Oracle® Fusion Middleware

Developing Applications with Oracle User Messaging Service Using Oracle WebLogic Server Proxy Plug-Ins

14*c* (14.1.2.0.0)
F89315-02
February 2025

Oracle Fusion Middleware Developing Applications with Oracle User Messaging Service Using Oracle WebLogic Server Proxy Plug-Ins, 14*c* (14.1.2.0.0)

F89315-02

Primary Author: Oracle Corporation

Contents

A Using the User Messaging Service Sample Applications

Preface

This document describes how to use Oracle User Messaging Service.

Audience

This guide is intended for process developers who use Oracle User Messaging Service to send and receive messages from their applications.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

For more information, see the following documents:

- *Release Notes*
- *Administering Oracle User Messaging Service*

- *Developing SOA Applications with Oracle SOA Suite*
- *WLST Command Reference for Infrastructure Components*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Overview

This chapter provides an overview of Oracle User Messaging Service (UMS) APIs and it includes the following sections:

- Introduction to User Messaging Service
  Oracle User Messaging Service (UMS) provides a common service responsible for sending out messages from applications to devices or services using various protocols. It also routes incoming messages from devices or services to applications.
- Overview of User Messaging Service APIs
  Oracle Fusion Middleware application developers can either use the UMS Java API or the UMS Web Service API to implement messaging services in their applications.
- Deprecated APIs
  The Parlay X Multimedia Messaging API and the User Messaging Service EJB API are deprecated in this release.
- User Messaging Service Sample Applications
  The code samples for Oracle User Messaging Service are available on Oracle Technology Network.

Introduction to User Messaging Service

Oracle User Messaging Service (UMS) provides a common service responsible for sending out messages from applications to devices or services using various protocols. It also routes incoming messages from devices or services to applications.

To learn more about the architecture and features of Oracle User Messaging Service, see the "Introduction to Oracle User Messaging Service" chapter in *Administering Oracle User Messaging Service*.

Overview of User Messaging Service APIs

Oracle Fusion Middleware application developers can either use the UMS Java API or the UMS Web Service API to implement messaging services in their applications.

Use the UMS Java API if the application runs from the same JEE server as the UMS Messaging Engine. Otherwise, use the UMS Web Services API. In the case of Web Services API, the provided UMS client code wraps the actual Web Services calls and make use of the UMS Web Services API very similar to the UMS Java API.

For more information about the classes and interfaces, see *User Messaging Service Java API Reference*.

How does the UMS Java API work?

The UMS Client library consists of two main components, `MessagingClient` and `ListenerManager`. The `MessagingClient` is created through the `MessagingClientFactory` and it is the handle that the client uses to send and receive messages, to register `AccessPoints` (represents the addresses an application uses for receiving messages), and to set listeners. `Addresses` and `AccessPoints` are created through the `MessagingFactory`. The

`ListenerManager` handles the callbacks when the application wants to receive messages or status information asynchronously using listeners.

When a `MessagingClient` is created, it registers the application in the user messaging engine and when a message or status arrives, the `ListenerManager` calls the listener's `onMessage` or `onStatus` method.

The following code snippet illustrates how to create a Messaging client, how to register an access point, and how to send a message:

```
//Create MessagingClient
HashMap<String, Object> clientParameters = new HashMap<String, Object>();
clientParameters.put(ApplicationInfo.APPLICATION_NAME, appName);
MessagingClient messagingClient =
MessagingClientFactory.createMessagingClient(clientParameters);

//Create sender and recipient addresses
Address sender = MessagingFactory.createAddress("EMAIL:alice@example.com");
Address recipient = MessagingFactory.createAddress("EMAIL:bob@example.com");

//Create and register an accessPoint for the Address(es) you want to be able to receive
messages for.
AccessPoint accessPoint = MessagingFactory.createAccessPoint(sender);
messagingClient.registerAccessPoint(accessPoint);

//Create and send a message
Message message = MessagingFactory.createTextMessage("Hello World");
message.setSubject("Test Subject");
message.addRecipient(recipient);
message.addSender(sender);
String mid = messagingClient.send(message);
```

Deprecated APIs

The Parlay X Multimedia Messaging API and the User Messaging Service EJB API are deprecated in this release.

The UMS Java API replaces the deprecated EJB API and the UMS Web Services API replaces the deprecated Parlay X Multimedia Messaging API. See Sending and Receiving Messages using the User Messaging Service Java API and Sending and Receiving Messages using the User Messaging Service Web Service API.

For more information about the deprecated APIs, see *Developing Applications with Oracle User Messaging Service Using Oracle WebLogic Server Proxy Plug-Ins*.

User Messaging Service Sample Applications

The code samples for Oracle User Messaging Service are available on Oracle Technology Network.

Oracle Technology Network
http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html.

> **Note:**
>
> Unless explicitly identified as such, the sample codes are not certified or supported by Oracle; it is intended for educational or testing purposes only.

You can build and deploy these sample applications using Oracle JDeveloper 14c, and also manage communication preferences through a web interface (see *Administering User Communication Preferences*).

2

Sending and Receiving Messages using the User Messaging Service Java API

This chapter describes how to use the User Messaging Service (UMS) client API to develop applications. This API serves as a programmatic entry point for Fusion Middleware application developers to incorporate messaging features within their enterprise applications.
For more information about the classes and interfaces, see *User Messaging Service Java API Reference*.

> **Note:**
>
> To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, refer to the samples at:
>
> `http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html`

- Introduction to the UMS Java API
  The UMS API provides a plain old java (POJO/POJI) programming model and this eliminates the needs for application developers to package and implement various Jakarta EE modules (such as an EJB module) in an application to access UMS features.
- Creating a UMS Client Instance and Specifying Runtime Parameters
  The `MessagingClient` object is essential to the UMS Java API user. This object is used to, for instance, send a message, retrieve status information for a sent message, and to receive messages synchronously. The `MessagingClient` object is created using the `MessagingClientFactory.createMessagingClient()` method.
- Sending a Message
  Use the `MessagingFactory` class to create a UMS Message object for the client application. The `MessagingFactory` class is also used to create `Addresses`, `AccessPoints`, `MessageFilters`, and `MessageQueries`.
- Retrieving Message Status
  After sending a message, use Oracle UMS to retrieve the message status either synchronously or asynchronously.
- Receiving a Message
  An application that wants to receive incoming messages must register one or more access points that represent the recipient addresses of the messages.
- Configuring for a Cluster Environment
  The API supports an environment where client applications and the UMS server are deployed in a cluster environment.
- Using UMS Client API for XA Transactions
  UMS provides support for XA enabled transactions for outbound and inbound messages. The industry standard, X/Open XA protocol, is widely supported in other Oracle products such as Business Process Management (BPM).

- Using UMS Java API to Specify Message Resends
  When a message send attempt is classified as a complete failure, that is, the failover chain is exhausted, the message is automatically scheduled for resend by the UMS Server. This is repeated until the message is successfully sent or the configured number of resends is reached.
- Selecting a Driver Programmatically
  Unless multiple drivers of the same type is used, the UMS Server engine selects a driver based on the delivery type that is defined in the UMS Message sent by the client application.
- Setting up Priority and Expiration time for a Message
  In certain circumstances such as a high load, you may benefit from specifying UMS Message priority when sending a message.
- Specifying User Preference Application Partitioning Profile ID
  User Communication Preferences are invoked for recipient addresses that contains the USER prefix (for example, USER:john.doe). The preferences can be partitioned into profiles.
- Configuring Security
  Client applications may need to specify one or more additional configuration parameters to establish a secure listener.
- Threading Model
  Client applications that use the UMS Java API are usually multi-threaded. Typical scenarios include a pool of EJB instances, each of which uses a `MessagingClient` instance; and a servlet instance that is serviced by multiple threads in a web container.

Introduction to the UMS Java API

The UMS API provides a plain old java (POJO/POJI) programming model and this eliminates the needs for application developers to package and implement various Jakarta EE modules (such as an EJB module) in an application to access UMS features.

This reduces the complexity of the client application, and also, reduces application development time because developers can create applications to run in a Jakarta EE container without performing any additional packaging of modules, or obtaining specialized tools to perform such packaging tasks.

Consumers do not need to deploy any EJB or other Jakarta EE modules in their applications, but must ensure that the UMS libraries are available in an application' s runtime class path. The deployment is as a shared library, `oracle.sdp.client`.

The samples with source code are available on Oracle Technology Network (OTN).

Creating a UMS Client Instance and Specifying Runtime Parameters

The `MessagingClient` object is essential to the UMS Java API user. This object is used to, for instance, send a message, retrieve status information for a sent message, and to receive messages synchronously. The `MessagingClient` object is created using the `MessagingClientFactory.createMessagingClient()` method.

Client applications can specify a set of parameters at runtime when instantiating a MessagingClient object. For example, you configure a `MessagingClient` instance by specifying parameters as key-value pairs in a `java.util.Map<String, Object>`. Among other things, the

configuration parameters serve to identify the client application, point to the UMS server, and establish security credentials. Client applications are responsible for storing and loading the configuration parameters using any available mechanism.

The `MessagingClient` needs to be released when the application stops or undeployes, or when the EJB bean that uses the client destroyes.

Table 2-1 lists some configuration parameters that may be set for the Java API. In typical use cases, most of the parameters do not need to be provided and the API implementation uses sensible default values.

Table 2-1 Configuration Parameters Specified at Runtime

Parameter	Notes
`APPLICATION_NAME`	Optional. By default, the client is identified by its deployment name. This identifier can be overridden by specifying a value for key `ApplicationInfo.APPLICATION_NAME`.
`APPLICATION_INSTANCE_NAME`	Optional. Only required for certain clustered use cases or to take advantage of session-based routing.
`SDPM_SECURITY_PRINCIPAL`	Optional. By default, the client's resources are available to any application with the same application name and any security principal. This behavior can be overridden by specifying a value for key `ApplicationInfo.SDPM_SECURITY_PRINCIPAL`. If a security principal is specified, then all subsequent requests involving the application's resources (messages, access points, and so on.) must be made using the same security principal.
`MESSAGE_LISTENER_THREADS` `STATUS_LISTENER_THREADS`	Optional. When listeners are used to receive messages or statuses asynchronously, the number of listener worker threads can be controlled by specifying values for the MessagingConstants.`MESSAGE_LISTENER_THREADS` and `MessagingConstants.STATUS_LISTENER_THREADS` keys.
`LISTENER_TRANSACTION_MODE`	Optional. When receiving messages, using a listener in the transaction can be controlled by specifying a value for this parameter.

To release resources used by the `MessagingClient` instance when it is no longer needed, call `MessagingClientFactory.remove(client)`. If you do not call this method, some resources such as worker threads and JMS listeners may remain active.

Example 2-1 shows a sample code for creating a `MessagingClient` instance:

Example 2-1 Creating a MessagingClient Instance

```
Map<String, Object> params = new HashMap<String, Object>();
// params.put(key, value);  // if optional parameters need to be specified.
MessagingClient messagingClient = MessagingClientFactory.createMessagingClient(params);
```

A `MessagingClient` cannot be reconfigured after it is instantiated. Instead, you must create a new instance of the `MessagingClient` class using the desired configuration.

The API reference for class `MessagingClientFactory` can be accessed from the Javadoc.

Sending a Message

Use the `MessagingFactory` class to create a UMS Message object for the client application. The `MessagingFactory` class is also used to create `Addresses`, `AccessPoints`, `MessageFilters`, and `MessageQueries`.

