umStore = (oracle.um.ms.UMStore)
session.getValue("umStore");
 oracle.um.ms.Folder rootFolder = umStore.getRoot();
 out.println(listSubFolders(rootFolder, ""));
} catch (Exception e) {
 out.println(e.getMessage());
}
%>
  </select></td>
…
  <td class=fieldData colspan=3 nowrap><input type=file
name=fileAttach size=35></td>
…
  <td class=fieldData colspan=5><textarea name=body cols=50
rows=12><%= body %></textarea></td>
…
</table>
```

```
</form>
…
```

The contents of the form are filled with appropriate data as needed (e.g. in a message reply or forward). In the Unified Messaging client, the form will call the msg_comp_email_send.jsp template where the createMessage method is invoked as follows:

```
umSession.createMessage(to, cc, subject, body, fcc, msg, fds);
```

Parameter	Description
to	The destination address(es)
cc	Carbon copy address(es)
subject	The subject of the message
body	The content of the message (the "text")
fcc	A folder to copy the new message to ("Folder Carbon copy")
msg	A Message object that this new message will contain (e.g. the forwarded message)
fds	A oracle.um.util.FileStreamDataSource that contains the attachments (see msg_comp_email_send.jsp for an example)

Searching for Messages

The searchFolder method for retrieving message objects with specific values is:

```
public searchFolder search(String[] searchAttribute, String[]
searchOperator,String[] searchValue,String startFolderName, boolean
nestedFolder,String compoundOperator)
```

Parameter	Description
searchAttribute	The portion of the message to search, specified as a String array. The value may be BODY, FROM, TO, RECEIVED, SUBJECT, PRIORITY, SIZE, or READ.
searchOperator	The operator to use in the search, specified as a String array. The value may be EQUAL_TO, NOT_EQUAL_TO, CONTAIN, NOT_CONTAIN, GREATER_THAN, or LESS_THAN.
searchValue	The specific value to search for, specified as a String array. The value may be one of the following constants: ALL, YES, NO, LOW, MEDIUM, or HIGH; or any user-specified string, such as a name or date.
startFolderName	The name of the folder from which to begin the search.
nestedFolder	Whether to search through all nested folders, or to search only in the current folder. The value may be TRUE or FALSE. FALSE means "search only the folder specified in startFolderName."
CompoundOperator	If multiple search operators have been listed, specifies how to combine the search terms. The value may be AND or OR.

To specify the search criteria, set three parameters:

- searchAttribute
- searchOperator
- searchValue

The following table shows how these three parameters can be used together to specify search criteria:

searchAttribute	searchOperator	searchValue
BODY	CONTAIN	User-defined string
	NOT_CONTAIN	
FROM	CONTAIN	User-defined string
	NOT_CONTAIN	
TO	CONTAIN	User-defined string
	NOT_CONTAIN	

RECEIVED	LESS_THAN	User-defined string
	GREATER_THAN	
	EQUAL_TO	
	NOT_EQUAL_TO	
SUBJECT	CONTAIN	User-defined string
	NOT_CONTAIN	
PRIORITY		LOW, MEDIUM, HIGH
READ		YES, NO
SIZE	LESS_THAN	User-defined string representing an integer
	GREATER_THAN	
	EQUAL_TO	
	NOT_EQUAL_TO	

As in message composition, the Unified Messaging client uses a form to retrieve the search criteria from the user:

```
<form name=searchMsgForm
       action="<%= response.encodeUrl("search_message.jsp") %>"
       target=Result method=post>
```

The real work happens in search_message.jsp where the call to the search method is made:

```
oracle.um.ms.SearchFolder sf =null;
…
sf = s.search(a,o,v,folder,sub,matchStr);
```

The vector of attributes, operators, and values, is passed in with the folder to search in, a flag for the subfolder search, and the type of compound operation to perform. This call then returns a SearchFolder object that can be treated in a manner similar to a regular messaging folder object. This makes it easy for the Unified Messaging client to display the results of the search using the same approach; as a message list.

Searching the Directory

If you want to search a directory to retrieve information about addresses, use the Session class with the GSMDir and Directory classes. (GSM stands for Global System for Mobile communications).

- The GSMDir object represents a public directory that can be searched by any user.
- The Directory object represents the user's private directory, which can only be searched by the owner of the directory.

To search the content of a directory, you must first create one of these directory objects using the Session object. You can create both GSMDir and Directory objects using the Session object's getDirectory method. The getDirectory method is overloaded; that is, you can use the same method for different purposes, depending on the number and type of parameters used in calling the method. In this case, you pass six parameters to getDirectory to retrieve a GSMDir object and five parameters when you wish to retrieve a Directory object.

Searching for Users using the GSMDir Object

The parameters for the getDirectory method when retrieving a GSMDir object are as follows:

```
public GSMDir getDirectory (String type, String name, String city, String
fax, String phone, String operation)
```

Parameter Description

The following table describes the parameters for the getDirectory method:

type	The type of the search (must be "GSM")
name	The name of the person to search for
city	The city to search for
fax	The fax number to search for
phone	The phone number to search for
operation	The operation of the search

A form is used to gather these parameters from the user as in message composition. This form then calls the search_directory.jsp template to do the actual search. Here is the way the Unified Messaging client instantiates a GSMDir object when performing a search in search_directory.jsp:

```
oracle.um.sdk.GSMDir gsmDir = (oracle.um.sdk.GSMDir)
umSession.getDirectory("GSM",name,city,fax,phone,"AND");
```

Searching for Address Book Entries Using the Directory Object

The parameters for the getDirectory method when retrieving a Directory object is:

```
public Directory getDirectory (String type, String name, String
startValue,
String endValue, String operation)
```

Parameter	Description
type	The type of the search (ALL, PRIVATE_ALIAS, or PRIVATE_GROUP)
name	The name of the address element used for the search operation
startValue	The start value of the search
endValue	The end value of the search
operation	The operation of the search

A form is used to gather these parameters from the user as in message composition. This form then calls the search_addrbook.jsp template to do the actual search. Here is the way the Unified Messaging client instantiates a Directory object when performing a search in search_addrbook.jsp:

