

Oracle® Fusion Middleware

Developing Applications with Oracle User Messaging Service

Release 12c (12.1.2)

E27789-02

August 2013

Oracle Fusion Middleware Developing Applications with Oracle User Messaging Service Release 12c (12.1.2)
E27789-02

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Primary Author: Savija Vijayaraghavan

Contributing Author: Swati Thacker

Contributor:

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	ix
Audience	ix
Documentation Accessibility	ix
Related Documents	ix
Conventions	ix
1.1 Introduction to User Messaging Service.....	1-1
1.2 User Messaging Service Sample Applications.....	1-1
2.1 Introduction to the UMS Java API.....	2-1
2.1.1 Creating a Java EE Application Module.....	2-2
2.2 Creating a UMS Client Instance.....	2-2
2.2.1 Creating a MessagingEJBClient Instance Using a Programmatic or Declarative Approach	2-2
2.2.2 API Reference for Class MessagingClientFactory.....	2-3
2.3 Sending a Message.....	2-3
2.3.1 Creating a Message.....	2-3
2.3.1.1 Creating a Plaintext Message.....	2-3
2.3.1.2 Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts)	2-3
2.3.1.3 Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types	2-4
2.3.2 API Reference for Class MessageFactory	2-4
2.3.3 API Reference for Interface Message	2-5
2.3.4 API Reference for Enum DeliveryType	2-5
2.3.5 Addressing a Message	2-5
2.3.5.1 Types of Addresses	2-5
2.3.5.2 Creating Address Objects.....	2-5
2.3.5.2.1 Creating a Single Address Object	2-5
2.3.5.2.2 Creating Multiple Address Objects in a Batch.....	2-5
2.3.5.2.3 Adding Sender or Recipient Addresses to a Message	2-5
2.3.5.3 Creating a Recipient with a Failover Address.....	2-5
2.3.5.4 API Reference for Class AddressFactory	2-6
2.3.5.5 API Reference for Interface Address	2-6
2.3.6 Retrieving Message Status.....	2-6
2.3.6.1 Synchronous Retrieval of Message Status	2-6
2.3.6.2 Asynchronous Notification of Message Status	2-6
2.4 Receiving a Message.....	2-6

2.4.1	Registering an Access Point	2-6
2.4.2	Synchronous Receiving.....	2-7
2.4.3	Asynchronous Receiving.....	2-7
2.4.4	Message Filtering	2-7
2.5	Using the UMS Enterprise JavaBeans Client API to Build a Client Application.....	2-8
2.5.1	Overview of Development	2-9
2.5.2	Configuring the Email Driver	2-9
2.5.3	Using JDeveloper 11g to Build the Application	2-9
2.5.3.1	Opening the Project.....	2-9
2.5.4	Deploying the Application.....	2-11
2.5.5	Testing the Application.....	2-11
2.6	Using the UMS Enterprise JavaBeans Client API to Build a Client Echo Application..	2-13
2.6.1	Overview of Development	2-14
2.6.2	Configuring the Email Driver	2-14
2.6.3	Using JDeveloper 11g to Build the Application	2-15
2.6.3.1	Opening the Project.....	2-15
2.6.4	Deploying the Application.....	2-18
2.6.5	Testing the Application.....	2-18
2.7	Creating a New Application Server Connection.....	2-20
3.1	Introduction to the UMS Java API.....	3-2
3.2	Creating a UMS Client Instance and Specifying Runtime Parameters	3-2
3.3	Sending a Message.....	3-3
3.3.1	Creating a Message.....	3-4
3.3.1.1	Creating a Plaintext Message.....	3-4
3.3.1.2	Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts) 3-4	
3.3.1.3	Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types 3-5	
3.3.2	Addressing a Message	3-6
3.3.2.1	Types of Addresses	3-6
3.3.2.2	Creating Address Objects.....	3-6
3.3.2.2.1	Creating a Single Address Object	3-6
3.3.2.2.2	Creating Multiple Address Objects in a Batch.....	3-6
3.3.2.2.3	Adding Sender or Recipient Addresses to a Message	3-6
3.3.2.3	Creating a Recipient with a Failover Address.....	3-6
3.3.2.4	API Reference for Class MessagingFactory	3-6
3.3.2.5	API Reference for Interface Address	3-6
3.3.3	Sending Group Messages	3-7
3.3.3.1	Sending Messages to a Group.....	3-7
3.3.3.2	Sending Messages to a Group Through a Specific Channel	3-7
3.3.3.3	Sending Messages to an Application Role.....	3-8
3.3.3.4	Sending Messages to an Application Role Through a Specific Channel.....	3-8
3.3.4	User Preference Based Messaging	3-9
3.4	Retrieving Message Status.....	3-9
3.4.1	Synchronous Retrieval of Message Status.....	3-9
3.4.2	Asynchronous Receiving of Message Status	3-9
3.4.2.1	Creating a Listener Programmatically.....	3-9
3.4.2.2	Default Status Listener.....	3-10

3.4.2.3	Per Message Status Listener.....	3-10
3.5	Receiving a Message.....	3-10
3.5.1	Registering an Access Point	3-11
3.5.2	Synchronous Receiving.....	3-11
3.5.3	Asynchronous Receiving.....	3-11
3.5.3.1	Creating a Listener Programmatically.....	3-12
3.5.3.2	Default Message Listener	3-12
3.5.3.3	Per Access Point Message Listener	3-12
3.5.4	Message Filtering	3-13
3.6	Configuring for a Cluster Environment	3-13
3.7	Using UMS Client API for XA Transactions	3-14
3.7.1	About XA Transactions.....	3-14
3.7.2	Sending and Receiving XA Enabled Messages	3-14
3.8	Using UMS Java API to Specify Message Resends	3-16
3.9	Configuring Security	3-17
3.10	Threading Model.....	3-17
3.10.1	Listener Threading	3-18
4.1	Introduction to the UMS Web Service API	4-1
4.2	Creating a UMS Client Instance and Specifying Runtime Parameters	4-2
4.3	Sending a Message.....	4-3
4.3.1	Creating a Message.....	4-4
4.3.1.1	Creating a Plaintext Message.....	4-4
4.3.1.2	Creating a Multipart/Mixed Message (with Text and Binary Parts).....	4-4
4.3.1.3	Creating a Multipart/Alternative Message (with Text/Plain and Text/HTML Parts) 4-4	
4.3.1.4	Creating Delivery Channel-Specific Payloads in a Single Message for Recipients with Different Delivery Types 4-5	
4.3.2	API Reference for Interface Message	4-6
4.3.3	API Reference for Enum DeliveryType.....	4-6
4.3.4	Addressing a Message	4-6
4.3.4.1	Types of Addresses	4-6
4.3.4.2	Creating Address Objects.....	4-6
4.3.4.2.1	Creating a Single Address Object	4-6
4.3.4.2.2	Creating Multiple Address Objects in a Batch.....	4-6
4.3.4.2.3	Adding Sender or Recipient Addresses to a Message	4-6
4.3.4.3	Creating a Recipient with a Failover Address.....	4-7
4.3.4.4	Recipient Types.....	4-7
4.3.4.5	API Reference for Class MessagingFactory	4-7
4.3.4.6	API Reference for Interface Address	4-7
4.3.5	User Preference Based Messaging	4-7
4.4	Retrieving Message Status.....	4-8
4.4.1	Synchronous Retrieval of Message Status.....	4-8
4.4.2	Asynchronous Receiving of Message Status	4-8
4.4.2.1	Creating a Listener Programmatically.....	4-8
4.4.2.2	Publish the Callback Service	4-9
4.4.2.3	Stop a Dynamically Published Endpoint	4-9
4.4.2.4	Registration.....	4-9