See *User Messaging Service Java API Reference* for more information about these methods.

When the client application sends a message, the UMS API returns a String identifier that the client application can later use to retrieve message delivery status. The status returned is the latest known status based on UMS internal processing and delivery notifications received from external gateways.

The types of messages that can be created include plaintext messages, multipart messages that can consist of text/plain and text/html parts, and attachments. However, note that the protocol implemented by a driver may limit the kind of message that can be sent through a driver. To address this problem, it is possible to create payloads specific to a delivery channel (`DeliveryType`) in a single message as described in Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types.

The *Device Address Type* used for different drivers are explained in the following table:

Table 2-2 Device Address Types for Drivers

Drivers	Address Type
SMPP	Device Address Type - SMS Example - SMS:1234
XMPP	Device Address Type - IM Example - IM:n.n@example.com
Extension	Device Address Type - URI:<protocol> Example - can be any thing, e.g URI:myExt:n_n
Email	Device Address Type - EMAIL Example - EMAIL:n.n@example.com
GCM	Device Address Type - URI:gcm Example - URI:gcm:sdks98sdfj098dsslkjasqijer93
Twitter	Device Address Type - URI:twitter Example -URI:twitter:n_n
APNS	Device Address Type - URI:apns Example -URI:apns:sdks98sdfj098dsslkjasqijer93

- Creating a Message
- Creating an Address
- User Preference Based Messaging
- Sending Group Messages

Creating a Message

This section describes the various types of messages that can be created. The message properties are explained in the table below Table 2-3:

- Creating a Plaintext Message
- Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts)
- Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types
- Creating a message with Unicode characters like Emojis
- Creating a Message with an Attachment (works only for Email)

Creating a Plaintext Message

Example 2-2 shows how to create a plaintext message using the UMS Java API.

Example 2-2 Creating a Plaintext Message Using the UMS Java API

```
Message message = MessagingFactory.createTextMessage("This is a Plain Text message.");
```

or

```
Message message = MessagingFactory.createMessage();
message.setContent("This is a Plain Text message.", "text/plain;charset=utf-8");
```

Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts)

Example 2-3 shows how to create a multipart or alternative message using the UMS Java API.

Example 2-3 Creating a Multipart or Alternative Message Using the UMS Java API

```
Message message = MessagingFactory.createMessage();
message.setSubject("Testing attachments");
MimeMultipart mp = new MimeMultipart("related");
MimeBodyPart part1 = new MimeBodyPart();
part1.setText("A sample pdf.");
MimeBodyPart pdfPart = new MimeBodyPart();
pdfPart.attachFile("/tmp/sample-content.pdf");
pdfPart.setDisposition(Part.ATTACHMENT);
mp.addBodyPart(part1);
mp.addBodyPart(pdfPart);
message.setContent(mp, "related");
```

Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types

When sending a message to multiple recipients, or to a USER-address that is resolved by the User Preferences in run-time, there could be multiple channels involved. Oracle UMS application developers are required to specify the correct multipart format for each channel.

Example 2-4 shows how to create delivery channel (`DeliveryType`) specific payloads in a single message for recipients with different delivery types.

Each top-level part of a multiple payload multipart/alternative message should contain one or more values of this header. The value of this header should be the name of a valid delivery type. Refer to the available values for *DeliveryType* in the enum `DeliveryType`.

Example 2-4 Creating Delivery Channel-specific Payloads in a Single Message for Recipients with Different Delivery Types

```
Message message = MessagingFactory.createMessage();

// create a top-level multipart/alternative MimeMultipart object.
MimeMultipart mp = new MimeMultipart("alternative");

// create first part for SMS payload content.
MimeBodyPart part1 = new MimeBodyPart();
part1.setContent("Text content for SMS.", "text/plain");

part1.setHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "SMS");

// add first part
mp.addBodyPart(part1);

// create second part for EMAIL and IM payload content.
MimeBodyPart part2 = new MimeBodyPart();
MimeMultipart part2_mp = new MimeMultipart("alternative");
MimeBodyPart part2_mp_partPlain = new MimeBodyPart();
part2_mp_partPlain.setContent("Text content for EMAIL/IM.", "text/plain");
part2_mp.addBodyPart(part2_mp_partPlain);
MimeBodyPart part2_mp_partRich = new MimeBodyPart();
part2_mp_partRich.setContent("<html><head></head><body><b><i>" + "HTML content for EMAIL/
IM." +
"</i></b></body></html>", "text/html");
part2_mp.addBodyPart(part2_mp_partRich);
part2.setContent(part2_mp, "multipart/alternative");

part2.addHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "EMAIL");
part2.addHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "IM");

// add second part
mp.addBodyPart(part2);

// set the content of the message
message.setContent(mp, "multipart/alternative");

// set the MultiplePayload flag to true
message.setMultiplePayload(true);
```

The API reference for class `MessagingFactory` , interface `Message` and enum `DeliveryType` can be accessed from *User Messaging Service Java API Reference*.

Creating a message with Unicode characters like Emojis

If non-ASCII characters are used, then the charset on the UMS message must be set accordingly. For example, to UTF-8. Example 2-5 is sample code that illustrates creating messages with the character 'ä' and the emoticon.

Example 2-5 Creating a message with Unicode characters like Emojis

```
// message with 'ä'
Message message1 = MessagingFactory.createTextMessage("\u00e4", "UTF-8");
// message with SMILING FACE WITH SUNGLASSES (U+1F60E)
Message message2 = MessagingFactory.createTextMessage("\ud83d\ude0e", "UTF-8");
```

Creating a Message with an Attachment (works only for Email)

Example 2-6 shows how to create a message with an attachment.

Example 2-6 Creating a Message Attachment

```
Message message = MessagingFactory.createMessage();
message.setSubject("Testing attachments");

MimeMultipart mp = new MimeMultipart("mixed");

MimeBodyPart part1 = new MimeBodyPart();
part1.setText("A sample pdf.");

MimeBodyPart part2 = new MimeBodyPart();
part2.attachFile("/tmp/sample-content.pdf");

mp.addBodyPart(part1);
mp.addBodyPart(part2);
message.setContent(mp, MimeType.MULTIPART_MIXED.toString());
```

Creating an Address

This section describes the various types of UMS addresses available and how to create address objects.

- Types of Addresses
- Creating Address Objects
- Creating a Recipient with a Failover Address
- Recipient Types
- API Reference for Class MessagingFactory
- API Reference for Interface Address

Types of Addresses

This section describes the various type of addresses available in UMS:

- Device Address: A device address can be of various types, such as email addresses, instant messaging addresses, uri:twitter, uri:apns, uri:gcm, uri:popup and telephone numbers. See "Creating Address Objects".
- Failover address: A backup or failover address that will be used if the message failed to send to the original address. See "Creating a Recipient with a Failover Address".
- User Address: User addresses are user IDs in a user repository that during a message send will be resolved to device addresses. See "User Preference Based Messaging ".
- Group Address - Group Addresses are LDAP groups (or enterprise roles) that during a message send will be resolved to User and/or Device Addresses. See "Sending Group Messages".
- Application Role Address: Application Role Addresses will, during a message send, be resolved to Group, User and/or Devices Addresses. See "Sending Messages to an Application Role".

Creating Address Objects

You can address senders and recipients of messages by using the class `MessagingFactory` to create `Address` objects.

- Creating a Single Address Object
- Creating Multiple Address Objects in a Batch
- Adding Sender or Recipient Addresses to a Message
- Adding Display Name to a Sender Address When using Email Delivery Type

Creating a Single Address Object

Example 2-7 shows code for creating a single `Address` object:

Example 2-7 Creating a Single Address Object

```
Address recipient = MessagingFactory.createAddress("EMAIL:john@example.com");
```

Creating Multiple Address Objects in a Batch

Example 2-8 shows code for creating multiple `Address` objects in a batch:

Example 2-8 Creating Multiple Address Objects in a Batch

```
String[] recipientsStr = {"EMAIL:john@example.com", "SMS:123456"};
Address[] recipients = MessagingFactory.createAddress(recipientsStr);
```

Adding Sender or Recipient Addresses to a Message

Example 2-9 shows code for adding sender or recipient addresses to a message:

Example 2-9 Adding Sender or Recipient Addresses to a Message

```
Address sender = MessagingFactory.createAddress("EMAIL:john@example.com");
Address recipient = MessagingFactory.createAddress("EMAIL:jane@example.com");
message.addSender(sender);
message.addRecipient(recipient);
```

Adding Display Name to a Sender Address When using Email Delivery Type

Example 2-10 shows code for adding display name to a sender when using email delivery type:

Example 2-10 Adding Display Name to a Sender Address When using Email Delivery Type

```
Address sender = MessagingFactory.createAddress("EMAIL:\"Mr Bob\" <bob@example.com>");
message.addSender(sender);
```

Creating a Recipient with a Failover Address

Example 2-11 shows a sample code for creating a recipient with a failover address:

Example 2-11 Creating a Single Address Object with Failover

```
String recipientWithFailoverStr = "EMAIL:john@example.com, SMS:123456";
Address recipient = MessagingFactory.createAddress(recipientWithFailoverStr);
```

or

```
Address recipient = MessagingFactory.createAddress("EMAIL:john@example.com");
Address failoverAddr = MessagingFactory.createAddress("SMS:123456");
recipient.setFailoverAddress(failoverAddr);
```

Recipient Types

The UMS Java API provides support for sending and receiving messages with To/Cc/Bcc recipients for use with the email driver:

- To send a message and specify a Cc/Bcc recipient, create the `oracle.sdp.messaging.Address` object using `oracle.sdp.messaging.MessagingFactory.buildAddress` method. The arguments are the address value (for example, `user@domain.com`), delivery type (for example, `DeliveryType.EMAIL`), and email mode (for example, "Cc" or "Bcc").
- To determine the recipient type of an existing address object, for example in a received message, use the `oracle.sdp.messaging.MessagingFactory.getRecipientType` method, passing it the `Address` object. It returns a string indicating the recipient type.

API Reference for Class MessagingFactory

The API reference for class `MessagingFactory` can be accessed from *User Messaging Service Java API Reference*.

API Reference for Interface Address

The API reference for interface `Address` can be accessed from *User Messaging Service Java API Reference*.

User Preference Based Messaging

When sending a message to a user recipient (to leverage the user's messaging preferences), you can pass current values for various business terms in the message as metadata. The UMS server matches the supplied facts in the message against conditions for business terms specified in the user's messaging filters and sends the message to the device address that matches the user's preferences for this message.

For more information about user preferences, see Administering User Communication Preferences.

> **Note:**
>
> All facts must be added as metadata in the `Message.NAMESPACE_NOTIFICATION_PREFERENCES` namespace. Metadata in other namespaces are ignored (for resolving User Communication Preferences).

Example 2-12 shows how to specify a user recipient and supply facts for business terms for the user preferences in a message. For a complete list of supported business terms, refer to *Administering User Communication Preferences*.

Example 2-12 User Preference Based Messaging

```
Message message = MessagingFactory.createMessage();
// create and add a user recipient
Address userRecipient1 = MessagingFactory.createAddress("USER:sampleuser1");
message.addRecipient(userRecipient1);
// specify business term facts
message.setMetaData(Message.NAMESPACE_NOTIFICATION_PREFERENCES, "Customer
Name", "ACME");
//  where "Customer Name" is the Business Term name, and "ACME" is the
Business Term value (i.e, fact).
```

Sending Group Messages

You can send messages to a group of users by sending it to a group URI, or sending a message to LDAP groups (or enterprise roles) and application roles.