```
oracle.um.sdk.Directory addrDir = (oracle.um.sdk.Directory)
umSession.getDirectory(addressType,addressName,searchValue,searchValue,searchOpe
ration);
```

For example, to search for private aliases whose FULL_NAME attributes start with the letters "a" to "g", you would pass PRIVATE_ALIAS for type of search, FULL_NAME for the name of the address element to search, "a" as the startValue, "g" as the endValue, and START as the operation. For more details regarding options and variations, please refer to the reference or the javadoc.

The SDK Package

The `oracle.um.sdk` package is used to perform high-level tasks, such as creating new user accounts, creating notes, and converting audio and facsimile files. The classes of the `oracle.um.sdk` package use the classes of the `oracle.um.ms` package to carry out some of these tasks.

Address Class

The Address class represents an entry in the Unified Messaging user's address book. Use the Address class to represent the user's private aliases and distribution lists. A private alias contains the contact information for any person. A private distribution list is a list of aliases. Addresses are created using `Session.createAddress`.

The following tables list the attributes and methods of the Address class:

Attribute	Description
PRIVATE_ALIAS	Specifies that the search is for PRIVATE_ALIASes only.
PRIVATE_GROUP	Specifies that the search is for PRIVATE_GROUPs (distribution lists).

Method	Description
delete()	Deletes this address from LDAP.
get(String Directory)	Gets the value of a given property name (dynamic properties).
getId()	Gets this address id.
getMembers()	Returns a directory list with the aliases in this DL.
getType()	Gets this address type Returns either PRIVATE_ALIAS or PRIVATE_GROUP.
insert(String[], String[])	Inserts and array of aliases to the current object's distribution list.
next()	Gets the next address in a directory search.
previous()	Gets the previous address in a directory search.
remove(String[], String[])	Removes aliases from the current object's distribution list.
update(String, String, String[], String[])	Updates an address with new values.

AdministratorList Class

The AdministratorList class is used exclusively by the Unified Messaging administrator. An AdministratorList object contains a list of Unified Messaging accounts, retrieved by performing a search. The administrator can use this list to log into a particular account or to change account information.

The following table lists the method of the AdministratorList class:

Method	Description
getList(int)	Gets the settings of a UM account stored in this[index].

Audio Class

The Audio class handles conversion of g726 and WAV formatted files from the message store as either WAV files or files in RealAudio format. Conversion requires two steps: first `prepareAudio` starts a background process for retrieving and converting the data. Then the file is played by the Web browser using `getWavecontent` or `getRealcontent`.

The following table lists the methods of the Audio class:

Method	Description
createMetaFile()	Creates a meta file for the RealAudio media.
getRealcontent()	Gets the meta information for the temporary RealAudio file.
getRealcontentMIME()	Gets the MIME type of the RealAudio.
getRealfile()	Get the filename for the RealAudio data.
getServer()	Gets the location of the RealAudio server.
getWavecontent()	Gets the content of the Wave audio file.
getWavecontentMIME()	Gets the MIME type of the Wave data.
getWavefile()	Gets the filename for the Wave audio.
prepareAudio(BodyPart, String, String, String)	Starts background processes.
setMIMERealPlayer()	Sets the RealAudio MIME type for the external browser player.
setMIMERealPlugin()	Set the RealAudio MIME type for the browser plugin player

Directory Class

The Directory class represents a set of addresses retrieved by performing a search on the customer's address book. The following tables list the attributes and methods of the Directory class:

Attributes	Description
ALL	Searches for both aliases and groups.
OP_CONTAINS	Searches for addresses 'contain' the specified value.
OP_EQUAL	Searches for addresses 'equal' to the specified value.
OP_MEMBER	Searches for addresses in the specified distribution list.
OP_NOT_MEMBER	Searches for addresses and / or groups not in the specified distribution list.
OP_START	Searches for addresses 'start' with the specified value.
PRIVATE_ALIAS	Searches for PRIVATE_ALIAS only.
PRIVATE_GROUP	Searches for PRIATE_GROUP (distribution lists) only.

Method	Description
getCount()	Gets the maximum number of records.
getElement(int)	Gets an address in the list.
getIndex()	Gets the current index.
isTooMany()	Indicates whether the search results in too many hits.

Fax Class

The Fax class is for on-the-fly facsimile image conversion. The `prepareFax` method converts a facsimile image to GIF format. Then the `getGifContent`, `getGifContentMIME`, and `getGifFile` methods are used to retrieve and display the image. Currently, this class takes only two types of images: GIF files and TIFF files. Implementation for other types of images requires modification to the `prepareFax` method.

The following table lists the methods of the Fax class:

Method	Description
getGifContent()	Gets the content in GIF format.
getGifContentMIME()	Gets the Content MIME Type of the bodypart.
getGifFilename()	Returns the filename property.
prepareFax(BodyPart)	Sets a content of a BodyPart.

GSMAddress Class

The GSMAddress class represents any person's entry in an LDAP server, whether the person is a Unified Messaging customer or not. (GSM stands for Global System for Mobile communications.)

The following table lists the methods of the GSMAddress class:

Method	Description
get(String)	Gets the value of a property by name.
getCustomerName()	Gets the customer name.
getFaxNumber()	Gets the fax number.
getPhoneId()	Gets the value of phone_id.
next()	Returns the next GSM address from the GSM directory.
previous()	Returns the previous address from the current GSM directory.
set(String, Object)	Sets the value of a property by name.

GSMDir Class

The GSMDir class represents a set of addresses retrieved by performing a search on the GSM (public) address directory. Just as the Directory class contains a list of private addresses, such as personal address book, the GSMDir class contains a list of public addresses, like a corporate directory.

The following tables list the attributes and methods of the GSMDir class:

Method	Description
get(int)	Gets a GSM address specified by the index.
get(String)	Gets a property by name (dynamic properties).

Method	Description
getCount()	Gets the number of records that can be retrieved.
getElement(int)	Gets a GSM address from this.
getIndex()	Gets the current index.
getSearchOperator()	Gets the search operator.
isToomany()	Indicates whether the query returns too many results.
reset()	Resets the list.
set(String, Object)	Sets a property by name (dynamic properties)
setCity(String)	Sets the city name to search for.
setFaxNumber(String)	Sets the fax number to search for.
setLastName(String)	Sets the last name to search for.
setListAmount(int)	Sets the size of the memory buffer.
setPhoneId(String)	Sets the phone number to search for.
setSearchOperator(String)	Sets the search operator.

List Class

The List class is an Abstract class, used to define many of the common functions used by other classes that contain lists of objects, such as AdministratorList and CustomerNotes.

The following table lists the methods of the List class:

Method	Description
getCount()	Gets the number of records to retrieve.
getElement(int)	Gets an element in the list.
getIndex()	Gets the current index.
reset()	Resets the list.
setListAmount(int)	Sets the size of the memory buffer.

MsgStores Class

The MsgStores class retrieves all the message stores defined for a Unified Messaging user. This class can be created by the Session class. Each element in a `MsgStores` object is a `Store` object of the message store available to this customer.

The following table lists the method of the MsgStores class:

Method	Description
toString()	Returns a string of all message stores.

Note Class

The Note class represents a customer note object, consisting of the following parts: NoteID, Subject, Text, and Signature. Notes are primarily used by Unified Messaging administrators to record reminders related to a specific customer.

The following table lists the methods of the Note class:

Method	Description
delete(String)	Deletes a note given the note id.
get(String)	Gets the value of a property by name.
getNoteId()	Gets the note id.
getSignature()	Gets the signature.
getSubject()	Gets the subject.
getText()	Gets the text.
next()	Returns the next note.
previous()	Returns the previous note.
set(String, Object)	Sets the value of a property by name.
setCreateDate(String)	Sets the date the note is created.
setCustomerId(String)	Sets the customer id.
setNoteId(String)	Sets the note id.
setSignature(String)	Sets the signature, or the author of the note.
setSubject(String)	Sets the subject of the note.
setText(String)	Sets the text of the note.

Method	Description
update(String, String, String, String)	Updates the attributes of the current object.

NotificationRule Class

The NotificationRule class administers the creation, deletion, and maintenance of notification rules. Rules are used to set reminders, such as which messages will trigger user notification via a pager.

The following table lists the methods of the NotificationRule class:

Method	Description
delete()	Deletes this rule.
get(String)	Gets the value of a property by name.
set(String, Object)	Sets the value of a property by name.
update(String[], String[])	Updates the rules.

PagerDevice Class

The PagerDevice class manages the notification rules for a pager device. The rules can be retrieved, modified, updated, enabled, and disabled.

The following table lists the methods of the PagerDevice class:

Method	Description
get(String)	Gets the value of a property by name.
getRule(int)	Gets the notification rule for the current device.
setEnabled(String[])	Enables the given list of rules and disables the rest for the device.