4.5	Receiving a Message.....	4-9
4.5.1	Registering an Access Point	4-10
4.5.2	Synchronous Receiving.....	4-10
4.5.3	Asynchronous Receiving.....	4-10
4.5.3.1	Creating a Listener Programmatically.....	4-11
4.5.3.2	Default Message Listener	4-11
4.5.3.3	Per Access Point Message Listener	4-12
4.5.4	Message Filtering.....	4-12
4.6	Configuring for a Cluster Environment	4-12
4.7	Using UMS Web Service API to Specify Message Resends.....	4-13
4.8	Configuring Security	4-13
4.8.1	Client and Server Security	4-13
4.8.2	Listener or Callback Security	4-14
4.9	Threading Model.....	4-14
4.10	Sample Chat Application with Web Services APIs.....	4-14
4.10.1	Overview.....	4-15
4.10.1.1	Provided Files	4-15
4.10.2	Running the Pre-Built Sample	4-15
4.10.3	Testing the Sample.....	4-17
4.10.4	Creating a New Application Server Connection.....	4-20
5.1	Introduction to Parlay X Messaging Operations.....	5-1
5.2	Send Message Interface.....	5-2
5.2.1	sendMessage Operation.....	5-2
5.2.2	getMessageDeliveryStatus Operation	5-3
5.3	Receive Message Interface	5-4
5.3.1	getReceivedMessages Operation.....	5-4
5.3.2	getMessage Operation.....	5-5
5.3.3	getMessageURIs Operation.....	5-5
5.4	Oracle Extension to Parlay X Messaging.....	5-6
5.4.1	ReceiveMessageManager Interface	5-6
5.4.1.1	startReceiveMessage Operation	5-6
5.4.1.2	stopReceiveMessage Operation.....	5-7
5.5	Parlay X Messaging Client API and Client Proxy Packages.....	5-7
5.6	Sample Chat Application with Parlay X APIs	5-8
5.6.1	Overview	5-8
5.6.1.1	Provided Files	5-9
5.6.2	Running the Pre-Built Sample	5-9
5.6.3	Testing the Sample.....	5-11
5.6.4	Creating a New Application Server Connection.....	5-13
6.1	Introduction to User Communication Preferences	6-1
6.1.1	Terminology	6-2
6.2	Managing User Preferences.....	6-2
6.2.1	Managing Communication Channels.....	6-2
6.2.1.1	Creating a Channel.....	6-3
6.2.1.2	Modifying a Channel	6-3
6.2.1.3	Deleting a Channel.....	6-4
6.2.1.4	Setting a Default Channel.....	6-4

6.2.2	Managing Filters	6-5
6.2.2.1	Creating a Filter	6-6
6.2.2.2	Modifying a Filter	6-7
6.2.2.3	Deleting a Filter.....	6-7
6.2.2.4	Disabling a Filter.....	6-7
6.2.2.5	Organizing Filters.....	6-8
6.2.3	Configuring Preference Settings.....	6-8
6.3	Administering User Communication Preferences	6-9
6.3.1	About Business Terms.....	6-9
6.3.2	Configuring Profiles by using Oracle Enterprise Manager	6-10
6.3.3	Managing User Data using WLST Commands	6-13
6.4	Integrating UCP Web User Interface	6-14
6.4.1	Integrate ADF Web Application with UCP	6-14
6.4.1.1	Create a New ADF Application	6-14
6.4.1.2	Create an ADF Web Page	6-15
6.4.1.3	Connect UCP Task Flow Library.....	6-17
6.4.1.4	Add a Region in the New Page	6-19
6.4.1.5	Reference UCP Libraries.....	6-20
6.4.1.6	Manage Project Deployment Profile.....	6-22
6.4.1.7	Create Application Deployment Profile.....	6-24
6.4.2	Deploy Your Application	6-25
6.4.2.1	Deploy Application	6-25
6.4.2.2	Configure Application Server Connection	6-25
6.4.3	Verify Your Application	6-27
6.5	Java Application Interface	6-28
6.5.1	Obtain Delivery Preferences	6-29
6.5.2	Manage Channels	6-29
6.5.3	Manage Filters.....	6-30

A Using the User Messaging Service Sample Applications

A.1	Using the UMS Client API to Build a Client Application	A-1
A.1.1	Overview of Development	A-2
A.1.2	Configuring the Email Driver	A-2
A.1.3	Using JDeveloper 12c to Build the Application.....	A-2
A.1.3.1	Opening the Project.....	A-2
A.1.4	Deploying the Application	A-3
A.1.5	Testing the Application.....	A-4
A.2	Using the UMS Client API to Build a Client Echo Application	A-6
A.2.1	Overview of Development	A-7
A.2.2	Configuring the Email Driver	A-7
A.2.3	Using Oracle JDeveloper 12c to Build the Application	A-8
A.2.3.1	Opening the Project.....	A-8
A.2.4	Deploying the Application	A-10
A.2.5	Testing the Application.....	A-11
A.3	Creating a New Application Server Connection.....	A-13

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 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.

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.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

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

- [Section 1.1, "Introduction to User Messaging Service"](#)
- [Section 1.2, "User Messaging Service Sample Applications"](#)

1.1 Introduction to User Messaging Service

Oracle User Messaging Service provides a common service responsible for sending out messages from applications to devices. It also routes incoming messages from devices to applications.

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

1.2 User Messaging Service Sample Applications

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

<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.

This document describes how to develop applications using User Messaging Service Java API (see [Chapter 3, "Sending and Receiving Messages using the User Messaging Service Java API"](#)) and User Messaging Service Web Services API (see [Chapter 4, "Sending and Receiving Messages using the User Messaging Service Web Service API"](#)). You can build and deploy these sample applications using Oracle JDeveloper 12c, and also manage communication preferences through a web interface (see [Chapter 6, "User Communication Preferences"](#)).

Sending and Receiving Messages using the User Messaging Service EJB API

This chapter describes how to use the User Messaging Service (UMS) EJB API to develop applications, and describes how to build two sample applications, `usermessagingsample-ejb.ear` and `usermessagingsample-echo-ejb.ear`. This chapter includes the following sections:

Note: The User Messaging Service EJB API (described in this chapter) is deprecated. Use the User Messaging Service Java API instead, as described in [Chapter 3, "Sending and Receiving Messages using the User Messaging Service Java API"](#).

- [Section 2.1, "Introduction to the UMS Java API"](#)
- [Section 2.2, "Creating a UMS Client Instance"](#)
- [Section 2.3, "Sending a Message"](#)
- [Section 2.4, "Receiving a Message"](#)
- [Section 2.5, "Using the UMS Enterprise JavaBeans Client API to Build a Client Application"](#)
- [Section 2.6, "Using the UMS Enterprise JavaBeans Client API to Build a Client Echo Application"](#)
- [Section 2.7, "Creating a New Application Server Connection"](#)

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>

2.1 Introduction to the UMS Java API

The UMS Java API supports developing applications for Enterprise JavaBeans clients. It consists of packages grouped as follows:

- Common and Client Packages
 - `oracle.sdp.client`

- `oracle.sdp.client.filter`: A `MessageFilter` is used by an application to exercise greater control over what messages are delivered to it.
- User Preferences Packages
 - `oracle.sdp.client.userprefs`
 - `oracle.sdp.client.userprefs.tools`

2.1.1 Creating a Java EE Application Module

There are two choices for a Java EE application module that uses the UMS Enterprise JavaBeans Client API:

- Enterprise JavaBeans Application Module - Stateless Session Bean - This is a back end, core message-receiving or message-sending application.
- Web Application Module - This is for applications that have an HTML or web front end.