- Sending Messages to a Group
- Sending Messages to a Group Through a Specific Channel
- Sending Messages to an Application Role
- Sending Messages to an Application Role Through a Specific Channel

Sending Messages to a Group

You can send messages to an LDAP group or to enterprise roles.

To send a message to a group, use the `MessagingFactory` class to create a recipient address of type `GROUP` and send the message as shown in Example 2-13.

Example 2-13 Creating and addressing a message to a group

```
Address groupAddr = MessagingFactory.createAddress("GROUP:MyGroup");
Message message = MessagingFactory.createTextMessage("Sending message to a group");
message.addRecipient(groupAddr);
message.setSubject("Testing groups");
String id = messagingClient.send(message);
```

The group address `groupAddr` is eventually replaced by user addresses and the result will be as shown in Example 2-14.

Example 2-14 Group Address replaced by user addresses

```
Address groupMember1 = MessagingFactory.createAddress("USER:MyGroupMember1");
Address groupMember2 = MessagingFactory.createAddress("USER:MyGroupMember2");
Address groupMember3 = MessagingFactory.createAddress("USER:MyGroupMember3");
Message message = MessagingFactory.createTextMessage("Sending message to a group");
message.addRecipient(groupMember1);
message.addRecipient(groupMember2);
message.addRecipient(groupMember3);
message.setSubject("Testing groups");
String id = messagingClient.send(message);
```

It is the User Preferences for each user that determines where the message will eventually reach. For more information, see *Administering User Communication Preferences*.

Sending Messages to a Group Through a Specific Channel

You can specify the outgoing channel before sending a group message. To specify the outgoing channel for a group message, you must set the **DeliveryType** property of the group address (`groupAddr`) as shown in Example 2-15.

Example 2-15 Creating and addressing a message to a group through a channel

```
Address groupAddr = MessagingFactory.createAddress("GROUP:MyGroup");
groupAddr.setDeliveryType(DeliveryType.EMAIL);
Message message = MessagingFactory.createTextMessage("Sending message to a group");
message.addRecipient(groupAddr);
message.setSubject("Testing groups through email");
String id = messagingClient.send(message);
```

The group is resolved to users, then each user's email address is fetched. The user's email address in this case is the same as that used for User Preferences. If no email address exists for a user, that user is skipped.

Sending Messages to an Application Role

An application role is a collection of users, groups, and other application roles; it can be hierarchical. Application roles are defined by application policies and not necessarily known to a JakartaEE container. For more information about application roles, see *Securing Applications with Oracle Platform Security Services*.

> **Note:**
>
> An application role may map to other application roles, such as the following roles:
>
> - Authenticated role: Any user who successfully authenticates. This may result in a large number of recipients.
> - Anonymous role: There will no recipient for this role.

To send a message to an Application role, use must create a recipient address of type application role by using the *`MessagingFactory`* class. An application role belongs to an application ID (also known as application name or application stripe). Therefore, both these parameters must be specified in the recipient address as shown in Example 2-16.

Example 2-16 Creating and addressing a message to an application role

```
Address appRoleAddr =
MessagingFactory.createAppRoleAddress("myAppRole", "theAppId");
Message message = MessagingFactory.createTextMessage("Message to an application role");
message.addRecipient(appRoleAddr);
message.setSubject("Testing application roles");
String id = messagingClient.send(message);
```

The application role `myAppRole` is eventually replaced by user addresses.

If the application id is that of the calling application, then you need not specify the application id when creating the recipient address. UMS will automatically fetch the application id that is specified in the `application.name` parameter in the `JpsFilter`(`web.xml`) or `JpsInterceptor`(`ejb-jar.xml`). For more information about Filter and Interceptor parameters, see *Securing Applications with Oracle Platform Security Services*.

Sending Messages to an Application Role Through a Specific Channel

The user can specify a channel for the outgoing message in the same way as specifying a channel for sending a message to a group. You must set the delivery type on the application role address.

The following is an example of sending a message to an application role specifying email as the delivery channel:

Example 2-17 Creating and addressing a message to an application through a channel

```
Address appRoleAddr =
MessagingFactory.createAppRoleAddress("myAppRole", "theAppId");
appRoleAddr.setDeliveryType(DeliveryType.EMAIL);
Message message = MessagingFactory.createTextMessage("Message to an application role");
message.addRecipient(appRoleAddr);
message.setSubject("Testing application roles");
String id = messagingClient.send(message);
```

Retrieving Message Status

After sending a message, use Oracle UMS to retrieve the message status either synchronously or asynchronously.

There will be one status object per recipient that contains status information, which helps you understand if the message is pending, if the message was sent successfully, if the message was failed to send, if there are failover addresses, and if the message is automatically resent.

- Synchronous Retrieval of Message Status
- Asynchronous Receiving of Message Status

Synchronous Retrieval of Message Status

To perform a synchronous retrieval of current status, use the following flow from the `MessagingClient` API:

```
String messageId = messagingClient.send(message);
Status[] statuses = messagingClient.getStatus(messageId);
```

or,

```
Status[] statuses = messagingClient.getStatus(messageId, address[]) --- where
 address[] is an array of one or more of the recipients set in the message.
```

Asynchronous Receiving of Message Status

When asynchronously receiving status, the client application uses the `MessagingClient` object to specify a `Listener` object and an optional correlator object. When incoming status arrives, the listener' s `onStatus` callback is invoked. The originally-specified correlator object is also passed to the callback method.

- Creating a Listener Programmatically
- Default Status Listener
- Per Message Status Listener

Creating a Listener Programmatically

Listeners are purely programmatic. You create a listener by implementing the `oracle.sdp.messaging.Listener` interface. You can implement it as any concrete class - one of your existing classes, a new class, or an anonymous or inner class.

The following code example shows how to implement a status listener:

```
import oracle.sdp.messaging.Listener;

    public class StatusListener implements Listener {

      @Override
      public void onMessage(Message message, Serializable correlator) {
      }
      @Override
      public void onStatus(Status status, Serializable correlator) {
          System.out.println("Received Status: " + status + " with optional correlator:
" +

correlator);
      }
    }
```

You pass a reference to the `Listener` object to the `setStatusListener` or `send` methods, as described in "Default Status Listener" and "Per Message Status Listener". When a status arrives for your message, the UMS infrastructure invokes the Listener's `onStatus` method as appropriate.

Default Status Listener

The client application typically sets a default status listener (Example 2-18). When the client application sends a message, delivery status callbacks for the message invoke the default listener's `onStatus` method.

Example 2-18 Default Status Listener

```
messagingClient.setStatusListener(new MyStatusListener());
messagingClient.send(message);
```

Per Message Status Listener

In this approach, the client application sends a message and specifies a Listener object and an optional correlator object (Example 2-19). When delivery status callbacks are available for that message, the specified listener's `onStatus` method is invoked. The originally-specified correlator object is also passed to the callback method.

> **Note:**
>
> Oracle UMS uses a weak reference when storing the `Listener` object. This means that the client application is responsible for keeping a reference to the `Listener` object to prevent it from being garbage collected.

Example 2-19 Per Message Status Listener

```
statusListener = new MyStatusListener();
messagingClient.send(message, statusListener, null);
```

Receiving a Message

An application that wants to receive incoming messages must register one or more access points that represent the recipient addresses of the messages.

The server matches the recipient address of an incoming message against the set of registered access points, and routes the incoming message to the in-queue of the application that registered the matching access point. From the application perspective there are two modes for receiving a message from its in-queue, synchronous and asynchronous.

Registering an Access Point

`AccessPoint` represents one or more device addresses for receiving incoming messages.

Use `MessagingFactory.createAccessPoint` to create an access point and `MessagingClient.registerAccessPoint` to register it for receiving messages.

To register an email access point:

```
Address apAddress = MessagingFactory.createAddress("EMAIL:user1@example.com");
AccessPoint ap = MessagingFactory.createAccessPoint(apAddress);
MessagingClient.registerAccessPoint(ap);
```

To register an SMS access point for the number `9000`:

```
AccessPoint accessPointSingleAddress =
 MessagingFactory.createAccessPoint(AccessPoint.AccessPointType.SINGLE_ADDRESS,
 DeliveryType.SMS, "9000");
messagingClient.registerAccessPoint(accessPointSingleAddress);
```

To register SMS access points in the number range `9000` to `9999`:

```
AccessPoint accessPointRangeAddress =
 MessagingFactory.createAccessPoint(AccessPoint.AccessPointType.NUMBER_RANGE,
 DeliveryType.SMS,"9000,9999");
messagingClient.registerAccessPoint(accessPointRangeAddress);
```

Synchronous Receiving

Use the `MessagingClient.receive` method to synchronously receive messages that UMS makes available to the application. This is a convenient polling method for light-weight clients that do not want the configuration overhead associated with receiving messages asynchronously. When receiving messages without specifying an access point, the application receives messages for any of the access points that it has registered. Otherwise, if an access point is specified, the application receives messages sent to that access point.

Receive is a nonblocking operation. If there are no pending messages for the application or access point, the call returns `null` immediately. Receive is not guaranteed to return all available messages, but may return only a subset of available messages for efficiency reasons.

> **Note:**
>
> A single invocation does not guarantee retrieval of all available messages. You must poll in a loop until you receive `null` to ensure receiving all available messages.

Asynchronous Receiving

When asynchronously receiving messages, the client application registers an access point and specifies a `Listener` object and an optional correlator object. When incoming messages arrive at the specified access point address, the listener' s `onMessage` callback is invoked. The originally-specified correlator object is also passed to the callback method.

- Creating a Listener
- Default Message Listener
- Per Access Point Message Listener

Creating a Listener

You create a listener by implementing the `oracle.sdp.messaging.Listener` interface. You can implement it as any concrete class - one of your existing classes, a new class, or an anonymous or inner class.

The following code example shows how to implement a message listener:

```
import oracle.sdp.messaging.Listener;

    public class MyListener implements Listener {

      @Override
      public void onMessage(Message message, Serializable correlator) {
          System.out.println("Received Message: " + message + " with optional
correlator: " +
correlator);
      }
      @Override
      public void onStatus(Status status, Serializable correlator) {
          System.out.println("Received Status: " + status + " with optional correlator:
" +
correlator);
      }

    }
```

You pass a reference to the Listener object to the `setMessageListener` or `registerAccessPoint` methods, as described in "Default Message Listener" and "Per Access Point Message Listener". When a message arrives for your application, the UMS infrastructure invokes the Listener's `onMessage` method.

Default Message Listener

The client application typically sets a default message listener (Example 2-20). When Oracle UMS receives messages addressed to any access points registered by this client application, it invokes the `onMessage` callback for the client application's default listener, unless there is a specific listener registered for the Access Point that corresponds to the received message.

To remove a default listener, call this method with a null argument.

Example 2-20 Default Message Listener

```
messagingClient.setMessageListener(new MyListener());
```

See the sample application `usermessagingsample-echo` for detailed instructions on asynchronous receiving.

Per Access Point Message Listener

The client application can also register an access point and specify a `Listener` object and an optional `correlator` object (Example 2-21). When incoming messages arrive at the specified access point address, the specified listener' s `onMessage` method is invoked. The originally-specified `correlator` object is also passed to the callback method.