Registration Class

The Registration class is used for inactive users when they want to register and create a Unified Messaging account. Using the Registration class, Unified Messaging sends a randomly generated registration code to the user via a pager. After this code has been successfully identified, a Unified Messaging account will be created. This account will use a custom password and e-mail alias. The Registration class can be retrieved though the Session class.

The following table lists the methods of the Registration class:

Method	Description
checkRegistrationCode(String)	Verifies the registration code from the user.
createAccount(String)	Creates a Unified Messaging account given that the password is known.
createAccount(String, String)	Creates a Unified Messaging account given that the phone_id is known.
createAccount(String, String, String)	Creates the Unified Messaging account.
createRegistrationCode(String)	Creates a registration code.

SMS Class

The SMS class administers the creation and sending (to one receiver) or broadcasting (to all Unified Messaging users) of Short Message Service (SMS) messages.

The following table lists the methods of the SMS class:

Method	Description
broadcast(String)	Broadcasts an SMS message to all active UM users.
send(String, String)	Sends an SMS message to the specified receivers.

SMSMessage Class

The SMSMessage class contains a list of SMS `Message` objects retrieved by performing a search.

The following table lists the methods of the SMSMessage class:

Method	Description
getDate()	Gets the date of the message.
getError()	Gets any error that may have occurred.
getMessage()	Gets the body of the message.
getPhone()	Gets the phone associated with the message.
getStatus()	Gets the status of the message.

Method	Description
getType()	Gets the type of the message.
next()	Gets the next SMS message.
previous()	Gets the previous SMS message.

Session Class

The Session class is the main class of the Unified Messaging SDK. A Session object must be instantiated before a user can connect to Unified Messaging. The Session object registers the user, creates a new Unified Messaging account, creates e-mail messages, creates user notification rules, and prepares audio and facsimile objects for conversion. The Session class also acts as a factory class for other classes developed for use with the Unified Messaging utilities. For example, it starts an SMS connection and lists the pager devices for the account.

The following tables list the attributes and methods of the Sessions class:

Attribute	Description
ALL	Valid searchOperation for searchAttribute = READ.
AND	valid compoundOperator values for search() function .
BODY	The message body.
CONTAIN	valid searchOperation for searchAttribute=SUBJECT,BODY,FROM,TO.
EQUAL_TO	Valid searchOperation for searchAttribute=SENT,RECEIVED,and SIZE.
fax	The Fax object; only one is needed.
FROM	Message from name.
LOW	Valid searchOperation for searchAttribute=PRIORITY.
PRIORITY	Message Priority.
READ	Message read.
RECEIVED	Received Date.
SENT	Sent date.
settings	The settings for this session.
SIZE	Size of message.

Attribute	Description
SUBJECT	Subject of the message.
TO	Message sent to name.

Method	Description
close()	Closes the session.
createAddress(String, String, String, String[], String[])	Creates an address both in the UM Database and in InterOffice.
createBroadcastMessage(String, String, String, String, FileStreamDataSource[])	Creates and sends a Broadcast message.
createMessage(String, String, String, String)	Creates and sends a message.
createMessage(String, String, String, String, FileStreamDataSource[])	Creates and sends a message.
createMessage(String, String, String, String, Message)	Creates and sends a message.
createMessage(String, String, String, String, Message, FileStreamDataSource[])	Creates and sends a message.
createMessage(String, String, String, String, String, Message, FileStreamDataSource[])	Creates and sends a message.
createMessage(String[], String[])	Creates a message and either sends or saves it.
createMessage(String[], String[], FileStreamDataSource[])	Creates a message and either sends or saves it.
createMessage(String[], String[], Message)	Creates a message and either sends or saves it.

Method	Description
createMessage(String[], String[], Message, FileStreamDataSource[])	Creates a message and either sends or saves it.
createNote(String, String, String, String)	Creates a note for a customer.
createNotificationRule(String, String, String[], String[])	Creates a new notification rule attached to the given user and device id.
createNotificationRule(String, String[], String[])	Creates a new notification rule to the specified device id.
createUser(String, String, String, String, String)	Creates a new user.
createUser(String[], String[])	Creates a new user.
deleteUser(String)	Deletes a user from the UM system.
getAddressByAlias(String)	Gets an address by alias.
getAddressById(String)	This method is no longer supported.
getAddresses()	Gets all addresses (aliases and DLs) created by the current user.
getAdministratorList(String, String, String)	Gets all users, inactive and active, matching the search criteria.
getAudio()	Gets the Audio object.
getClassId()	Public version property.
getDirectory(String, String, String, String, String)	Gets the search result as a Directory object.
getDirectory(String, String, String, String, String, String)	Gets the result from the GSM search.
getDirectoryContext()	Gets the directory context for this session.
getdomainQualifier()	Gets the domain qualifier.
getFax()	Gets the Fax object.
getLanguages()	Gets the Languages object.
getMsgStores()	Gets all the message stores for this user.
getMsgStores(String)	Gets all the message stores for another user.
getNoteById(String)	Gets a customer note by the note id.

Method	Description
getNotesByCustomer(String)	Gets all notes given the customer id.
getNotificationRuleById(String, String)	Gets the notification rule given a device id and the sequence id.
getPagerDevice(String, String)	Gets a pager device given the pager id and the customer id.
getPagerDeviceById(String)	Gets a pager device given an identifier.
getPagerDeviceById(String, String)	Gets a pager device given an identifier and a rule filter.
getPersonByUID(String)	Gets a person by the user id.
getRegistration()	Gets a registration object to identify and create the actual account.
getSettings()	Gets the settings for the current user.
getSettings(String)	Gets the settings of another user.
getSMS(String, String)	Gets a list of SMS messages for a specified customer.
getSMSConnection()	Gets the SMS connection.
getSMSMessage(String, String, String)	Gets a specific SMS message for the given parameters.
getUMMailSession()	Gets the external mail session.
getUMMsgStores()	Gets the UM's view of the message stores for this user.
getUsername()	Gets the account username.
isAdministrator()	Indicates whether the current user is an administrator.
saveMessage(String, String, String, String, String)	Create and saves a message.
saveMessage(String, String, String, String, String, FileStreamDataSource[])	Create and saves a message.
saveMessage(String, String, String, String, String, Message, FileStreamDataSource[])	Creates and saves a message. Used to store message in folders without sending them.
saveMessage(String[], String[])	Creates and saves a message. The FCC attribute should be specified to a folder.

Method	Description
saveMessage(String[], String[], FileStreamDataSource[])	Creates and saves a message. The FCC attribute should be specified to a folder.
saveMessage(String[], String[], Message)	Creates and saves a message.The FCC attribute should be specified to a folder.
saveMessage(String[], String[], Message, FileStreamDataSource[])	Creates and saves a message. The FCC attribute should be specified to a folder.
search(String[], String[], String[], String, boolean, String)	Searches the UMStore for messages matching the specified criteria.

Settings Class

The Settings class exposes the settings of the user for the current session. The settings can be retrieved and updated as necessary.

The following table lists the methods of the Settings class:

Method	Description
changePassword(String, String)	Allows a user to change their password.
get(String)	Gets the value of a property by name. Properties are found in the um_personal_profile table.
getAvailableGSMSearch()	Gets the avaiable_gsm_srch table attribute.
getClassId()	Public version property.
getDecPassword()	Gets the password in plaintext.
getDomain()	Gets the InterOffice domain.
getFaxCoverpage()	Gets the fax_coverpage property.
getForwardFax()	Gets the forward_fax property.
getForwardMail()	Gets the forward_mail property.
getHashtable()	Gets the hashtable as a string - for debugging.
getPaging()	Gets the paging property
getQuotaInProcent()	Gets the percentage of quota used.

Method	Description
isEncPasswordCorrect(String)	Indicates whether the given password is the same as the encrypted one in the database.
isPasswordCorrect(String)	Indicates whether the given password matches with the password in the database.
isQuotaCheckpointExceeded()	Indicates whether the quota used exceeds the checkpoint.
isQuotaUsed()	Indicates whether the account exceeds the quota.
refresh()	Refresh the user settings.
refresh(String)	Refreshes either the IO settings (type='I') or the UM settings (type='E').
resetPassword(String, String, String)	Resets a user's password.
set(String, Object)	Sets the value of a property by name.
setAccount(String, String)	Sets and updates the user's email alias and password.
setAvailableGSMSearch(String)	Sets and updates the availability of GSM search.
setFaxForward(String, String, String, String, String)	Sets and updates the user's fax forwarding setting.
setMailForward(String, String)	Sets and updates the user's mail forwarding setting.
setNotification(String)	Sets and updates the status of notification.
setPersonalCodes(String, String, String, String)	Sets and updates the personal codes (puk, mobile, pin and fax codes).
setQuotaCheckpoint(String)	Sets the check point for the quota available: default is 80%.