Whichever application module is selected uses the UMS Client API to register the application with the UMS Server and subsequently invoke operations to send or retrieve messages, status, and register or unregister access points. For a complete list of operations refer to the UMS Javadoc.

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

2.2 Creating a UMS Client Instance

This section describes the requirements for creating a UMS Enterprise JavaBeans Client. You can create a `MessagingEJBClient` instance by using the code in the `MessagingClientFactory` class.

When creating an application using the UMS Enterprise JavaBeans Client, the application must be packaged as an EAR file, and the `usermessagingclient-ejb.jar` module bundled as an Enterprise JavaBeans module.

2.2.1 Creating a MessagingEJBClient Instance Using a Programmatic or Declarative Approach

[Example 2-1](#) shows code for creating a `MessagingEJBClient` instance using the programmatic approach:

Example 2-1 Programmatic Approach to Creating a MessagingEJBClient Instance

```
ApplicationInfo appInfo = new ApplicationInfo();
appInfo.setApplicationName("SampleApp");
appInfo.setApplicationInstanceName("SampleAppInstance");
MessagingClient mClient =
    MessagingClientFactory.createMessagingEJBClient(appInfo);
```

You can also create a `MessagingEJBClient` instance using a declarative approach. The declarative approach is normally the preferred approach since it enables you to make changes at deployment time.

You must specify all the required Application Info properties as environment entries in your Java EE module's descriptor (`ejb-jar.xml` or `web.xml`).

[Example 2-2](#) shows code for creating a `MessagingEJBClient` instance using the declarative approach:

Example 2-2 Declarative Approach to Creating a MessagingEJBClient Instance

```
MessagingClient mClient = MessagingClientFactory.createMessagingEJBClient();
```

2.2.2 API Reference for Class MessagingClientFactory

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

2.3 Sending a Message

You can create a message by using the code in the `MessageFactory` class and `Message` interface of `oracle.sdp.client`.

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.

2.3.1 Creating a Message

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

2.3.1.1 Creating a Plaintext Message

[Example 2-3](#) shows how to create a plain text message using the UMS Java API.

Example 2-3 Creating a Plaintext Message Using the UMS Java API

```
Message message = MessageFactory.getInstance().createTextMessage("This is a Plain Text message.");
Message message = MessageFactory.getInstance().createMessage();
message.setContent("This is a Plain Text message.", "text/plain");
```

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

[Example 2-4](#) shows how to create a multipart or alternative message using the UMS Java API.

Example 2-4 Creating a Multipart or Alternative Message Using the UMS Java API

```
Message message = MessageFactory.getInstance().createMessage();
MimeMultipart mp = new MimeMultipart("alternative");
MimeBodyPart mp_partPlain = new MimeBodyPart();
mp_partPlain.setContent("This is a Plain Text part.", "text/plain");
mp.addBodyPart(mp_partPlain);
MimeBodyPart mp_partRich = new MimeBodyPart();
mp_partRich
    .setContent(
        "<html><head></head><body><b><i>This is an HTML
part.</i></b></body></html>",
        "text/html");
mp.addBodyPart(mp_partRich);
message.setContent(mp, "multipart/alternative");
```

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

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

[Example 2-5](#) 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 2-5 Creating Delivery Channel-specific Payloads in a Single Message for Recipients with Different Delivery Types

```
Message message = MessageFactory.getInstance().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);
```

2.3.2 API Reference for Class MessageFactory

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

2.3.3 API Reference for Interface Message

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

2.3.4 API Reference for Enum DeliveryType

The API reference for enum `DeliveryType` can be accessed from the Javadoc.

2.3.5 Addressing a Message

This section describes type of addresses and how to create address objects.

2.3.5.1 Types of Addresses

There are two types of addresses, *device addresses* and *user addresses*. A device address can be of various types, such as email addresses, instant messaging addresses, and telephone numbers. User addresses are user IDs in a user repository.

2.3.5.2 Creating Address Objects

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

2.3.5.2.1 Creating a Single Address Object [Example 2-6](#) shows code for creating a single `Address` object:

Example 2-6 Creating a Single Address Object

```
Address recipient =
AddressFactory.getInstance().createAddress("Email:john@example.com");
```

2.3.5.2.2 Creating Multiple Address Objects in a Batch [Example 2-7](#) shows code for creating multiple `Address` objects in a batch:

Example 2-7 Creating Multiple Address Objects in a Batch

```
String[] recipientsStr = {"Email:john@example.com", "IM:jabber|john@example.com"};
Address[] recipients = AddressFactory.getInstance().createAddress(recipientsStr);
```

2.3.5.2.3 Adding Sender or Recipient Addresses to a Message [Example 2-8](#) shows code for adding sender or recipient addresses to a message:

Example 2-8 Adding Sender or Recipient Addresses to a Message

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

2.3.5.3 Creating a Recipient with a Failover Address

[Example 2-9](#) shows code for creating a recipient with a failover address:

Example 2-9 Creating a Single Address Object with Failover

```
String recipientWithFailoverStr = "Email:john@example.com,
IM:jabber|john@example.com";
Address recipient =
```

```
AddressFactory.getInstance().createAddress(recipientWithFailoverStr);
```

2.3.5.4 API Reference for Class AddressFactory

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

2.3.5.5 API Reference for Interface Address

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

2.3.6 Retrieving Message Status

You can use Oracle UMS to retrieve message status either synchronously or asynchronously.

2.3.6.1 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.
```

2.3.6.2 Asynchronous Notification of Message Status

To retrieve an asynchronous notification of message status, perform the following:

1. Implement a status listener.
2. Register a status listener (declarative way)
3. Send a message (`messagingClient.send(message);`)
4. The application automatically gets the status through an `onStatus(status)` callback of the status listener.

2.4 Receiving a Message

This section describes how an application receives messages. To receive a message you must first register an access point. From the application perspective there are two modes for receiving a message, synchronous and asynchronous.

2.4.1 Registering an Access Point

`AccessPoint` represents one or more device addresses to receive incoming 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.

You can use `AccessPointFactory.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 =
    AccessPointFactory.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 =
    AccessPointFactory.createAccessPoint(AccessPoint.AccessPointType.NUMBER_RANGE,
        DeliveryType.SMS, "9000,9999");
messagingClient.registerAccessPoint(accessPointRangeAddress);
```

2.4.2 Synchronous Receiving

You can use the method `MessagingClient.receive` to synchronously receive messages. This is a convenient polling method for light-weight clients that do not want the configuration overhead associated with receiving messages asynchronously. This method returns a list of messages that are immediately available in the application inbound queue.

It performs a nonblocking call, so if no message is currently available, the method returns null.

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

2.4.3 Asynchronous Receiving

Asynchronous receiving involves many tasks, including configuring MDBs and writing a Stateless Session Bean message listener. See the sample application `usermessagingsample-echo` for detailed instructions.

2.4.4 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.

You can use `MessageFilterFactory.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. When a message is received, it is passed through the filter chain in order; if the message matches a filter's criterion, the filter's action is taken immediately. If no filters match the message, the default action is to accept the message and deliver it to the application.

For example, to reject a message with the subject "spam":

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

```
messagingClient.registerMessageFilter(subjectFilter);
```

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

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

2.5 Using the UMS Enterprise JavaBeans 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 Enterprise JavaBeans 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.