> **Note:**
>
> Oracle UMS uses a weak reference when storing the `Listener` object. This means that the client application is responsible for keeping a reference to the `Listener` object to prevent it from being garbage collected.

Example 2-21 Per Access Point Message Listener

```
mlistener = new MyListener();
messagingClient.registerAccessPoint(accessPoint, mlistener, null);
```

Message Filtering

A `MessageFilter` is used by an application to exercise greater control over what messages are delivered to it. A `MessageFilter` contains a matching criterion and an action. An application can register a series of message filters; they are applied in order against an incoming (received) message; if the criterion matches the message, the action is taken. For example, an application can use `MessageFilters` to implement necessary blacklists, by rejecting all messages from a given sender address. If no filters match the message, the default action is to accept the message and deliver it to the application.

Use `MessagingFactory.createMessageFilter` to create a message filter, and `MessagingClient.registerMessageFilter` to register it. The filter is added to the end of the current filter chain for the application. For example, to reject a message with the subject `"spam"`:

```
MessageFilter subjectFilter = MessagingFactory.createMessageFilter("spam",
 MessageFilter.FieldType.SUBJECT, null, MessageFilter.Action.REJECT);
messagingClient.registerMessageFilter(subjectFilter);
```

To reject messages from email address `spammer@foo.com`:

```
MessageFilter senderFilter =
 MessagingFactory.createBlacklistFilter("spammer@foo.com");
messagingClient.registerMessageFilter(senderFilter);
```

Configuring for a Cluster Environment

The API supports an environment where client applications and the UMS server are deployed in a cluster environment.

For a clustered deployment to function as expected, client applications must be configured correctly. The following rules apply:

- Two client applications are considered to be instances of the same application if they use the same `ApplicationName` configuration parameter when creating the UMS Messaging Client. If not set explicitly, UMS will use the client application's deployment name as the `ApplicationName`.
- Instances of the same application share most of their configuration, and artifacts such as `Access` Points and Message Filters that are registered by one instance are shared by all instances.
- The `ApplicationInstanceName` configuration parameter enables you to distinguish instances from one another. Typically this parameter is synthesized by the API implementation, and does not need to be populated by the application developer. Refer to the Javadoc for cases in which this value must be populated.
- Listener correlators are instance-specific. If two different instances of an application register listeners and supply different correlators, then when instance A's listener is invoked, correlator A is supplied; when instance B's listener is invoked, correlator B is supplied.

Using UMS Client API for XA Transactions

UMS provides support for XA enabled transactions for outbound and inbound messages. The industry standard, X/Open XA protocol, is widely supported in other Oracle products such as Business Process Management (BPM).

> **Note:**
>
> You do not need to install the XA support feature, as this feature is included in the UMS server and in the UMS client. Also note that the XA support is available only for the POJO API, not for the Web Services API.

About XA Transactions

JMS services defines a common set of enterprise messaging concepts and facilities. It is used in User Messaging Service (UMS) for messaging, queuing, sorting, and routing. Java Transaction API (JTA) specifies local Java interfaces between a transaction manager and the

parties involved in a distributed transaction system - the application, the resource manager, and the application server. The JTA package consists of the following three components:

- A high-level application interface that allows a transactional application to demarcate transaction boundaries.
- A Java mapping of the industry standard X/Open XA protocol that allows a transactional resource manager to participate in a global transaction controlled by an external transaction manager.
- A high-level transaction manager interface that allows an application server to control transaction boundary demarcation for an application being managed by the application server.

JTA is used by a JMS services provider to support XA transactions (also known as distributed transactions). The JMS provider that supports XA Resource interface is able to participate as a resource manager in a distributed transaction processing system that uses a two-phase commit transaction protocol.

Sending and Receiving XA Enabled Messages

The XA support enables UMS to send messages from within a transaction boundary only when the transaction is committed. If the transaction is rolled back, then the sending of the message fails. A commit leads to a successful transaction; whereas rollback leaves the message unaltered. UMS provides XA transaction support for both outbound and inbound messages.

Outbound messaging using XA

The messages sent from a UMS client application to recipients via UMS server are called outbound messages. When an XA transaction is enabled on a UMS client, an outbound message is sent to the UMS server, only *if* the transaction is committed. Upon successful transaction, the message is safely stored and prepared for delivery to the recipients. If the client transaction fails to commit and a rollback occurs, then the message is not sent to the UMS server for delivery.

The following code snippet demonstrates how to send an outbound message using XA:

```
transaction.begin();
// Some business logic
// ...
String messageID = mClient.send(message);
// Some business logic
// ...
transaction.commit();
```

Inbound messaging using XA

The messages received by a UMS driver, processed by the UMS Server Engine, and routed to a UMS client are called inbound messages. When an XA transaction is enabled on a UMS client, an inbound message is retrieved from the UMS server and deleted from UMS server store, only *if* the transaction is committed. If a transaction rollback occurs, then the message is left unaltered in the UMS server for later redelivery.

The following code snippet demonstrates how to receive an inbound message using XA:

```
transaction.begin();
messages = mClient.receive();

           for (Message receivedMessage : messages) {
            // process individual messages here.
```

```
}
transaction.commit();
```

To receive messages that failed to commit due to a server crash, the server and the client must be restarted, or the specific server migration procedure must be executed. For more information, see chapter Configuring Advanced JMS System Resources in *Oracle Fusion Middleware Configuring and Managing JMS for Oracle WebLogic Server*.

Using a listener for XA transactions

You can also use a listener in a transaction while receiving messages. This is done by specifying the constant `MessagingConstants.LISTENER_TRANSACTED_MODE`. Set the value of this constant to `TRUE` or `FALSE` when creating a `MessagingClient` instance, as shown in the example below.

> **Note:**
>
> If you use a listener, transactions will be committed when the messaging constant `LISTENER_TRANSACTED_MODE` is set to `TRUE` and when no exceptions are raised. When `LISTENER_TRANSACTED_MODE` is set to `FALSE`, transactions will be committed irrespective of the exceptions.
>
> If you want to roll back a transaction, set the exception accordingly. For more information about `ListenerException`, see *User Messaging Service Java API Reference*.

Example 2-22 Using a listener to receive XA enabled messages

```
Map<String, Object> params = new HashMap<String, Object>();
params.put(MessagingConstants.LISTENER_TRANSACTED_MODE, Boolean.TRUE);
MessagingClient mClient = MessagingClientFactory.createMessagingClient(params);

mListener = new MyListener();
mClient.registerAccessPoint(MessagingFactory.createAccessPoint(receiverAddr), mListener,
null);

private class MyListener implements Listener {

    @Override
    public void onMessage(Message message,
    Serializable correlator) throws ListenerException {

     }}
```

For more information about the messaging constant, see *User Messaging Service Java API Reference*.

Using EJB calls for XA transactions

You can send XA enabled messages using EJB calls. To roll back the transaction, specify the `setRollbackOnly()` method. For more information about this method, see: http://docs.oracle.com/javaee/7/api/javax/ejb/EJBContext.html#setRollbackOnly()

You can also control the scope of a transaction by specifying the transaction attributes (such as NotSupported, RequiresNew, and Never) as described in the Jakarta EE tutorial at:

http://docs.oracle.com/javaee/6/tutorial/doc/bncij.html

Example 2-23 Sending XA enabled messaging using an EJB call

```
Map<String, Object> params = new HashMap<String, Object>();
MessagingClient mClient =
MessagingClientFactory.createMessagingClient(params);
MimeMultipart mp = new MimeMultipart("alternative");
MimeBodyPart part1 = new MimeBodyPart();
Message message = MessagingFactory.createMessage();
...
...

mClient.sendMessage();

if(failure)
setRollbackOnly()
```

Using UMS Java API to Specify Message Resends

When a message send attempt is classified as a complete failure, that is, the failover chain is exhausted, the message is automatically scheduled for resend by the UMS Server. This is repeated until the message is successfully sent or the configured number of resends is reached.

However, using the UMS Java API it is possible to override the number of resends on a per message basis by calling the `setMaxResend` method as illustrated in the following example:

```
MessageInfo msgInfo = message.getMessageInfo();
msgInfo.setMaxResend(1);
String mid = messagingClient.send(message);
```

When you examine the status of a sent message as explained in Retrieving Message Status, you get information about both the failover chain and the resends by calling `getTotalFailovers()`/`getFailoverOrder()` and `getMaxResend()`/`getCurrentResend()` on the `Status` object. When failover order equals total failovers, the API user knows that the failover chain is exhausted. However, the resend functionality works as a loop over the failover chain. When `maxResend` equals `currentResend` and failover order equals total failovers then the resend and failover chain is completely exhausted.

For more information about `setMaxResend`, `getTotalFailovers` , `getFailoverOrder`, and other methods, see *User Messaging Service Java API Reference*.

Selecting a Driver Programmatically

Unless multiple drivers of the same type is used, the UMS Server engine selects a driver based on the delivery type that is defined in the UMS Message sent by the client application.

For example, in a message, if the recipient address is "EMAIL:john@example.com" the Email driver is selected, if the recipient address is "SMS:1234" the SMS driver is selected, if the recipient address is "uri:twitter" then, Twitter driver is selected and so on.

The `DeliveryType` enum defines the delivery channel for a message. For more information, see *User Messaging Service Java API Reference*.

However, if the system topology requires multiple instances of a driver, for instance, two Email drivers configured with different Email servers, the outgoing UMS Message can be created in such a way that a specific driver is selected by the UMS Server engine. To do this, you must ensure that the following properties for the message in the client application maps to the UMS Driver configuration settings:

UMS Message Property	UMS Driver Configuration Parameter
`DeliveryType`	`SupportedDeliveryTypes` The recipient address delivery type must match `SupportedDeliveryTypes`. Otherwise, the driver is not selected.
`ContentType`	`SupportedContentTypes` The `ContentType` object in the UMS message must match the `SupportedContentTypes` parameter in the driver configuration. Otherwise, the driver is not selected.
Sender Address	`SenderAddresses` If a value is defined for the `SenderAddresses` parameter in the driver configuration, then the UMS Message Sender Address must match with that value. Otherwise, the driver is not selected.
`MessageInfo` - protocol	`SupportedProtocols` If the `MessageInfo` object in the UMS Message has a value set for `protocol`, then that protocol should be same as the one defined for the `SupportedProtocols` parameter in the driver configuration. Otherwise, the driver is not selected.
`MessageInfo` - carriers	`SupportedCarriers` If the `MessageInfo` object in the UMS Message has a value set for `carrier`, then that carrier should be same as the one defined for the `SupportedCarriers` parameter in the driver configuration. Otherwise, the driver is not selected.
`MessageInfo` - application name	`SupportedApplicationName` If the `MessageInfo` object in the UMS Message has a value set for `Application Name`, then that application name should be same as the one defined for the `SupportedApplicationName` parameter in the driver configuration. Otherwise, the driver is not selected.

Note:

If no driver passes the above conditions, a failure status is returned to the application and a WARNING log is also generated.

If multiple drivers pass the above conditions, one of them is chosen by the UMS engine.

If exactly one driver passes the above conditions then that driver is selected.

For more information about the driver configuration parameters, see "Configuring User Messaging Service Drivers" in *Administering Oracle User Messaging Service*.

Setting up Priority and Expiration time for a Message

In certain circumstances such as a high load, you may benefit from specifying UMS Message priority when sending a message.