update()	Updates the settings in the um_personal_profile table and in the InterOffice system.
updateAccountSettings(String, String, String, String, String)	Updates a set of account references.

Trace Class

The Trace class acts as a Unified Messaging template trace functionality. The class should be instantiated when a new SDK mail session is constructed. A trace record

will be inserted into the UM_SDK_TRACE table, including the customer ID, a keyword, and a description.

The following table lists the methods of the Trace class:

Method	Description
close()	Closes the trace.
finalize()	Calls close and finalizes the object.
write(String)	Writes a trace record to the UM_SDK_TRACE table.
write(String, String)	Writes a trace record to the UM_SDK_TRACE table.

The MS Package

The `oracle.um.ms` package contains classes for interacting with an IMAP4-compliant messaging system.

BodyPart Class

The BodyPart class represents a body part in a multipart message. This class encapsulates Javamail's BodyPart and MimeBodyPart classes and exposes only those functions related to Unified Messaging. A BodyPart object is an item contained within a message. This item can be an attached file, an attached message, or the actual text of the e-mail message. A BodyPart object can contain another whole message which, in itself, can be another multipart message containing many body parts. This sequence can go several levels deep: one BodyPart object may contain another message and that message, in turn, may contain more messages.

The following table lists the methods of the BodyPart class:

Method	Description
getContent()	Gets the current object's content.
getContentId()	Gets current object's MIME content-id.
getContentMIME()	Gets current object's MIME content-type.
getContentType()	Gets current object's MIME content-type.
getDescription()	Gets the current object's description.
getDisposition()	Gets current object's disposition.

Method	Description
getEncoding()	Gets the transfer encoding scheme of the current object's MIME message.
getFileName()	Gets the current object's filename.
getSize()	Gets the current object's size in bytes.
isAttachment()	Checks whether the current object's is an attachment.
isInLine()	Checks whether the current object's content is to be displayed in-line.
isMimeType(String)	Checks whether the current object's is a MIME type.
setContent(Object, String)	Set the current object's content
setDataHandler(DataHandler)	Sets the data handler for the content.
setDescription(String)	Set the current object's description.
setDisposition(String)	Sets the current object's disposition.
setFileName(String)	Sets the current object's filename.
setText(String)	Sets the content to text / plain.

Folder Class

The Folder class represents a folder in a message store. This class encapsulates JavaMail's Folder class and exposes only those functions related to Unified Messaging. A folder can contain other folders or messages. Subfolders, if they exist, can also be retrieved. Messages are sorted in the order of arrival into the folder.

The following tables list the attributes and methods of the Folder Class:

Attribute	Description
ASCENDING	Sets the sorting order to ascending.
DESCENDING	Sets the sorting order to descending.
FROM	Specifies that messages should be sorted using FROM.
RECEIVED_DATE	Specifies that messages should be sorted using Received_Date.
SUBJECT	Specifies that messges should be sorted using SUBJECT.

Method	Description
appendMessages(Message[])	
copyMessages(int[], Folder)	Copies specified messages in this to the specified folder.
copyMessages(Message[], Folder)	Copies JavaMail messages from this to the specified folder.
createFolder(String)	Creates a subfolder in this.
delete()	Deletes this from the message store.
deleteMessages(int[])	Deletes specified messages from this.
getFolder(int)	Gets the subfolder in this specified by fldrindx.
getFolder(String)	Gets the subfolder specified by foldername.
getFolderCount()	Gets the number of subfolders in this.
getFolders()	Gets all the subfolders in this.
getFullName()	Gets the full name of this.
getMessage(int)	Retrieves the message of a given index.
getMessageByUID(long)	Gets a message by UID.
getMessageCount()	Gets the number of messages in this.
getMessages()	Gets all messages in this.
getMessages(int, int)	Gets all messages between startnum and endnum.
getMessages(int[])	Gets the messages specified by an array of message indices.
getMessages(Message[])	Gets an array of messages which encapsulate the given JavaMail messages.
getName()	Get this name
getNewMessageCount()	Gets the number of messages that have arrived since this was last opened.
getNextMessage(int)	Gets the next message given the current message index.
getParent()	Gets this parent folder Returns null if this is root.
getStore()	Gets the message store that this is part of.
getUIDValidity()	Gets the UID Validity value for this.
getUnreadMessageCount()	Gets the number of unread messages in this.

Method	Description
hasNewMessages()	Indicates whether new messages have arrived since this was last opened.
holdsFolders()	Indicates whether this can contain subfolders.
holdsMessages()	Indicates whether this can contain messages.
isRoot()	Indicates whether this is the root folder.
moveFolder(Folder)	Moves this folder to a different parent folder.
moveMessages(int[], Folder)	Moves messages from this to another folder.
moveMessages(int[], String)	Moves an list of messages from this to the specified folder.
preFetch()	Prefetches information about the first few messages in this.
refresh()	Refreshes the subfolder list.
renameTo(String)	Renames this.
search(SearchTerm)	Searches this and returns messages based on the search criteria.
setSort(String, String)	Sets the sorting order for messages in this.
toString()	Returns the full name of this.

InternetAddress Class

This class extends JavaMail's InternetAddress class to provide Name Resolution through the JNDI API. This class exposes only those functions related to Unified Messaging and resolves private Addresses and Persons in the Directory. This class is used until a generic Name Resolution mechanism is implemented in the Mail Transfer Agent (MTA).

The following table lists the methods of the InternetAddress class:

Method	Description
main(String[])	Class tester
ResolveEmailAddress(DirContext, String, String, MsgSrv, String)	Resolves a list of addresses delimited by comma to an array of valid email addresses.
resolveMobileNumber(String, String, MsgSrv)	Resolves a list of aliases / dl's to the corresponding phone numbers.

Method	Description
ResolveName(String, DirContext, String, String, MsgSrv, String)	Resolves string Address List delimited by comma space or semi-colon.

Message Class

The Message class represents a message in a message store. It encapsulates JavaMail's Message class and exposes only those functions related to Unified Messaging. If a message is composed of multiple parts (such as a message and an attachment), it contains one MultiPart object which holds the multiple parts of the message. Each of these parts of the message can then be retrieved by going through the BodyPart objects included in the MultiPart object. Normally a message will contain a MultiPart object.

The following tables list the attributes and methods of the Message class:

Attribute	Description
IMPORTANCE_LOW	Importance Settings.
PRIORITY_HIGH	Priority Settings.

Method	Description
delete()	Deletes this.
getBcc()	Gets a comma-delimited list of email addresses in the Bcc field.
getBody()	Gets the email body of this.
getBodyMIME()	Gets this MIME body.
getCc()	Gets a comma-delimited list of email addresses in the Cc field.
getContent()	Gets this content.
getContentId()	Gets this content-id.
getContentMIME()	Gets this MIME content-type.
getContentType()	Gets this content-type.
getDescription()	Return a description String for this part.

Method	Description
getDisposition()	Returns the disposition of this message.
getEncoding()	Gets the transfer encoding scheme of this message.
getFileName()	Gets this filename.
getFlags()	Gets the flags associates with this message.
getFolder()	Gets the parent folder that this message belongs.
getFrom()	Gets the sender's address.
getImportance()	Retrieves the Importance flag for this message.
getMessageInputStream()	Gets an input stream of this message as the content.
getMessageNumber()	Gets the number of this.
getPriority()	Retrieves the priority flag for this message.
getReceivedDate()	Gets the date the message was received.
getReplyTo()	Gets the Replyto address for this message.
getSentDate()	Gets the date this message was sent.
getSize()	Gets the size of this message in bytes.
getSubject()	Gets the Subject field of this message.
getTo()	Gets a comma-separated list of addresses in the TO field.
getUID()	Gets the UID of this message, if supported by the message store.
getXMessage()	Returns the JavaMail message (xMessage)
hasAttachments()	Indicates whether this message contains attachments.
hashCode()	Overrides the default hashCode.
isAttachment()	Indicates whether the message is an attachement.
isDraft()	Indicates whether this message is a draft.
isInLine()	Indicates whether the content should be displayed in-line.
isMimeType(String)	Indicates whether this message is of a specified MIME type.
isNew()	Indicates whether this is a new message since the folder was last opened.
isRead()	Indicates whether this message has been read.
next()	Gets the next message in order from this folder.

Method	Description
previous()	Gets the previous message from this folder.
setCc(String)	Resolve a comma-delimited list of names to email addresses.
setContent(Object)	Sets the content of this.
setDisposition(String)	Sets the disposition of this.
setDraft(boolean)	Sets the Draft flag for this object.
setFrom(String)	Sets the email address to the FROM field.
setImportance(String)	Sets the importance flag.
setPriority(String)	Sets the priority flag.
setReplyTo(String)	Sets the email address for the Reply to field.
setSentDate(Date)	Sets the SentDate for this message
setSubject(String)	Sets the Subject for this message
setTo(String)	Sets a comma-delimited list of names in the TO field.