Note: To learn more about the code samples for Oracle User Messaging Service, or to run the samples yourself, see the Oracle SOA Suite samples.

Once you have navigated to this page, you can find code samples for Oracle User Messaging Service by entering the search term "UMS" and clicking **Search**.

Of the two application modules choices described in [Section 2.1.1, "Creating a Java EE Application Module,"](#) this sample focuses on the 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 Enterprise JavaBeans Client API to create an UMS Enterprise JavaBeans Client instance (which in turn registers the application's info) and sends messages.

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

- `usermessagingsample-ejb.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.client` shared library.
 - `usermessagingclient-ejb.jar` -- Contains the Message Enterprise JavaBeans Client deployment descriptors.
 - * `META-INF`
 - `ejb-jar.xml`
 - `weblogic-ejb-jar.xml`
 - `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.

2.5.1 Overview of Development

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

1. [Section 2.5.2, "Configuring the Email Driver"](#)
2. [Section 2.5.3, "Using JDeveloper 11g to Build the Application"](#)
3. [Section 2.5.4, "Deploying the Application"](#)
4. [Section 2.5.5, "Testing the Application"](#)

2.5.2 Configuring the Email Driver

To enable the Oracle User Messaging Service's email driver to perform 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.

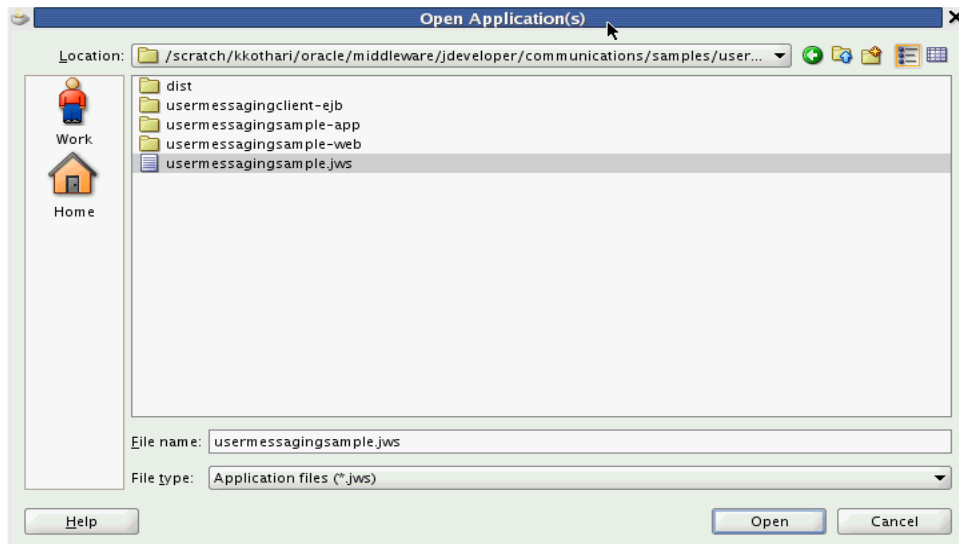
Note: This sample application is generic and can support outbound messaging through other channels when the appropriate messaging drivers are deployed and configured.

2.5.3 Using JDeveloper 11g to Build the Application

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

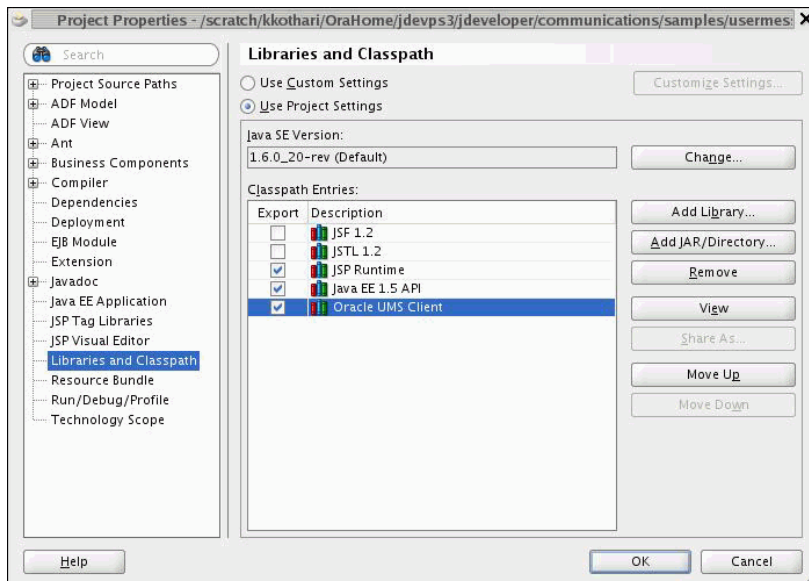
2.5.3.1 Opening the Project

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

Figure 2–1 Oracle JDeveloper Main Window

In the Oracle JDeveloper main window, the project appears.

2. Satisfy the build dependencies for the sample application by ensuring the "Oracle UMS Client" library is used by the web module.
 1. In the Application Navigator, right-click web module **usermessagingsample-web**, and select **Project Properties**.
 2. In the left pane, select **Libraries and Classpath**.

Figure 2–2 Verifying Libraries

3. Click **OK**.
3. Verify that the **usermessagingclient-ejb** project exists in the application. This is an Enterprise JavaBeans module that packages the messaging client beans used by UMS applications. The module allows the application to connect with the UMS server.

4. 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 application info that is registered with the UMS Server is specified programmatically in `SampleUtils.java` in the project ([Example 2-10](#)).

Example 2-10 Application Information

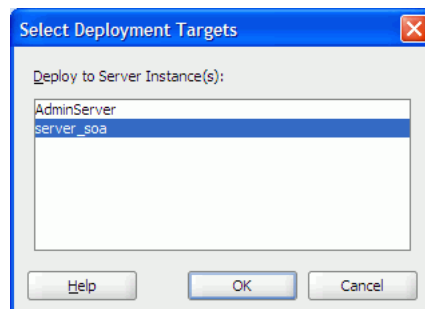
```
ApplicationInfo appInfo = new ApplicationInfo();
appInfo.setApplicationName(SampleConstants.APP_NAME);
appInfo.setApplicationInstanceName(SampleConstants.APP_INSTANCE_NAME);
appInfo.setSecurityPrincipal(request.getUserPrincipal().getName());
```

2.5.4 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 [Section 2.7](#), "Creating a New Application Server Connection."
2. Deploy the application by selecting the **usermessagingsample** application, **Deploy**, **usermessagingsample**, **to**, and **SOA_server** ([Figure 2-3](#)).

Figure 2-3 Deploying the Project



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.

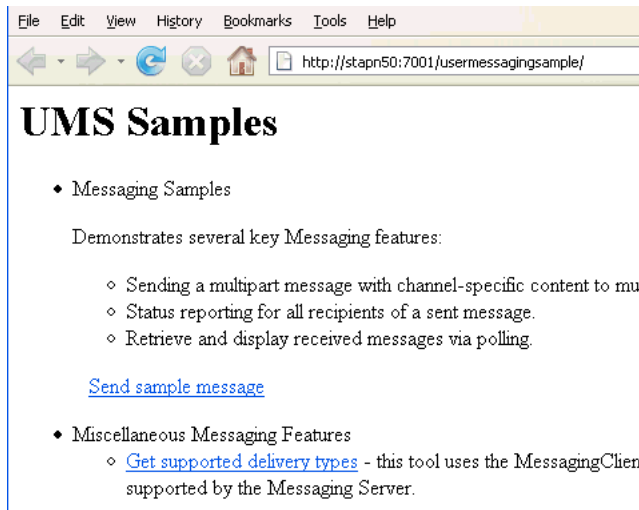
2.5.5 Testing the Application

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.