The message priority is used in the internal JMS queues. Also, if the protocol implemented by the driver supports priority, the UMS Message priority is translated to the corresponding protocol priority. Specify the priority as illustrated in the following example:

Example 2-24 Setting up high priority for a message

```
MessageInfo msgInfo = message.getMessageInfo();
msgInfo.setPriority(MessagePriorityType.HIGH);
String mid = messagingClient.send(message);
```

For information about all available priority types, see `MessagePriorityType` definition in *User Messaging Service Java API Reference*.

The expiration time can be set for a message if the message is only valid for a limited period of time. Note that it depends on the underlying messaging protocol if the expiration setting is honored or not, see table Table 2-3

Example 2-25 Setting up expiry time for a message

```
MessageInfo msgInfo = message.getMessageInfo();
msgInfo.setExpiration(3600);
String mid = messagingClient.send(message);
```

Table 2-3 shows which drivers, that is, underlying messaging protocols, that support message priority and/or message expiration:

Table 2-3 Message Properties

Driver	Expiry (Yes/No)	Priority (Yes/No)
SMPP	Yes	Yes
XMPP	No	No
Email	No	Yes
Extension	No	No
Twitter	No	No
GCM	Yes	No
APNS	Yes	No

Specifying User Preference Application Partitioning Profile ID

User Communication Preferences are invoked for recipient addresses that contains the USER prefix (for example, USER:john.doe). The preferences can be partitioned into profiles.

Below is an example of how to specify a profile when sending a message:

Example 2-26 Specifying application partitioning profile id

```
Address recipient = MessagingFactory.createAddress("USER:john.doe");
message.addRecipient(recipient);
MessageInfo msgInfo = message.getMessageInfo();
msgInfo.setProfileId("myProfileId");
String mid = messagingClient.send(message);
```

For information about User Communication Preferences and profiles, see *Administering User Communication Preferences*.

Configuring Security

Client applications may need to specify one or more additional configuration parameters to establish a secure listener.

For more information, see Table 2-1.

Threading Model

Client applications that use the UMS Java API are usually multi-threaded. Typical scenarios include a pool of EJB instances, each of which uses a `MessagingClient` instance; and a servlet instance that is serviced by multiple threads in a web container.

The UMS Java API supports the following thread model:

- Each call to `MessagingClientFactory.createMessagingClient` returns a new `MessagingClient` instance.
- When two `MessagingClient` instances are created by passing parameter maps that are equal to `MessagingClientFactory.createMessagingClient`, they are instances of the same client. Instances created by passing different parameter maps are instances of separate clients.
- An instance of `MessagingClient` is not thread safe when it has been obtained using `MessagingClientFactory.createMessagingClient`. Client applications must ensure that a given instance is used by only one thread at a time. They may do so by ensuring that an instance is only visible to one thread at a time, or by synchronizing access to the `MessagingClient` instance.
- Two instances of the same client (created with identical parameter maps) do share some resources – notably they share Message and Status Listeners, and use a common pool of Worker threads to execute asynchronous messaging operations. For example, if instance A calls `setMessageListener()`, and then instance B calls `setMessageListener()`, then B's listener is the active default message listener.

The following are typical use cases:

- To use the UMS Java API from an EJB (either a Message Driven Bean or a Session Bean) application, the recommended approach is to create a `MessagingClient` instance in the bean' s `ejbCreate` (or equivalent `@PostConstruct`) method, and store the `MessagingClient` in an instance variable in the bean class. The EJB container ensures that only one thread at a time uses a given EJB instance, which ensures that only one thread at a time accesses the bean' s `MessagingClient` instance.
- To use the UMS Java API from a Servlet, there are several possible approaches. In general, web containers create a single instance of the servlet class, which may be accessed by multiple threads concurrently. If a single `MessagingClient` instance is created and stored in a servlet instance variable, then access to the instance must be synchronized.

  Another approach is to create a pool of `MessagingClient` instances that are shared among servlet threads.

  Finally, you can associate individual `MessagingClient` instances with individual HTTP Sessions. This approach allows increased concurrency compared to having a single `MessagingClient` for all servlet requests. However, it is possible for multiple threads to

access an HTTP Session at the same time due to concurrent client requests, so synchronization is still required in this case.

- Listener Threading

Listener Threading

For asynchronous receiving described in Asynchronous Receiving of Message Status and Asynchronous Receiving UMS by default uses one thread for incoming messages and one thread for incoming status notifications (assuming at least one message or status listener is registered, respectively). Client applications can increase the concurrency of asynchronous processing by configuring additional worker threads. This is done by specifying integer values for the `MessagingConstants.MESSAGE_LISTENER_THREADS` and `MessagingConstants.STATUS_LISTENER_THREADS` keys, settings these values to the desired number of worker threads in the configuration parameters used when creating a `MessagingClient` instance. In this case, the application's listener must be written to handle multi-threaded execution.

3

Sending and Receiving Messages using the User Messaging Service Web Service API

This chapter describes how to use the User Messaging Service (UMS) Web Service API to develop applications. This API serves as a programmatic entry point for Fusion Middleware application developers to implement UMS messaging applications that run in a remote container relative to the UMS server.

> **Note:**
>
> To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, see the samples at:
>
> `http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html`

- Introduction to the UMS Web Service API
  The UMS Web Service API is functionally identical to the Java API. The JAX-WS and JAXB bindings of the web service types and interfaces have similar names as the corresponding Java API classes. That means, the client code looks the same for both the UMS Java API and UMS Web Service API. However, the API classes are in their own package spaces and the classes from these two APIs are not interoperable.
- Creating a UMS Client Instance and Specifying Runtime Parameters
  The `MessagingClient` object is the fundamental UMS API object that you must create in the Client application using the UMS Web Service API.
- Sending a Message
  Invoking the `send` method of `MessagingClient` object sends the message in a Web Service request to the UMS Server, where it is processed accordingly.
- Retrieving Message Status
  After sending a message, you can use Oracle UMS to retrieve the message status either synchronously or asynchronously.
- Receiving a Message
  This section describes how an application receives messages.
- Configuring for a Cluster Environment
  The UMS Web Services API supports an environment where client applications and the UMS server are deployed in a cluster environment.
- Using UMS Web Service API to Specify Message Resends
  When a message send attempt is classified as a complete failure, that is, the failover chain is exhausted, the message is automatically scheduled for resend by the UMS Server. This is repeated until the message is successfully sent or the configured number of resends is reached.
- Configuring Security
  This section contains information related to configuring security.

- Threading Model
  Instances of the Web Services `MessagingClient` class are not thread-safe due to the underlying services provided by the JAX-WS stack.

Introduction to the UMS Web Service API

The UMS Web Service API is functionally identical to the Java API. The JAX-WS and JAXB bindings of the web service types and interfaces have similar names as the corresponding Java API classes. That means, the client code looks the same for both the UMS Java API and UMS Web Service API. However, the API classes are in their own package spaces and the classes from these two APIs are not interoperable.

The UMS Web Service API consists of packages grouped as follows:

- Common and Client Packages
  - `oracle.ucs.messaging.ws`
  - `oracle.ucs.messaging.ws.types`
- Web Service API Web Service Definition Language (WSDL) files:
  - `messaging.wsdl`: defines the operations invoked by a web service client.
  - `listener.wsdl`: defines the callback operations that a client must implement to receive asynchronous message or status notifications.

The samples with source code are available on Oracle Technology Network (OTN).

Creating a UMS Client Instance and Specifying Runtime Parameters

The `MessagingClient` object is the fundamental UMS API object that you must create in the Client application using the UMS Web Service API.

You can create a instance of `oracle.ucs.messaging.ws.MessagingClient` by using the public constructor. Client applications can specify a set of parameters at runtime when instantiating a client object. For example, you configure a `MessagingClient` instance by specifying parameters as a map of key-value pairs in a `java.util.Map<String, Object>`. Among other things, the configuration parameters serve to identify the web service endpoint URL identifying the UMS server to communicate with, and other web service-related information such as security policies. Client applications are responsible for storing and loading the configuration parameters using any available mechanism.

You are responsible for mapping the parameters to or from whatever configuration storage mechanism is appropriate for your deployment. The `MessagingClient` class uses the specified key/value pairs for configuration, and passes through all parameters to the underlying JAX-WS service. Any parameters recognized by JAX-WS are valid. Table 3-1 lists the most common configuration parameters:

Table 3-1 Configuration Parameters Specified at Runtime

Key	Type	Use
`javax.xml.ws.BindingProvider.ENDPOINT_ADDRESS_PROPERTY`	String	Endpoint URL for the remote UMS WS. This is typically "http://<host>:<port>/ucs/messaging/webservice".

Table 3-1 (Cont.) Configuration Parameters Specified at Runtime

Key	Type	Use
`javax.xml.ws.BindingProvider.USERNAME_PROPERTY`	String	Username to be asserted in WS-Security headers when relevant
`oracle.ucs.messaging.ws.ClientConstants.POLICIES`	String[]	Array of OWSM WS-Security policies to attach to the client's requests. These must match the policies specified on the server side.
`oracle.wsm.security.util.SecurityConstants.Config.KEYSTORE_RECIPIENT_ALIAS_PROPERTY`	String	Used for OWSM policy attachment. Specifies an alternate alias to use for looking up encryption and signing keys from the credential store.
`oracle.wsm.security.util.SecurityConstants.ClientConstants.WSS_CSF_KEY`	String	Used for OWSM policy attachment. Specifies a credential store key to use for looking up remote username/password information from the Oracle Web Services Management credential store map.

To know more about the OWSM policy parameters, see *Java API Reference for Oracle Web Services Manager*.

A `MessagingClient` cannot be reconfigured after it is instantiated. Instead, a new instance of the `MessagingClient` class must be created using the new configuration.

Example 3-1 shows code for creating a `MessagingClient` instance using username/token security.

Example 3-1 Creating a MessagingClient Instance, Username/Token Security

```
HashMap<String, Object> config = new HashMap<String, Object>();
config.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
 "http://example.com:8001/ucs/messaging/webservice");
config.put(ClientConstants.POLICIES, new String[] {"oracle/wss11_username_token_
with_message_protection_client_policy"});
config.put(BindingProvider.USERNAME_PROPERTY, "user1");
config.put(oracle.wsm.security.util.SecurityConstants.Config.CLIENT_CREDS_
LOCATION, oracle.wsm.security.util.SecurityConstants.Config.CLIENT_CREDS_LOC_
SUBJECT);
config.put(oracle.wsm.security.util.SecurityConstants.ClientConstants.WSS_CSF_KEY,
 "user1-passkey");
config.put(MessagingConstants.APPLICATION_NAME, "MyUMSWSApp");
mClient = new MessagingClient(config);
```

Example 3-2 shows code for creating a `MessagingClient` instance using SAML token security.

Example 3-2 Creating a MessagingClient Instance, SAML Token Security

```
HashMap<String, Object> config = new HashMap<String, Object>();
config.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
 "http://example.com:8001/ucs/messaging/webservice");
config.put(ClientConstants.POLICIES, new String[] {"oracle/wss11_saml_token_
identity_switch_with_message_protection_client_policy"});
config.put(BindingProvider.USERNAME_PROPERTY, "user1");
config.put(oracle.wsm.security.util.SecurityConstants.Config.CLIENT_CREDS_
LOCATION, oracle.wsm.security.util.SecurityConstants.Config.CLIENT_CREDS_LOC_
SUBJECT);
config.put(oracle.wsm.security.util.SecurityConstants.Config.KEYSTORE_RECIPIENT_
```

```
ALIAS_PROPERTY, "example.com");
config.put(MessagingConstants.APPLICATION_NAME, "MyUMSWSApp");
mClient = new MessagingClient(config);
```

A `MessagingClient` cannot be reconfigured after it is instantiated. Instead, you must create a new instance of the `MessagingClient` class using the desired configuration.