toString()	Displays information about this.

MultiPart Class

The MultiPart class represents a multipart of a message. A Multipart object is a container that contains one or more BodyPart objects.This class encapsulates the JavaMail Multipart and MimeMultipart classes and exposes only those functions related to Unified Messaging.

The following table lists the methods of the MultiPart class:

Method	Description
addBodyPart(BodyPart)	Adds a BodyPart to this.
getBody()	Gets the content of this in the best displayable format.
getBodyContentType()	Returns either 'text/plain' or 'text/html' as the content type.
getBodyMIME()	Same as getBodyContentType but complient with IOSDK.
getBodyPart(int)	Gets this BodyPart specified by the location index.
getContentType()	Gets the content type of this In all cases, this will be multipart/alternative.

Method	Description
getCount()	Gets the number of BodyPart objects in this.

SearchFolder Class

The SearchFolder class represents a folder in which the search result messages are stored. This class is a subclass of the Folder class and exposes the functions to allow the templates to easily access the messages from a search operation.

The following table lists the methods of the SearchFolder class:

Method	Description
appendMessages(Message[])	Puts the set of matching messages into the SearchFolder object.
copyMessages(int[], Folder)	This method is not supported for this object.
delete()	This method is not supported for this object.
deleteMessages(int[])	This method is not supported for this object.
getMessage(int)	Gets the message at the specified index
getMessageByUID(long)	This method is not supported for this object.
getMessageCount()	Gets the number of matching messages from a search operation.
getMessages()	Gets all the messages of the search.
getMessages(int, int)	Gets the messages between the two specified .
getMessages(int[])	Gets the messages specified by an array of message indices.
getUnreadMessageCount()	This method is not supported for this object.
hasNewMessages()	This method is not supported for this object.
moveMessages(int[], Folder)	This method is not supported for this object.
moveMessages(int[], String)	This method is not supported for this object.

Store Class

The Store class is the main class of Unified Messaging's external message store connection. It represents a connection to an IMAP4-enabled message store where the user's e-mail, voice mail, and facsimiles are stored. A Unified Messaging user may have many of these message store objects, one for each external connection.

The Store class encapsulates the Javamail Store class, exposing only those methods and properties pertinent to Unified Messaging. A user's Unified Messaging session is created by the `oracle.um.sdk.Session` object. From that object, the user can retrieve external message stores by calling the getMsgStores method.

The following tables list the attributes and methods of the Store class:

Attribute	Description
RETRIEVE_BY_CUSTOMER	Constant used to select the type of retrieval by customerid.
RETRIEVE_BY_PHONE	Constant used to select the type of retrieval by phone_id.

Method	Description
close()	Closes the message store connection.
connect()	Connects to the message store.
create()	Creates the message store record.
createFolder(Folder, String)	Creates a folder in this.
createFolder(String)	Creates a folder in this.
emptyWastebasket()	Empties the wastebasket.
getAddress()	Gets the address of this message store.
getAggregate()	Gets the String value of this AGGREGATE attribute.
getCreateDate()	Gets the date this was created.
getCustomerId()	Gets this customer id.
getFolder(String)	Gets a specified folder in the root folder of this.
getInbox()	Gets the inbox folder of this.
getMigration()	
getMsgStoreName()	Gets the name of this message store.
getMsgStoreType()	Gets the type of this message store.
getPort()	Gets the port this uses.
getProtocol()	Gets this protocol.
getRetrievedDate()	Gets the date the folders for this were last opened.

Method	Description
getRoot()	Gets the root folder of this.
getUpdateDate()	Gets the date this was last updated.
getUserId()	Gets the userid.
getWastebasket()	Gets the wastebasket folder from this.
isAggregate()	Indicates whether this is to be aggregated to the UM
isAggregate	Indicates whether this is connected to the message
retrieve(int)	Retrieves this record by either the customer or the phone
search(SearchTerm, String, boolean)	Searches for messages in the UM message store folders.
setAddress(String)	Sets the address of this message store.
setAggregate(boolean)	Sets this AGGREGATE value.
setCreateDate(Date)	Sets the date this was created.
setCustomerId(String)	Sets this customer id.
setMsgStoreName(String)	Sets the name of this message store.
setMsgStoreType(String)	Sets the type of this message store.
setPassword(String)	Sets the password.
setPhoneId(String)	Sets this phone id.
setPort(int)	Sets the port this uses.
setProtocol(String)	Sets this protocol.
setUserId(String)	Sets the userid.
toString()	Returns the full address of this.
update	Updates the information of the current object in the database.

Transport Class

The Transport class represents a Transport in Unified Messaging. It encapsulates JavaMail's Transport class and exposes only those functions related to Unified Messaging.

The following table lists the methods of the Transport class:

Method	Description
send(Session, Message)	Sends the message using the default transport.

UMInbox Class

The UMInbox class consolidates inboxes marked "aggregate" to provide a single view of messages from multiple sources. This class represents an inbox folder in UMStore, and it aggregates the inboxes of those message stores being managed by UMStore. UMInbox encapsulates Javamail's Folder class and exposes only those functions related to Unified Messaging.

The following table lists the methods of the UMInbox class:

Method	Description
copyMessages(int[], Folder)	Copies specifed messages from this to the specified folder.
createFolder(String)	Overrides Folder.createFolder.
deleteMessages(int[])	Deletes the specified messages from this.
getMessage(int)	Gets the message of the given message number.
getMessageCount()	Gets the number of messages in all folders being aggregated.
getMessages()	Gets all the messages in this.
getMessages(int, int)	Gets all messages between startnum and endnum, inclusive.
getMessages(int[])	Gets all messages specified by the array of message numbers.
getNewMessageCount()	Gets the number of new messages since this was last opened.
getUnreadMessageCount()	Gets the number of unread messages in this.
hasNewMessages()	Indicates whether new messages have arrived.
holdsFolders()	Indicates whether this can hold subfolders.
holdsMessages()	Indicates whether this can contain messages.
moveMessages(int[], Folder)	Moves messages from one folder to another.
preFetch()	Prefetches information about the first few messages in this.

Method	Description
search(SearchTerm)	Searches the different inboxes for messages matching the search criteria Returns an empty array if no matches are found.
setSort(String, String)	sets the sorting order for messages in this.

UMRoot Class

The UMRoot class represents the root folder under UMStore. This class encapsulates Javamail's Folder class, exposing only those functions related to the root folder functionality of the aggregated Unified Messaging view.

The following table lists the methods of the UMRoot class:

Method	Description
getFolder(int)	Gets the subfolder specified by the folder index.
getFolder(String)	Gets the subfolder.
getFolders()	Gets all subfolders in this.
holdsFolders()	Indicates whether this can hold subfolders.
holdsMessages()	Indicates whether this can hold messages.
search(SearchTerm)	Searches the UM root folder for messages matching the search criteria.

UMStore Class

The UMStore class is the main class of the aggregated view of Unified Messaging's external message store connection. "Aggregated view" means that Unified Messaging consolidates the inboxes of several message stores (e-mail, voice mail, and facsimiles) and presents them as one. The UMStore class encapsulates the Javamail Store class, exposing only those methods and properties pertinent to Unified Messaging. A Unified Messaging user normally has one UMStore object and may have several Store objects representing non-aggregated message stores. Non-aggregated message stores are provided to Unified Messaging users to consolidate their own personal message stores into Unified Messaging. The UMStore class allows accessibility to all other mail-related objects in this package. For example, the user can obtain the list of all folders in this message store.

The following table lists the methods of the UMStore class:

Method	Description
close()	Closes the message store connection.
connect()	Connects to the message store.
getCount()	Gets the number of Store objects.
getElement(int)	Gets the Store object which this UMStore is aggregating.
getFolder(String)	Gets the folder with the specified folder name.
getInbox()	Gets the aggregate inbox folder from this.
getRoot()	Gets the root folder.
search(SearchTerm, String, boolean)	Searches for messages in the UM message store folders.

4

Administering Unified Messaging

This chapter discusses how to administer your Unified Messaging system. The following topics are included:

- The Administrator's Inbox
- Creating New Accounts for Users
- Updating Existing Accounts
- Deleting Accounts
- Working with SMS
- Working with LDAP

The Administrator's Inbox

An administrator account, named Helpdesk, provides the administrator with a separate inbox for receiving, responding to, and storing support requests. This inbox is created as part of the Unified Messaging installation process.

Creating New Accounts for Users

Before a user can log in, the administrator must create and activate the user's account using an instance of the Session class. Use the createUser() method to create a new customer account. Here is a snippet of code from the admin_con_newUser.jsp template. This form gathers various attributes and sends them to the admin_createUser.jsp template:

```
…
<form name=newUserForm action="<%=
response.encodeUrl("admin_createUser.jsp") %>"
```

```
    target=Result method=post>
<table border=0 cellspacing=0 cellpadding=0 width=100%>
…
<td class=fieldData nowrap><input type=text
name=CUSTOMER_NAME value=""
    size=13><span class=reqField>*</span></td>
…
<td class=fieldData nowrap><input type=text name=PHONE value=""
size=13><span
    class=reqField>*</span></td>
…
<td class=fieldData nowrap><input type=text name=FAX value=""
size=13><span
    class=reqField>*</span></td>
…
<td class=fieldData nowrap><input type=text name=USER_NAME
value="" size=13><span
    class=reqField>*</span></td>
…
<td class=fieldData nowrap><input type=text name=PIN value=""
size=13><span
    class=reqField>*</span></td>
…
<td class=fieldHeader nowrap>Protocol</td>
<td class=fieldData nowrap><input type=text name=EML_PROTOCOL
value="" size=4><span
    class=reqField>*</span></td>
<td class=fieldData nowrap><input type=text name=VML_PROTOCOL
value="" size=4></td>
<td class=fieldData nowrap><input type=text name=FAX_PROTOCOL
value="" size=4></td>
…
<td class=fieldHeader nowrap>Address</td>
<td class=fieldData nowrap><input type=text name=EML_ADDRESS
value="" size=13><span
    class=reqField>*</span></td>
<td class=fieldData nowrap><input type=text name=VML_ADDRESS
value="" size=13></td>
<td class=fieldData nowrap><input type=text name=FAX_ADDRESS
value="" size=13></td>
…
<td class=fieldData nowrap><input type=text name=EML_USERID
value="" size=13><span
    class=reqField>*</span></td>
<td class=fieldData nowrap><input type=text name=VML_USERID
```