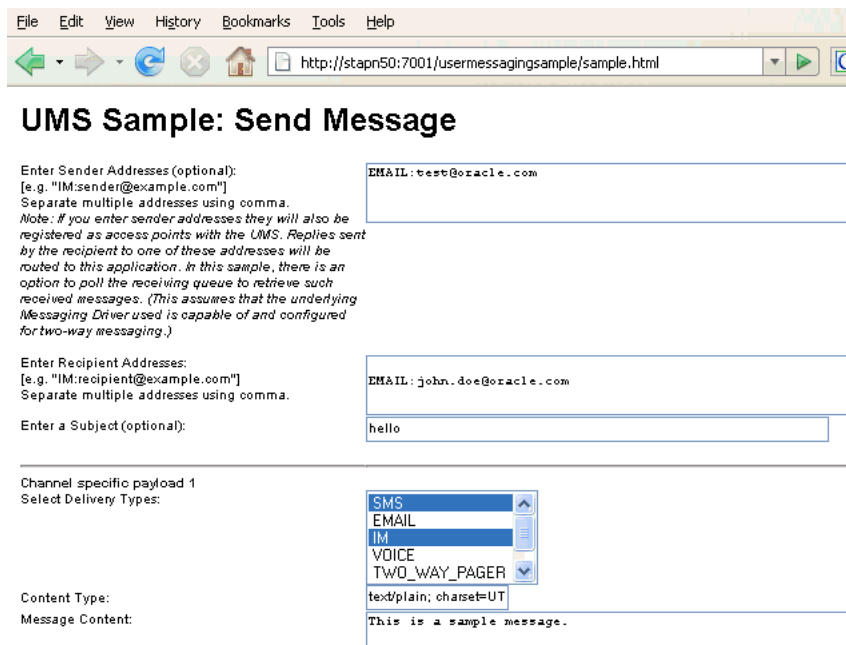
When prompted, enter login credentials. For example, username `weblogic`. The browser page for testing messaging samples appears (Figure 2-4).

Figure 2-4 Testing the Sample Application



- Click **Send sample message**. The Send Message page appears (Figure 2-5).

Figure 2-5 Addressing the Test Message



- As an optional step, enter the sender address in the following format:
Email: `sender_address`.
For example, enter `Email:sender@example.com`.
- 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: *testuser1*.

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 (Message received by Messaging engine for processing in [Figure 2-6](#)).


Figure 2-6 Message Status

UMS Sample to Send Messages

Sending message:

Sent message with id = f622d4dd984464dd008fbc395a2e5afa

Checking Status: [Refresh](#)

UMS: Message Status						
Status for message id: f622d4dd984464dd008fbc395a2e5afa						
Gateway Message ID	Address	Status Type	Status Content	Reporting Driver	Date	Failover Status
Recipient #1: EMAIL:john.doe@oracle.com ...						
	EMAIL:john.doe@oracle.com		Message received by Messaging engine for processing.		Jan 20, 2008 2:22:04 PM PST	false

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*.

2.6 Using the UMS Enterprise JavaBeans 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 Enterprise JavaBeans 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, see the Oracle SOA Suite samples.

Once you have navigated to this page, you can find code samples for Oracle User Messaging Service by entering the search term "UMS" and clicking **Search**.

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

- `usermessagingsample-echo-ejb.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.client` shared library.

- `usermessagingclient-ejb.jar` -- Contains the Message Enterprise JavaBeans Client deployment descriptors.
 - * `META-INF`
 - `ejb-jar.xml`
 - `weblogic-ejb-jar.xml`
- `usermessagingssample-echo-ejb.jar` -- Contains the application session beans (`ClientSenderBean`, `ClientReceiverBean`) that process a received message and return an echo response.
 - * `META-INF`
 - `ejb-jar.xml`
 - `weblogic-ejb-jar.xml`
- `usermessagingssample-echo-web.war` -- Contains the web-based front-end and servlets.
 - * `WEB-INF`
 - `web.xml`
 - `weblogic.xml`

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

2.6.1 Overview of Development

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

1. [Section 2.6.2, "Configuring the Email Driver"](#)
2. [Section 2.6.3, "Using JDeveloper 11g to Build the Application"](#)
3. [Section 2.6.4, "Deploying the Application"](#)
4. [Section 2.6.5, "Testing the Application"](#)

2.6.2 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.

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.

2.6.3 Using JDeveloper 11g 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:

2.6.3.1 Opening the Project

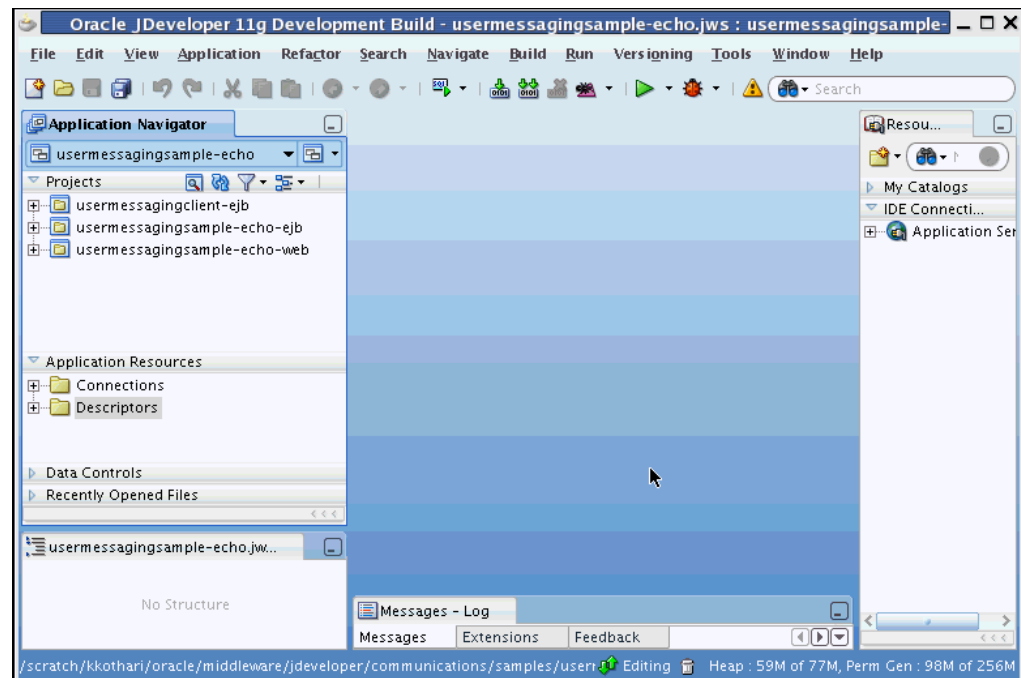
1. Unzip `usermessagingsample-echo-src.zip`, to the `JDEV_HOME/communications/samples/` directory. This directory must be used for the shared library references to be valid in the project.

Note: If you choose to use a different directory, you must update the `oracle.sdp.client` library source path to `JDEV_HOME/communications/modules/oracle.sdp.client_12.1.2/sdpclient.jar`.

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

In the Oracle JDeveloper main window, the project appears (Figure 2-7).