Factory methods are provided for creating Web Service API types in the class "`oracle.ucs.messaging.ws.MessagingFactory`".

Sending a Message

Invoking the `send` method of `MessagingClient` object sends the message in a Web Service request to the UMS Server, where it is processed accordingly.

The `send` method returns a `String message` identifier that the client application can later use to retrieve message delivery status, or to correlate with asynchronous status notifications that are delivered to a Listener. The status returned is the latest known status based on UMS internal processing and delivery notifications received from external gateways.

The types of messages that can be created include plaintext messages, multipart messages that can consist of text/plain and text/html parts, and messages that include the creation of delivery channel (`DeliveryType`) specific payloads in a single message for recipients with different delivery types.

- Creating a Message
- API Reference for Interface Message
- API Reference for Enum DeliveryType
- Creating an Address
- User Preferences in Messages

Creating a Message

This section describes the various types of messages that can be created.

- Creating a Plaintext Message
- Creating a Multipart/Mixed Message (with Text and Binary Parts)
- Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts)
- Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types

Creating a Plaintext Message

Example 3-3 shows two ways to create a plaintext message using the UMS Web Service API.

Example 3-3 Creating a Plaintext Message Using the UMS Web Service API

```
Message message = MessagingFactory.createTextMessage("This is a Plain Text
 message.");

or

Message message = MessagingFactory.createMessage();
```

```
message.setContent(new DataHandler(new StringDataSource("This is a Plain Text
 message.", "text/plain; charset=UTF-8")));
```

Creating a Multipart/Mixed Message (with Text and Binary Parts)

Example 3-4 shows how to create a multipart/mixed message using the UMS Web Service API.

Example 3-4 Creating a Multipart/Mixed Message Using the UMS Web Service API

```
Message message = MessagingFactory.createMessage();
MimeMultipart mp = new MimeMultipart("mixed");

// Create the first body part
MimeBodyPart mp_partPlain = new MimeBodyPart();
StringDataSource plainDS = new StringDataSource("This is a Plain Text part.",
 "text/plain; charset=UTF-8");
mp_partPlain.setDataHandler(new DataHandler(plainDS));
mp.addBodyPart(mp_partPlain);

byte[] imageData;
// Create or load image data in the above byte array (code not shown for brevity)

// Create the second body part
MimeBodyPart mp_partBinary = new MimeBodyPart();
ByteArrayDataSource binaryDS = new ByteArrayDataSource(imageData, "image/gif");
mp_partBinary.setDataHandler(binaryDS);
mp.addBodyPart(mp_partBinary);

message.setContent(new DataHandler(mp, mp.getContentType()));
```

Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts)

Example 3-5 shows how to create a multipart/alternative message using the UMS Web Service API.

Example 3-5 Creating a Multipart/Alternative Message Using the UMS Web Service API

```
Message message = MessagingFactory.createMessage();
MimeMultipart mp = new MimeMultipart("alternative");
MimeBodyPart mp_partPlain = new MimeBodyPart();
StringDataSource plainDS = new StringDataSource("This is a Plain Text part.", "text/
plain; charset=UTF-8");
mp_partPlain.setDataHandler(new DataHandler(plainDS));
mp.addBodyPart(mp_partPlain);

MimeBodyPart mp_partRich = new MimeBodyPart();
StringDataSource richDS = new StringDataSource(
    "<html><head></head><body><b><i>This is an HTML part.</i></b></body></html>",
    "text/html");
mp_partRich.setDataHandler(new DataHandler(richDS));
mp.addBodyPart(mp_partRich);

message.setContent(new DataHandler(mp, mp.getContentType()));
```

Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types

When sending a message to multiple recipients, there could be multiple channels involved. Oracle UMS application developers are required to specify the correct multipart format for each channel.

Example 3-6 shows how to create delivery channel (`DeliveryType`) specific payloads in a single message for recipients with different delivery types.

Each top-level part of a multiple payload multipart/alternative message should contain one or more values of this header. The value of this header should be the name of a valid delivery type. Refer to the available values for *DeliveryType* in the enum `DeliveryType`.

Example 3-6 Creating Delivery Channel-specific Payloads in a Single Message for Recipients with Different Delivery Types

```
Message message = MessagingFactory.createMessage();

// create a top-level multipart/alternative MimeMultipart object.
MimeMultipart mp = new MimeMultipart("alternative");

// create first part for SMS payload content.
MimeBodyPart part1 = new MimeBodyPart();
part1.setDataHandler(new DataHandler(new StringDataSource("Text content for SMS.",
 "text/plain; charset=UTF-8")));
part1.setHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "SMS");
// add first part
mp.addBodyPart(part1);

// create second part for EMAIL and IM payload content.
MimeBodyPart part2 = new MimeBodyPart();
MimeMultipart part2_mp = new MimeMultipart("alternative");
MimeBodyPart part2_mp_partPlain = new MimeBodyPart();
part2_mp_partPlain.setDataHandler(new DataHandler(new StringDataSource("Text
 content for EMAIL/IM.", "text/plain; charset=UTF-8")));
part2_mp.addBodyPart(part2_mp_partPlain);
MimeBodyPart part2_mp_partRich = new MimeBodyPart();
part2_mp_partRich.setDataHandler(new DataHandler(new
 StringDataSource("<html><head></head><body><b><i>" + "HTML content for EMAIL/IM."
 +
    "</i></b></body></html>", "text/html; charset=UTF-8")));
part2_mp.addBodyPart(part2_mp_partRich);
part2.setContent(part2_mp, part2_mp.getContentType());
part2.addHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "EMAIL");
part2.addHeader(Message.HEADER_NS_PAYLOAD_PART_DELIVERY_TYPE, "IM");
// add second part
mp.addBodyPart(part2);

// set the content of the message
message.setContent(new DataHandler(mp, mp.getContentType()));

// set the MultiplePayload flag to true
MimeHeader multiHeader = new MimeHeader();
multiHeader.setName(oracle.sdp.client.Message.HEADER_SDPM_MULTIPLE_PAYLOAD);
multiHeader.setValue(Boolean.TRUE.toString());
message.getHeaders().add(multiHeader);
```

API Reference for Interface Message

The API reference for interface Message can be accessed from the Javadoc.

API Reference for Enum DeliveryType

The API reference for enum `DeliveryType` can be accessed from the User Messaging Service Java API Reference.

Creating an Address

For information about the types of addresses and to understand how to create `Address` objects and to define failover address, see Creating an Address.

See *User Messaging Service Java API Reference* to know more about the `Address` interface.

- Recipient Types
- API Reference for Class MessagingFactory

Recipient Types

The WS API provides support for sending and receiving messages with To/Cc/Bcc recipients for use with the email driver:

- To send a message and specify a Cc/Bcc recipient, create the `oracle.ucs.messaging.ws.Address` object using `oracle.ucs.messaging.ws.MessagingFactory.buildAddress` method. The arguments are the address value (for example, user@domain.com), delivery type (for example, `DeliveryType.EMAIL`), and email mode (for example, "Cc" or "Bcc").
- To determine the recipient type of an existing address object, for example in a received message, use the `oracle.ucs.messaging.ws.MessagingFactory.getRecipientType` method, passing it the `Address` object. It returns a string indicating the recipient type.

API Reference for Class MessagingFactory

See User Messaging Service Java API Reference for information about the class `MessagingFactory`.

User Preferences in Messages

When you create a message using WS API, you can also supply facts for business terms for the user preferences in that message. See User Preference Based Messaging for more details.

Retrieving Message Status

After sending a message, you can use Oracle UMS to retrieve the message status either synchronously or asynchronously.

- Synchronous Retrieval of Message Status
- Asynchronous Receiving of Message Status

Synchronous Retrieval of Message Status

To perform a synchronous retrieval of current status, use the following flow from the `MessagingClient` API:

```
String messageId = messagingClient.send(message);
List<Status> statuses = messagingClient.getStatus(messageId, null)
```

or,

```
List<Status> statuses = messagingClient.getStatus(messageId, addresses) --- where
addresses is a "List<Address>" of one or more of the recipients set in the message.
```

Asynchronous Receiving of Message Status

To receive statuses asynchronously, a client application must implement the listener web service as described in `listener.wsdl`. There is no constraint on how the listener endpoint must be implemented. For example, one method is to use the `javax.xml.ws.Endpoint` JAX-WS Service API to publish a web service endpoint. This mechanism is available in Java SE 6 and does not require the consumer to explicitly define a Jakarta EE servlet module.

However, a servlet-based listener implementation is acceptable as well.

When sending a message, the client application can provide a reference to the listener endpoint, consisting of the endpoint URL and a SOAP interface name. As statuses are generated during the processing of the message, the UMS server invokes the listener endpoint' s `onStatus` method to notify the client application.

- Creating a Listener
- Publish the Callback Service
- Stop a Dynamically Published Endpoint
- Registration

Creating a Listener

You create a listener by implementing the `oracle.ucs.messaging.ws.Listener` interface. You can implement it as any concrete class - one of your existing classes, a new class, or an anonymous or inner class.

The following code example shows how to implement a status listener:

```
@PortableWebService(serviceName="ListenerService",
targetNamespace="http://xmlns.oracle.com/ucs/messaging/",
endpointInterface="oracle.ucs.messaging.ws.Listener",
wsdlLocation="META-INF/wsdl/listener.wsdl",
portName="Listener")
public class MyListener implements Listener {
  public MyListener() {
  }

  @Override
  public void onMessage(Message message, byte[] correlator) throws MessagingException {
    System.out.println("I got a message!");
  }  @Override
  public void onStatus(Status status, byte[] correlator) throws MessagingException {
    System.out.println("I got a status!");
```

```
    }
}
```

Publish the Callback Service

To publish the callback service, you can either declare a servlet in web.xml in a web module within your application, or use the JAX-WS javax.xml.ws.Endpoint class's publish method to programmatically publish a WS endpoint (Example 3-7):

Example 3-7 Publish the Callback Service

```
Listener myListener = new MyListener();
String callbackURL = "http://host:port/umswscallback";
Endpoint myEndpoint = javax.xml.ws.Endpoint.publish(callbackURL, myListener);
```

Stop a Dynamically Published Endpoint

To stop a dynamically published endpoint, call the `stop()` method on the Endpoint object returned from `Endpoint.publish()` (Example 3-8).

Example 3-8 Stop a Dynamically Published Endpoint

```
// When done, stop the endpoint, ideally in a finally block or other reliable cleanup
mechanism
myEndpoint.stop();
```

Registration

Once the listener web service is published, you must register the fact that your client has such an endpoint. There are the following relevant methods in the MessagingClient API:

- `setStatusListener(ListenerReference listener)`
- `send(Message message, ListenerReference listener, byte[] correlator)`

`setStatusListener()` registers a "default" status listener whose callback is invoked for any incoming status messages. A listener passed to `send()` is only invoked for status updates related to the corresponding message.

Receiving a Message

This section describes how an application receives messages.

An application that wants to receive incoming messages must register one or more access points that represent the recipient addresses of the messages. The server matches the recipient address of an incoming message against the set of registered access points, and routes the incoming message to the application that registered the matching access point. From the application perspective there are two modes for receiving a message, synchronous and asynchronous.