```
value="" size=13></td>
<td class=fieldData nowrap><input type=text name=FAX_USERID
value="" size=13></td>
…
<td class=fieldHeader nowrap>Password</td>
<td class=fieldData nowrap><input type=text name=EML_PASSWORD
value=""
    size=13><span class=reqField>*</span></td>
<td class=fieldData nowrap><input type=text name=VML_PASSWORD
value=""
     size=13></td>
<td class=fieldData nowrap><input type=text name=FAX_PASSWORD
value=""
     size=13></td>
…
</table>
</form>
…
```

This is the simple call to createUser() in admin_createUser.jsp:

```
umSession.createUser(att, val);
```

Updating Existing Accounts

When the administrator needs to update an account, the first step is to search through the list of customers to access the correct customer record. For this task, use the `getAdministratorList` method in the Session class.

The results of this search may be multiple Unified Messaging accounts, so to find the correct account, you must iterate through the AdministratorList object. The AdministratorList object is a list of Settings objects. Most of the attributes in the Settings class may be changed by either the user of the account or the administrator. For a list of Settings attributes and HTML samples, refer to the Javadoc for `oracle.um.sdk.Settings`.

This example includes two parts. The first example shows how to search for a Unified Messaging customer.

```
<%
String searchID = request.getParameter ("searchID")
String searchOp = request.getParameter ("searchOp")
String searchVal = request.getParameter ("searchVal")
Session umSession = (Session) session.getValue (""umSession")
```