Figure 2-7 Oracle JDeveloper Main Window

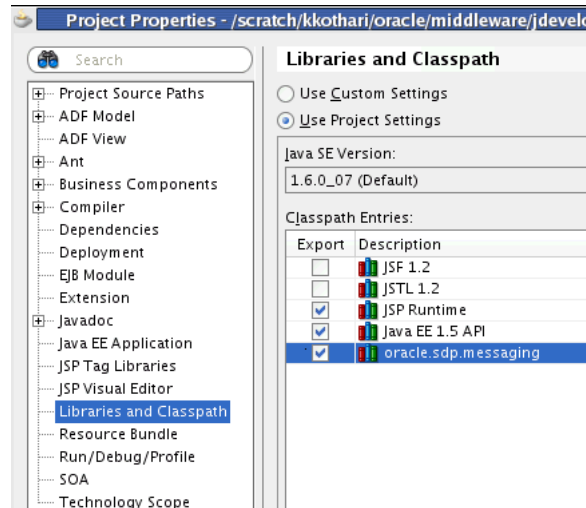


3. Verify that the build dependencies for the sample application have been satisfied by checking that the following library has been added to the `usermessagingsample-echo-web` and `usermessagingsample-echo-ejb` modules.
 - Library: `oracle.sdp.client`, Classpath: `JDEV_HOME/communications/modules/oracle.sdp.client_12.1.2/sdpclient.jar`. 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:

1. In the Application Navigator, right-click the module and select **Project Properties**.
2. In the left pane, select **Libraries and Classpath** (Figure 2–8).

Figure 2–8 Verifying Libraries



3. Click **OK**.
4. Verify that the **usermessagingclient-ejb** project exists in the application. This is an Enterprise JavaBeans module that packages the messaging client beans used by UMS applications. The module allows the application to connect with the UMS server.
5. Explore the Java files under the **usermessagingssample-echo-ejb** project to see how the messaging client APIs are used to asynchronously receive messages (`ClientReceiverBean`), and send messages (`ClientSenderBean`).
6. Explore the Java files under the **usermessagingssample-echo-web** project to see how the messaging client APIs are used to register and unregister access points.
7. Note that the application info that is registered with the UMS Server is specified declaratively in the **usermessagingclient-ejb** project's `ejb-jar.xml` file. (Example 2–11).

Example 2–11 Application Information

```

<env-entry>
  <env-entry-name>sdpm/ApplicationName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>UMSEchoApp</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>sdpm/ApplicationInstanceName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>UMSEchoAppInstance</env-entry-value>
</env-entry>

<env-entry>
  <env-entry-name>sdpm/ReceivingQueuesInfo</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>

<env-entry-value>OraSDPM/QueueConnectionFactory:OraSDPM/Queues/OraSDPMAppDefRcvQ1<

```

```

/env-entry-value>
</env-entry>

<env-entry>
  <env-entry-name>
    sdpm/MessageListenerSessionBeanJNDIName
  </env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>
    ejb/umsEchoApp/ClientReceiverLocal</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>
    sdpm/MessageListenerSessionBeanHomeClassName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>
    oracle.sdp.client.sample.ejbApp.ClientReceiverHomeLocal
  </env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>
    sdpm/StatusListenerSessionBeanJNDIName
  </env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>

<env-entry-value>ejb/umsEchoApp/ClientReceiverLocal</env-entry-value>
</env-entry>
<env-entry>

<env-entry-name>sdpm/StatusListenerSessionBeanHomeClassName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>

<env-entry-value>oracle.sdp.client.sample.ejbApp.ClientReceiverHomeLocal</env-
entry-value>
</env-entry>

```

8. Note that the Application Name (UMSEchoApp) and Application Instance Name (UMSEchoAppInstance) are also used in the Message Selector for the MessageDispatcherBean MDB, which is used for asynchronous receiving of messages and statuses placed in the application receiving queue ([Example 2-12](#)).

Example 2-12 Application Information

```

<activation-config-property>
  <activation-config-property-name>
    messageSelector
  </activation-config-property-name>
  <activation-config-property-value>
    appName='UMSEchoApp' or sessionName='UMSEchoApp-UMSEchoAppInstance'
  </activation-config-property-value>
</activation-config-property>

```

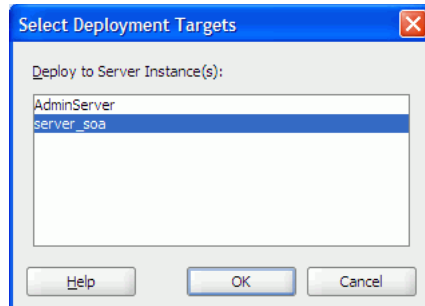
Note: If you chose a different Application Name and Application Instance Name for your own application, remember to update this message selector. Asynchronous receiving does not work, otherwise.

2.6.4 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 [Section 2.7, "Creating a New Application Server Connection."](#)
2. Deploy the application by selecting the **usermessagingsample-echo** application, **Deploy**, **usermessagingsample-echo**, **to**, and **SOA_server** ([Figure 2–9](#)).

Figure 2–9 Deploying the Project



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.

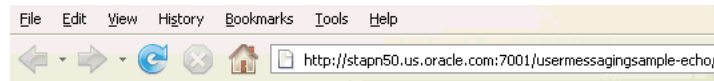
Note: Refer to *Administering Oracle User Messaging Service* for more information.

2.6.5 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 ([Figure 2–10](#)).

Figure 2–10 Testing the Sample Application

UMS Samples

- ◆ Sample for Two Way Messaging

Perform the following steps:

1. Click on "Register/unregister Access Points".
2. Enter address of access point.
For example, IM.myserver@example.com.
3. Click on Submit.
4. Using your client send a message to the access point.
5. The sample application will receive the message and echo it back to you.

[Register/Unregister Access Points](#)

2. Click **Register/Unregister Access Points**. The *Access Point Registration* page appears (Figure 2–11).

Figure 2–11 Registering an Access Point

Two Way Messaging Test requires the following steps:

1. Click on "Register/Unregister Access Points".
2. Enter address of access point, and optionally a keyword.
For example, IM.myserver@example.com.
For Demo - IM:<ServerJabberID>
3. Click on Submit.
4. Using your IM client send a message to your buddy (ServerJabberID). If a keyword was specified, the first token of the message must match the keyword.
5. The sample application will receive the message and echo it back to you.

Use this form to register/unregister access point addresses for this Sample App.

Enter an Address:
[e.g. "IM:sender@example.com"
or "EMAIL:sender@example.com"]

Enter a keyword (optional):

Action: Register
 Unregister

3. Enter the access point 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" in Figure 2–12).

Figure 2–12 Access Point Registration Status



5. Send a message from your messaging client (for email, your email client) to the address you just registered as an access point 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.

2.7 Creating a New Application Server Connection

Perform the steps in [Section A.3, "Creating a New Application Server Connection"](#) to create an Application Server Connection.

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*.

This chapter includes the following sections:

- [Section 3.1, "Introduction to the UMS Java API"](#)
- [Section 3.2, "Creating a UMS Client Instance and Specifying Runtime Parameters"](#)
- [Section 3.3, "Sending a Message"](#)
- [Section 3.4, "Retrieving Message Status"](#)
- [Section 3.5, "Receiving a Message"](#)
- [Section 3.6, "Configuring for a Cluster Environment"](#)
- [Section 3.7, "Using UMS Client API for XA Transactions"](#)
- [Section 3.8, "Using UMS Java API to Specify Message Resends"](#)
- [Section 3.9, "Configuring Security"](#)
- [Section 3.10, "Threading Model"](#)

The API provides a plain old java (POJO/POJI) programming model and this eliminates the needs for application developers to package and implement various Java EE modules (such as an EJB module) in an application to access UMS features. This reduces application development time because developers can create applications to run in a Java EE container without performing any additional packaging of modules, or obtaining specialized tools to perform such packaging tasks.