- Registering an Access Point
- Synchronous Receiving
- Asynchronous Receiving
- Message Filtering

Registering an Access Point

`AccessPoint` represents one or more device addresses for receiving incoming messages. For more details about access points, see Registering an Access Point.

Synchronous Receiving

Use the method `MessagingClient.receive` to synchronously receive messages that UMS makes available to the application. This is a convenient polling method for light-weight clients that do not want the configuration overhead associated with receiving messages asynchronously.

> **Note:**
>
> In a multi UMS server deployment, the WS API user must make sure “receive" is called on all UMS servers. This could be done by for instance making sure the Load Balancer uses a round-robin algorithm. Or, simply do not use receive in this use-case, use asynchronous receiving using a Listener as describe in Asynchronous Receiving instead.

Receive is a nonblocking operation. If there are no pending messages for the application or access point, the call returns immediately with an empty list. Receive is not guaranteed to return all available messages, but may return only a subset of available messages for efficiency reasons.

> **Note:**
>
> A single invocation does not guarantee retrieval of all available messages. You must poll to ensure receiving all available messages.

Asynchronous Receiving

To receive messages asynchronously, a client application must implement the `Listener` web service as described in `listener.wsdl`. There is no constraint on how the listener endpoint must be implemented. For example, one mechanism is using the `javax.xml.ws.Endpoint` JAX-WS Service API to publish a web service endpoint. This mechanism is available in Java SE 6 and does not require the consumer to explicitly define a Jakarta EE servlet module. However, a servlet-based listener implementation is also acceptable.

- Creating a Listener
- Default Message Listener
- Per Access Point Message Listener

Creating a Listener

You create a listener by implementing the `oracle.ucs.messaging.ws.Listener` interface. You can implement it as any concrete class - one of your existing classes, a new class, or an anonymous or inner class.

The following code example shows how to implement a message listener:

```
@PortableWebService(serviceName="ListenerService",
targetNamespace="http://xmlns.oracle.com/ucs/messaging/",
endpointInterface="oracle.ucs.messaging.ws.Listener",
wsdlLocation="META-INF/wsdl/listener.wsdl",
portName="Listener")
public class MyListener implements Listener {
  public MyListener() {
  }

  @Override
  public void onMessage(Message message, byte[] correlator) throws MessagingException {
    System.out.println("I got a message!");
  }  @Override
  public void onStatus(Status status, byte[] correlator) throws MessagingException {
    System.out.println("I got a status!");
  }
}
```

You pass a reference to the Listener object to the `setMessageListener` or `registerAccessPoint` methods, as described in "Default Message Listener" and "Per Access Point Message Listener". When a message arrives for your application, the UMS infrastructure invokes the Listener's `onMessage` method.

Default Message Listener

The client application typically sets a default message listener (Example 3-9). When Oracle UMS receives messages addressed to any access points registered by this client application, it invokes the `onMessage` callback for the client application's default listener.

To remove a default listener, call this method with a null argument.

Example 3-9 Default Message Listener

```
ListenerReference listenerRef = new ListenerReference();
listenerRef.setEndpoint("url_to_your_webservice_message_listener");
messagingClient.setMessageListener(listenerRef);
```

Per Access Point Message Listener

The client application can also register an access point and specify a `Listener` object and an optional correlator object (Example 3-10). When incoming messages arrive at the specified access point address, the specified listener' s `onMessage` method is invoked. The originally-specified correlator object is also passed to the callback method.

Example 3-10 Per Access Point Message Listener

```
AccessPoint accessPoint =
 MessagingFactory.createAccessPoint(AccessPointType.SINGLE_ADDRESS,
DeliveryType.EMAIL, "test@example.org");
ListenerReference listenerRef = new ListenerReference();
listenerRef.setEndpoint("url_to_your_webservice_message_listener");
```

```
byte[] correlator = null; // Not to correlate the callback
messagingClient.registerAccessPoint(accessPoint, listenerRef, correlator);
```

Message Filtering

A `MessageFilter` is used by an application to exercise greater control over what messages are delivered to it. For more details about creating message filters, see Message Filtering.

Configuring for a Cluster Environment

The UMS Web Services API supports an environment where client applications and the UMS server are deployed in a cluster environment.

For a clustered deployment to function as expected, client applications must be configured correctly as explained in Configuring for a Cluster Environment.

Using UMS Web Service API to Specify Message Resends

When a message send attempt is classified as a complete failure, that is, the failover chain is exhausted, the message is automatically scheduled for resend by the UMS Server. This is repeated until the message is successfully sent or the configured number of resends is reached.

However, using the UMS Web Services API it is possible to override the number of resends on a per message basis by calling the `setMaxResend` method as illustrated in the following example:

```
MessageInfo msgInfo = new oracle.ucs.messages.ws.types.MessageInfo();
msgInfo.setMaxResend(new Integer(1));
// When MessageInfo is created we must also set priority
msgInfo.setPriority(PriorityType.NORMAL);
message.setMessageInfo(msgInfo);
String mid = client.send(message, null, null);
```

The status of the failover addresses can be received by calling `getTotalFailovers()` and `getFailoverOrder()`. When failover order equals total failovers, the API user knows that the failover chain is exhausted. However, the resend functionality works as a loop over the failover chain. You can call `getMaxResend()` and `getCurrentResend()` to know when the resend and failover chain is completely exhausted.

For more information about `setMaxResend`, `getTotalFailovers()` and `getFailoverOrder()` methods, see *User Messaging Service Java API Reference*.

Configuring Security

This section contains information related to configuring security.

- Client and Server Security
- Listener or Callback Security

Client and Server Security

There are two supported security modes for the UMS Web Service: Security Assertions Markup Language (SAML) tokens and username tokens.

The supported SAML-based policy is "oracle/wss11_saml_token_with_message_protection_client_policy". This policy establishes a trust relationship between the client application and the UMS server based on the exchange of cryptographic keys. The client application is then allowed to assert a user identity that is respected by the UMS server. To use SAML tokens for WS-Security, some keystore configuration is required for both the client and the server.

See Example 3-2 for more details about configuring SAML security in a UMS web service client.

The supported username token policy is "oracle/wss11_username_token_with_message_protection_client_policy". This policy passes an encrypted username/password token in the WS-Security headers, and the server authenticates the supplied credentials. It is highly recommended that the username and password be stored in the Credential Store, in which case only a Credential Store key must be passed to the `MessagingClient` constructor, ensuring that credentials are not hard-coded or stored in an unsecure manner. See Example 3-1 for more details about configuring SAML security in a UMS web service client.

For more information about securing web services using Oracle Web Services Manager see *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Listener or Callback Security

Username token and SAML token security are also supported for the Listener callback web services. When registering a listener, the client application must supply additional parameters specifying the security policy and any key or credential lookup information that the server requires to establish a secure connection.

Example 3-11 illustrates how to establish a secure callback endpoint using username token security:

Example 3-11 Establishing a Secure Callback Endpoint Using Username Token Security

```
MessagingClient client = new MessagingClient(clientParameters);
...
ListenerReference listenerRef = new ListenerReference();
// A web service implementing the oracle.ucs.messaging.ws.Listener
// interface must be available at the specified URL.
listenerRef.setEndpoint(myCallbackURL);
Parameter policyParam = new Parameter();
policyParam.setName(ClientConstants.POLICY_STRING);
policyParam.setValue("oracle/
wss11_username_token_with_message_protection_client_policy");
listenerRef.getParameters.add(policyParam);
// A credential store entry with the specified key must be
// provisioned on the server side so it will be available when the callback
// is invoked.
Parameter csfParam = new Parameter();
csfParam.setName(oracle.wsm.security.util.SecurityConstants.ClientConstants.WSS_CSF_KEY);
csfParam.setValue("callback-csf-key");
listenerRef.getParameters.add(csfParam);
client.setMessageListener(listenerRef);
```

Threading Model

Instances of the Web Services `MessagingClient` class are not thread-safe due to the underlying services provided by the JAX-WS stack.

You are responsible for ensuring that each instance is used by only one thread at a time.

A

Using the User Messaging Service Sample Applications

This appendix describes how to create a client application that uses Oracle User Messaging Service (UMS) Java API.

> **Note:**
>
> To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, refer to the samples at:
>
> `http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html`.

- Using the UMS Client API to Build a Client Application
  This section describes how to create an application called *usermessagingsample*, a web client application that uses the UMS Client API for both outbound messaging and the synchronous retrieval of message status. *usermessagingsample* also supports inbound messaging. Once you have deployed and configured *usermessagingsample*, you can use it to send a message to an email client.
- Using the UMS Client API to Build a Client Echo Application
  This section describes how to create an application called usermessagingsample-echo, a demo client application that uses the UMS Client API to asynchronously receive messages from an email address and echo a reply back to the sender.
- Creating a New Application Server Connection
  You define an application server connection in Oracle JDeveloper, and deploy and run the application.
- Sample Chat Application with Web Services APIs
  This section describes how to create, deploy and run the sample chat application with the Web Services APIs provided with Oracle User Messaging Service on OTN.

Using the UMS Client API to Build a Client Application

This section describes how to create an application called *usermessagingsample*, a web client application that uses the UMS Client API for both outbound messaging and the synchronous retrieval of message status. *usermessagingsample* also supports inbound messaging. Once you have deployed and configured *usermessagingsample*, you can use it to send a message to an email client.

This sample focuses on a Web Application Module (WAR), which defines some HTML forms and servlets. You can examine the code and corresponding XML files for the web application module from the provided `usermessagingsample-src.zip` source. The servlets uses the UMS Client API to create an UMS Client instance (which in turn registers the application's information) and sends messages.

This application, which is packaged as a Enterprise ARchive file (EAR) called *usermessagingsample.ear*, has the following structure:

- `usermessagingsample.ear`
  - `META-INF`
    - `application.xml` -- Descriptor file for all of the application modules.
    - `weblogic-application.xml` -- Descriptor file that contains the `import` of the `oracle.sdp.messaging` shared library.
  - `usermessagingsample-web.ear` -- Contains the web-based front-end and servlets.
    - `WEB-INF`
    - `web.xml`
    - `weblogic.xml`

The prebuilt sample application, and the source code (usermessagingsample-src.zip) are available on OTN.

Overview of Development

The following steps describe the process of building an application capable of outbound messaging using `usermessagingsample.ear` as an example:

Configuring the Email Driver

To enable the Oracle User Messaging Service's email driver to perform outbound messaging and status retrieval, when you configure the email driver, enter the name of the SMTP mail server as the value for the `OutgoingMailServer` property.

For more information about configuring the email driver, see *Administering Oracle User Messaging Service*.

> **Note:**
>
> This sample application is generic and can support outbound messaging through other channels when the appropriate messaging drivers are deployed and configured.

Using JDeveloper 14c to Build the Application

This section describes using a Windows-based build of JDeveloper to build, compile, and deploy *usermessagingsample* through the following steps:

- Opening the Project

Opening the Project

1. Open `usermessagingsample.jws` (contained in the usermessagingsample-src.zip file) in Oracle JDeveloper.

   In the Oracle JDeveloper main window, the project appears.

2. To build the sample application, the web module should include the "Oracle UMS Client" library.

   a. In the Application Navigator, right-click web module **usermessagingsample-web**, and select **Project Properties**.

   b. In the left pane, select **Libraries and Classpath**.

   c. Click **OK**.