```
AdministratorList admList = umSession.getAdministratorList (searchID, searchVal,
searchOp);
%>
```

The second HTML page shows how to update that customer's password:

```
<%
Settings userSettings = admList.getElement (0);
String[]att = newString [1];
String[]val = newString [1];
att[0] = "password";
val[o] = "new_password";
userSettings.set (att, val);
userSettings.update ();
%>
```

Note that when constructing the search, you may also search on EMAIL_ID, PHONE_ID, or any other column in the UM_ PERSONAL_PROFILE table. For the Operation value, you may use START, EQUAL, or CONTAINS.

Deleting Accounts

To delete a Unified Messaging customer, call the `deleteUser` method.

```
<%
Session umSession = (Session)
    session.getValue ("umSession");
String phone_id = request.getParameter ("phone_id");
String customer_id = request.getParameter ("phone_id");
umSession.deleteUser (phone_id, customer_id);
%>
```

The deletion is successful if a record exists that matches the specified PHONE_ID and CUSTOMER_ID. Otherwise, an exception is thrown.

Working with SMS

In addition to receiving e-mail, voice mail, and facsimiles, customers who use pagers will want to be notified of the arrival of important messages. The Short Messaging Service (SMS) provides a connection to the SMS message store that allows customers to send and receive these short pager messages, which are limited to about 150 characters in length.

There are four ways an SMS message could be initiated from UM:

Scenario 1: During account activation, UM sends a registration code to the subscriber's pager.

Scenario 2: UM users with the notification feature activated are notified of the arrival of new messages.

Scenario 3: UM users can send SMS messages from the UM Internet client running on a Web browser.

Scenario 4: UM users can request that the next portion of e-mail text be displayed on the pager. (UM users can also originate an e-mail message from a pager, when the pager system includes this function.)

To send an SMS message, first create an SMS connection:

```
<%
SMS smsconn = umSession.getSMSConnection ();
%>
```

Then, make the following call to `smsconn.send`:

```
<%
String receiver = request.getParameter ("receiver");
String message = request.getParameter ("message");
smsconn = send (receiver, message);
%>
```

There is an outbound SMS process that handles SMS messages originating from UM. In Scenario 1 and Scenario 3, the SMS process passes the request from its input queue to the SMS gateway process for delivery. In Scenario 2, the SMS process checks to see if the notification rule is active and if the arrived message passes the user's notification rule. If the message passes both checks, SMS passes the contents of the e-mail (or the initial portion of e-mail, if it exceeds the SMSC buffer limitation) to the SMS gateway for delivery. In Scenario 4, SMS receives the request from an inbound SMS process and passes the next portion of the e-mail content to the SMS gateway process.

For inbound SMS delivery, the SMS inbound process checks to see whether the request is a new e-mail message or a request for the next portion of a current e-mail message. If the request is for a new e-mail message, SMS passes the request to the SMTP processor; otherwise, SMS passes the request to the outbound SMS process.

Customers may want to filter their messages based on time, sender, or other categories. The UM notification rules provide this functionality. UM provides API classes you can use to create, modify, and delete notification rules based on filtering options. A sample of creating, modifying, and deleting notification rules can be

found in the $ORACLE_HOME/um/templates/um directory, called pref_con_rules.jsp.

Sample code for the scenario where a UM user sends an SMS message from the UM Internet client can be found in $ORACLE_HOME/um/templates/um directory called msg_comp_sms_send.jsp.

Working with LDAP

Your company may already have an LDAP server containing a corporate directory listing both UM users and non-UM users; that is, users who are not part of your UM system. A UM user can search this server to obtain the e-mail address or phone number of both UM users and non-UM users.

You can expose this directory from within Unified Messaging by connecting the LDAP server to your UM application.

Use the GSMDir and GSMAddress classes in the `oracle.um.sdk` package to work with UM's own directory, as well as external LDAP servers.

An external LDAP server is likely to contain information about users who are not known to the UM system. When displaying information about these users, you will not be able to use the normal methods in GSMAddress class to display the various user properties. To display properties defined by the external LDAP server, you must first know the name of the properties as defined within that LDAP server. Then you can use the generic method to retrieve the values for these properties.

The following example shows how this is done in an HTML template file.

```
GSMAddress person = dir.getElement (1);
```

Where dir is of the GSM Dir and is the result of the umsession.getDirectory method.

Set `person` to the second element returned from searching through the external LDAP server:

```
person.get("FIRST_NAME");
person.get("ST");
person.get("STREET");
```

Note that you may need to test your code using all capital letters for the property name.

Using this technique, you can customize your UM application to display data stored on any LDAP server. The only requirement is that you need to know the names of the properties defined for the directory items on the LDAP server.

The default installation of UM will set up the UM system to use the same LDAP server for both the UM directory and a generic directory. See the Unified Messaging installation documentation for information about how to change this server name to point to your own LDAP server.

Index

F

H

I

L

M

N

O

P

R

S

T

U

W