Consumers of the UMS Java API are not required to use any Java EE mechanism such as environment entries or other Java EE deployment descriptor artifacts. Besides the overhead involved in maintaining Java EE descriptors, many client applications already have a configuration framework that does not rely on Java EE descriptors.

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>

3.1 Introduction to the UMS Java API

The UMS Java API is exposed as a POJO/POJI API. Consumers of the API can get an instance of a `MessagingClient` object using a factory method. The consumers do not need to deploy any EJB or other Java 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 UMS Java API consists of packages grouped as follows:

- Common and Client Packages
 - `oracle.sdp.messaging`
 - `oracle.sdp.messaging.filter`: A `MessagingFilter` is used by an application to exercise greater control over what messages are delivered to it.

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

3.2 Creating a UMS Client Instance and Specifying Runtime Parameters

This section describes the requirements for creating a UMS Client. You can create a `MessagingClient` instance by using the code in the `MessagingClientFactory` class. Specifically, use the `MessagingClientFactory.createMessagingClient()` method to create the instance.

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 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.

[Table 3–1](#) 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 3–1 Configuration Parameters Specified at Runtime

Parameter	Notes
<code>APPLICATION_NAME</code>	Optional. By default, the client is identified by its deployment name. This identifier can be overridden by specifying a value for key <code>ApplicationInfo.APPLICATION_NAME</code> .
<code>APPLICATION_INSTANCE_NAME</code>	Optional. Only required for certain clustered use cases or to take advantage of session-based routing.

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

Parameter	Notes
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 <code>ApplicationInfo.SDPM_SECURITY_PRINCIPAL</code> . 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 <code>MessagingConstants.MESSAGE_LISTENER_THREADS</code> and <code>MessagingConstants.STATUS_LISTENER_THREADS</code> keys.
RECEIVE_ACKNOWLEDGEMENT_MODE LISTENER_ACKNOWLEDGEMENT_MODE	Optional. When receiving messages, you can control the reliability mode by specifying values for the <code>MessagingConstants.RECEIVE_ACKNOWLEDGEMENT_MODE</code> (synchronous receiving) and <code>MessagingConstants.LISTENER_ACKNOWLEDGEMENT_MODE</code> (asynchronous receiving) keys.

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 3–1](#) shows code for creating a `MessagingClient` instance using the programmatic approach:

Example 3–1 Programmatic Approach to 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.

3.3 Sending a Message

The client application can create a message object using the `MessagingFactory` class of `oracle.sdp.messaging.MessagingFactory`. You can use other methods in this class to create `Addresses`, `AccessPoints`, `MessageFilters`, and `MessageQueries`. See *User Messaging Service Java API Reference* for more information about these methods.

The client application can then send the message. The 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 messages that include the creation of delivery channel (`DeliveryType`) specific payloads in a single message for recipients with different delivery types.

The section includes the following topics:

- [Creating a Message](#)
- [Addressing a Message](#)
- [Sending Group Messages](#)
- [User Preference Based Messaging](#)

3.3.1 Creating a Message

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

3.3.1.1 Creating a Plaintext Message

[Example 3–2](#) shows how to create a plaintext message using the UMS Java API.

Example 3–2 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");
```

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

[Example 3–3](#) shows how to create a multipart or alternative message using the UMS Java API.

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

```
Message message = MessagingFactory.createMessage();  
MimeMultipart mp = new MimeMultipart("alternative");  
MimeBodyPart mp_partPlain = new MimeBodyPart();  
mp_partPlain.setContent("This is a Plain Text part.", "text/plain");  
mp.addBodyPart(mp_partPlain);  
MimeBodyPart mp_partRich = new MimeBodyPart();  
mp_partRich  
    .setContent(  
        "<html><head></head><body><b><i>This is an HTML  
part.</i></b></body></html>",  
        "text/html");  
mp.addBodyPart(mp_partRich);  
message.setContent(mp, "multipart/alternative");
```

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

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

[Example 3–4](#) 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 3–4 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*.

3.3.2 Addressing a Message

This section describes type of addresses and how to create address objects.

3.3.2.1 Types of Addresses

There are two types of addresses, *device addresses* and *user addresses*. A device address can be of various types, such as email addresses, instant messaging addresses, and telephone numbers. User addresses are user IDs in a user repository.

3.3.2.2 Creating Address Objects

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

3.3.2.2.1 Creating a Single Address Object [Example 3–5](#) shows code for creating a single `Address` object:

Example 3–5 Creating a Single Address Object

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

3.3.2.2.2 Creating Multiple Address Objects in a Batch [Example 3–6](#) shows code for creating multiple `Address` objects in a batch:

Example 3–6 Creating Multiple Address Objects in a Batch

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

3.3.2.2.3 Adding Sender or Recipient Addresses to a Message [Example 3–7](#) shows code for adding sender or recipient addresses to a message:

Example 3–7 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);
```

3.3.2.3 Creating a Recipient with a Failover Address

[Example 3–8](#) shows code for creating a recipient with a failover address:

Example 3–8 Creating a Single Address Object with Failover

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

3.3.2.4 API Reference for Class `MessagingFactory`

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

3.3.2.5 API Reference for Interface `Address`

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

3.3.3 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.

3.3.3.1 Sending Messages to a Group

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

The message is resolved to the users by fetching the email address of each user. If the email address of a particular user specified in the group does not exist, then that user is skipped.

To send a message to a group, use the `MessagingFactory` property to create a recipient address of type `GROUP` and send the message as shown in [Example 3–9](#).

Example 3–9 *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 3–10](#).

Example 3–10 *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);
```

3.3.3.2 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 3–11](#).

Example 3–11 *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);
```

3.3.3.3 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 JavaEE container. For more information about application role, see *Oracle Fusion Middleware Application Security Guide*.

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, you must create a recipient address of type `APPROLE` by using the `MessagingFactory` property. The message should, then, be sent. 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 3–12](#).

Example 3–12 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 `APPROLE` is eventually replaced by user addresses. However, 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, see *Oracle Fusion Middleware Application Security Guide*.

3.3.3.4 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 3–13 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);
```


3.3.4 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.

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 3–14](#) 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 [Chapter 6, "User Communication Preferences."](#)

Example 3–14 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).
```

3.4 Retrieving Message Status

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

3.4.1 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.
```

3.4.2 Asynchronous Receiving of Message Status

When asynchronously receiving status, the client application specifies 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.

3.4.2.1 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.

3.4.2.2 Default Status Listener

The client application typically sets a default status listener ([Example 3–15](#)). When the client application sends a message, delivery status callbacks for the message invoke the default listener's `onStatus` method.

Example 3–15 Default Status Listener

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

3.4.2.3 Per Message Status Listener

In this approach, the client application sends a message and specifies a `Listener` object and an optional correlator object ([Example 3–16](#)). 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.

Example 3–16 Per Message Status Listener

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

3.5 Receiving a Message

This section describes how an application receives messages. To receive a message you must first register an access point.

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.

3.5.1 Registering an Access Point

The client application can create and register an access point, specifying that it wants to receive incoming messages sent to a particular address. Since the client application has not specified any message listeners, any received messages are held by UMS. The client application can then invoke the receive method to fetch the pending messages. 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.

`AccessPoint` represents one or more device addresses to receive incoming messages.

You can 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);
```

3.5.2 Synchronous Receiving

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

You can 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. This method returns an array of messages that are immediately available in the application inbound queue.