3. Explore the Java files under the **usermessagingsample-web** project to see how the messaging client APIs are used to send messages, get statuses, and synchronously receive messages. The MessagingClient instance is created in `SampleUtils.java` in the project.

Deploying the Application

Perform the following steps to deploy the application:

1. Create an Application Server Connection by right-clicking the application in the navigation pane and selecting New. Follow the instructions in Creating a New Application Server Connection.
2. Deploy the application by selecting the **usermessagingsample application**, **Deploy**, **usermessagingsample**, **to**, and **ums_server**.
3. Verify that the message `Build Successful` appears in the log.
4. Verify that the message `Deployment Finished` appears in the deployment log.

   You have successfully deployed the application.

   Before you can run the sample, you must configure any additional drivers in Oracle User Messaging Service and optionally configure a default device for the user receiving the message in User Communication Preferences.

> **Note:**
>
> Refer to *Administering Oracle User Messaging Service* for more information.

Testing the Application

Once **usermessagingsample** has been deployed to a running instance of Oracle WebLogic Server, perform the following:

1. Launch a web browser and enter the address of the sample application as follows: `http://host:http-port/usermessagingsample/`. For example, enter `http://localhost:7001/usermessagingsample/` into the browser's navigation bar.

   When prompted, enter login credentials. For example, username `weblogic`. The browser page for testing messaging samples appears.
2. Click **Send sample message**. The Send Message page appears.
3. As an optional step, enter the sender address in the following format:

   `Email:sender_address`.

   For example, enter `Email:sender@example.com`.
4. Enter one or more recipient addresses. For example, enter `Email:recipient@example.com`. Enter multiple addresses as a comma-separated list as follows:

   `Email:recipient_address1, Email:recipient_address2`.

   If you have configured User Communication Preferences, you can address the message simply to `User:username`. For example, `User:weblogic`.
5. As an optional step, enter a subject line or content for the email.
6. Click **Send**. The Message Status page appears, showing the progress of transaction. The **Status Content** field displays **Message received by Messaging engine for processing**.
7. Click **Refresh** to update the status. When the email message has been delivered to the email server, the **Status Content** field displays **Outbound message delivery to remote gateway succeeded.**.

Using the UMS Client API to Build a Client Echo Application

This section describes how to create an application called usermessagingsample-echo, a demo client application that uses the UMS Client API to asynchronously receive messages from an email address and echo a reply back to the sender.

> **Note:**
>
> To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, refer to the Oracle User Messaging Service samples at http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html.

This application, which is packaged as a Enterprise Archive file (EAR) called *usermessagingsample-echo.ear*, has the following structure:

- `usermessagingsample-echo.ear`
  - `META-INF`

    - `application.xml` -- Descriptor file for all of the application modules.
    - `weblogic-application.xml` -- Descriptor file that contains the `import` of the `oracle.sdp.messaging` shared library.
  - `usermessagingsample-echo-web.war` -- Contains the web-based front-end and servlets. It also contains the listener that processes a received message and returns an echo response
    - `WEB-INF`
    - `web.xml`
    - `weblogic.xml`

The prebuilt sample application, and the source code (`usermessagingsample-echo-src.zip`) are available on OTN.

- Overview of Development
- Configuring the Email Driver
- Using Oracle JDeveloper 14c to Build the Application
- Deploying the Application
- Testing the Application

Overview of Development

The following steps describe the process of building an application capable of asynchronous inbound and outbound messaging using `usermessagingsample-echo.ear` as an example:

1. Configuring the Email Driver
2. Using Oracle JDeveloper 14c to Build the Application
3. Deploying the Application
4. Testing the Application

Configuring the Email Driver

To enable the Oracle User Messaging Service's email driver to perform inbound and outbound messaging and status retrieval, configure the email driver as follows:

- Enter the name of the SMTP mail server as the value for the **OutgoingMailServer** property.
- Enter the name of the IMAP4/POP3 mail server as the value for the **IncomingMailServer** property. Also, configure the incoming user name, and password.

For more information about configuring the Email driver, refer to section Configuring the Email Driver in *Oracle Fusion Middleware Administering Oracle User Messaging Service*.

> **Note:**
>
> This sample application is generic and can support inbound and outbound messaging through other channels when the appropriate messaging drivers are deployed and configured.

Using Oracle JDeveloper 14c to Build the Application

This section describes using a Windows-based build of JDeveloper to build, compile, and deploy **usermessagingsample-echo** through the following steps:

- Opening the Project

Opening the Project

1. Unzip `usermessagingsample-echo-src.zip`, to the *`JDEV_HOME`*`/communications/ samples/` directory. This directory must be used for the shared library references to be valid in the project.

   > **Note:**
   >
   > If you choose to use a different directory, you must update the `oracle.sdp.messaging` library source path to *`JDEV_HOME`*`/ communications/ modules/oracle.sdp.messaging_12.1.3/ sdpmessaging.jar`.

2. Open `usermessagingsample-echo.jws` (contained in the .zip file) in Oracle JDeveloper.

   In the Oracle JDeveloper main window the project appears.

3. Verify that the build dependencies for the sample application have been satisfied by checking that the following library has been added to the `usermessagingsample-echo-web` module.

   - Library: *oracle.sdp.messaging*, Classpath: *`JDEV_HOME`*`/ communications/modules/ oracle.sdp.messaging_12.1.3/ sdpmessaging.jar`. This is the Java library used by UMS and applications that use UMS to send and receive messages.

   Perform the following steps for each module:

   a. In the Application Navigator, right-click the module and select **Project Properties**.

   b. In the left pane, select **Libraries and Classpath**.

   c. Click **OK**.

4. Explore the Java files under the **usermessagingsample-echo-web** project to see how the messaging client APIs are used to register and unregister access s, and how the `EchoListener` is used to asynchronously receive messages.

Deploying the Application

Perform the following steps to deploy the application:

1. Create an Application Server Connection by right-clicking the application in the navigation pane and selecting New. Follow the instructions in Creating a New Application Server Connection.
2. Deploy the application by selecting the **usermessagingsample-echo application**, **Deploy**, **usermessagingsample-echo**, **to**, and **ums_server**.
3. Verify that the message `Build Successful` appears in the log.
4. Verify that the message `Deployment Finished` appears in the deployment log.

   You have successfully deployed the application.

Before you can run the sample you must configure any additional drivers in Oracle User Messaging Service and optionally configure a default device for the user receiving the message in User Communication Preferences.

> **Note:**
>
> Refer to *Developing Applications with Oracle User Messaging Service* for more information.

Testing the Application

Once **usermessagingsample-echo** has been deployed to a running instance of Oracle WebLogic Server, perform the following:

1. Launch a web browser and enter the address of the sample application as follows: `http://host:http-port/usermessagingsample-echo/`. For example, enter `http://localhost:7001/usermessagingsample-echo/` into the browser's navigation bar.

   When prompted, enter login credentials. For example, username `weblogic`. The browser page for testing messaging samples appears.
2. Click **Register/Unregister Access Points**. The **Access Point Registration** page appears.
3. Enter the access address in the following format:

   `EMAIL:server_address`.

   For example, enter `EMAIL:myserver@example.com`.
4. Select the Action **Register** and Click **Submit**. The registration status page appears, showing **Registered**.
5. Send a message from your messaging client (for email, your email client) to the address you just registered as an access in the previous step.

   If the UMS messaging driver for that channel is configured correctly, you should expect to receive an echo message back from the **usermessagingsample-echo** application.

Creating a New Application Server Connection

You define an application server connection in Oracle JDeveloper, and deploy and run the application.

Perform the following steps to create an Application Server Connection.

1. Right-click the project and select **New**. From the context menu, select **From Gallery**. In the New Gallery window, navigate to **Connections** in the left pane, and select **Application Server Connection** from list of items.

   Click **OK**.
2. In the **Connection Name** field, enter your server connection name, for example, *SOA_server*, and click **Next**.
3. Select **WebLogic 12.x** from the Connection Type drop-down list.
4. In the Authentication screen, enter your application server's admin **Username** and **Password**. Click **Next**.

5. In the Configuration screen, enter the WebLogic Server host name, port, and SSL port, and domain name. Click **Next**.
6. On the Test screen, verify your connection by clicking **Test Connection**. If the test is successful, then you will see a confirmation message. Click **Finish**.

   The Application Server Connection has been created.

Sample Chat Application with Web Services APIs

This section describes how to create, deploy and run the sample chat application with the Web Services APIs provided with Oracle User Messaging Service on OTN.

> **Note:**
>
> To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, see the samples at:
>
> `http://www.oracle.com/technetwork/indexes/samplecode/sample-ums-1454424.html`.

Overview

This sample demonstrates how to create a web-based chat application to send and receive messages through email, SMS, or IM. The sample uses the Web Service APIs to interact with a User Messaging server. You define an application server connection in Oracle JDeveloper, and deploy and run the application.

The application is provided as a pre-built Oracle JDeveloper project that includes a simple web chat interface.

> **Note:**
>
> For this sample to work, a UMS Server must be available and properly configured with the required drivers.

Provided Files

The following files are included in the sample application:

- `usermessagingsample-ws-src.zip` – the archive containing the source code and Oracle JDeveloper project files.

- `usermessagingsample-ws.ear` - the pre-built sample application that can be deployed to the container.

Running the Pre-Built Sample

Perform the following steps to run and deploy the pre-built sample application:

1. Extract `usermessagingsample-ws-src.zip` and open **usermessagingsample-ws.jws** in Oracle JDeveloper.

   In the Oracle JDeveloper main window the project appears.

   The application contains one web module. All of the source code for the application is in place.

2. Satisfy the build dependencies for the sample application by ensuring the `Oracle UMS Client` library is used by the web module.

   a. In the Application Navigator, right-click web module `usermessagingsample-ws-war`, and select **Project Properties**.

   b. In the left pane, select **Libraries and Classpath**.

   c. Click **OK**.

3. Create an Application Server Connection by right-clicking the project in the navigation pane and selecting **New**. Follow the instructions in Creating a New Application Server Connection.

4. Deploy the project by selecting the **usermessasgingsample-ws project**, **Deploy**, **usermessasgingsample-ws**, **to**, and **ums_server**.

5. Verify that the message `Build Successful` appears in the log.

6. Verify that the message `Deployment Finished` appears in the deployment log.

   You have successfully deployed the application.

Testing the Sample

Perform the following steps to run and test the sample:

1. Open a web browser.

2. Navigate to the URL of the application as follows, and log in:

   `http://`*`host`*`:`*`port`*`/usermessagingsample-ws/`

   The Messaging Web Services sample web page appears. This page contains navigation tabs and instructions for the application.

3. Click **Configure** and enter the following values:

   - Specify the web service endpoint. For example, `http://example.com:8001/ucs/messaging/webservice`
   - Specify the Username and Password.
   - Specify a Policy (required if the User Messaging Service instance has WS security enabled).

4. Click **Save**.

5. Click **Manage**.

6. Enter an address and optional keyword at which to receive messages.

7. Click **Start**.

   Verify that the message `Registration operation succeeded` appears.
8. Click **Chat**.
9. Enter recipients in the **To:** field.
10. Enter a message.
11. Click **Send**.
12. Verify that the message is received.

Creating a New Application Server Connection

You define an application server connection in Oracle JDeveloper, and deploy and run the application. Perform the steps in Creating a New Application Server Connection to create an Application Server Connection.