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

3.5.3 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.

3.5.3.1 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 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.

3.5.3.2 Default Message Listener

The client application typically sets a default message listener ([Example 3-17](#)). This listener is invoked for any delivery statuses for messages sent by this client application that do not have an associated listener. 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 3-17 Default Message Listener

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

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

3.5.3.3 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 3-18](#)). 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 3–18 Per Access Point Message Listener

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

3.5.4 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.

You can 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. When a message is received, it is passed through the filter chain in order; if the message matches a filter's criterion, the filter's action is taken immediately. If no filters match the message, the default action is to accept the message and deliver it to 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);
```

3.6 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. Typically this parameter is synthesized by the API implementation and does not need to be populated by the application developer.
- 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.
- Application sessions are instance-specific. You can set the session flag on a message to ensure that any reply is received by the instance that sent the message.
- 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.

3.7 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.

3.7.1 About XA Transactions

Java Messaging Service (JMS) 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 Java Messaging Service (JMS) 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.

3.7.2 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();  
String messageID = mClient.send(message);  
transaction.commit();
```

Inbound messaging using XA

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

The messages received by a UMS driver from the UMS server and later 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.

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

    for (Message receivedMessage : messages) {
// process individual messages here.
    }
transaction.commit();
```

Once the external XA transaction is committed by the client, the messages received by the UMS driver will be permanently removed from the UMS server. If the transaction fails to commit and a rollback occurs, then the messages will be received after the transaction times out. This message will be redelivered with the `getJMSRedelivered` method returning true. For more information about this method, see *User Messaging Service Java API Reference*.

The transaction timed out can be changed by calling `setTransactionTimeout`. 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 3–19 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);

mClient.registerAccessPoint(MessagingFactory.createAccessPoint(receiverAddr),
new MyListener(), 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\(\)](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 Java EE tutorial at:

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

Example 3–20 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()
```

3.8 Using UMS Java API to Specify Message Resends

UMS allows you to schedule a message resend when the message send attempt fails. You can specify the maximum number of message resends by calling the `setMaxResend` method as shown in the following example:

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

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`, `getFailoverOrder`, and other methods, see *User Messaging Service Java API Reference*.

3.9 Configuring Security

Client applications may need to specify one or more additional configuration parameters (described in [Table 3-1](#)) to establish a secure listener.

3.10 Threading Model

Client applications that use the UMS Java API are usually multithreaded. 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.

3.10.1 Listener Threading

For asynchronous receiving described in [Section 3.4.2, "Asynchronous Receiving of Message Status"](#) and [Section 3.5.3, "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.

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.

This chapter includes the following sections:

- Section 4.1, "Introduction to the UMS Web Service API"
- Section 4.2, "Creating a UMS Client Instance and Specifying Runtime Parameters"
- Section 4.3, "Sending a Message"
- Section 4.4, "Retrieving Message Status"
- Section 4.5, "Receiving a Message"
- Section 4.6, "Configuring for a Cluster Environment"
- Section 4.7, "Using UMS Web Service API to Specify Message Resends"
- Section 4.8, "Configuring Security"
- Section 4.9, "Threading Model"
- Section 4.10, "Sample Chat Application with Web Services APIs"

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>

4.1 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 are named similarly to the corresponding Java API classes, but are in their own package space. Classes from the 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).

4.2 Creating a UMS Client Instance and Specifying Runtime Parameters

This section describes the requirements for creating a UMS Client. You can create an 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 4-1](#) lists the most common configuration parameters:

Table 4-1 Configuration Parameters Specified at Runtime

Key	Type	Use
<code>javax.xml.ws.BindingProvider.ENDPOINT_ADDRESS_PROPERTY</code>	String	Endpoint URL for the remote UMS WS. This is typically "http://<host>:<port>/ucs/messaging/webservice".
<code>javax.xml.ws.BindingProvider.USERNAME_PROPERTY</code>	String	Username to be asserted in WS-Security headers when relevant
<code>oracle.ucs.messaging.ws.ClientConstants.POLICIES</code>	String[]	Set of OWSM WS-Security policies to attach to the client's requests. These must match the policies specified on the server side.
<code>oracle.wsm.security.util.SecurityConstants.Config.KEYSTORE_RECIPIENT_ALIAS_PROPERTY</code>	String	Used for OWSM policy attachment. Specifies an alternate alias to use for looking up encryption and signing keys from the credential store.
<code>oracle.wsm.security.util.SecurityConstants.ClientConstants.WSS_CSF_KEY</code>	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.

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 4-1](#) shows code for creating a `MessagingClient` instance using username/token security, using the programmatic approach:

Example 4-1 Programmatic Approach to 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 4-2 shows code for creating a MessagingClient instance using SAML token security, using the programmatic approach:

Example 4-2 Programmatic Approach to 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".

4.3 Sending a Message

Invoking the send method causes the message to be delivered to UMS and 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.

4.3.1 Creating a Message

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

4.3.1.1 Creating a Plaintext Message

[Example 4-3](#) shows two ways to create a plaintext message using the UMS Web Service API.

Example 4-3 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")));
```

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

[Example 4-4](#) shows how to create a multipart/mixed message using the UMS Web Service API.

Example 4-4 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()));
```

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

[Example 4-5](#) shows how to create a multipart/alternative message using the UMS Web Service API.

Example 4-5 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()));

```

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

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

[Example 4–6](#) 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 4–6 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);
```

4.3.2 API Reference for Interface Message

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

4.3.3 API Reference for Enum DeliveryType

The API reference for enum `DeliveryType` can be accessed from the Javadoc.

4.3.4 Addressing a Message

This section describes type of addresses and how to create address objects.

4.3.4.1 Types of Addresses

There are two types of addresses, *device addresses* and *user addresses*. A device address can be of various types, such as email addresses, instant messaging addresses, and telephone numbers. User addresses are user IDs in a user repository.

4.3.4.2 Creating Address Objects

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

4.3.4.2.1 Creating a Single Address Object [Example 4-7](#) shows code for creating a single `Address` object:

Example 4-7 Creating a Single Address Object

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

4.3.4.2.2 Creating Multiple Address Objects in a Batch [Example 4-8](#) shows code for creating multiple `Address` objects in a batch:

Example 4-8 Creating Multiple Address Objects in a Batch

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

4.3.4.2.3 Adding Sender or Recipient Addresses to a Message [Example 4-9](#) shows code for adding sender or recipient addresses to a message:

Example 4-9 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);
```


4.3.4.3 Creating a Recipient with a Failover Address

[Example 4–10](#) shows code for creating a recipient with a failover address:

Example 4–10 *Creating a Single Address Object with Failover*

```
String recipientWithFailoverStr = "Email:john@example.com, IM:john@example.com";
Address recipient = MessagingFactory.createAddress(recipientWithFailoverStr);
```

4.3.4.4 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.

4.3.4.5 API Reference for Class `MessagingFactory`

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

4.3.4.6 API Reference for Interface `Address`

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

4.3.5 User Preference Based Messaging

When sending a message to a user recipient (to leverage the user's messaging preferences), you can pass facts (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.

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

[Example 4–11](#) 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 [Chapter 6, "User Communication Preferences."](#)

Example 4–11 *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
MessagingFactory.setMetadata(message, oracle.sdp.client.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).
```

4.4 Retrieving Message Status

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

4.4.1 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.
```

4.4.2 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 Java 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.

4.4.2.1 Creating a Listener Programmatically

Listeners are purely programmatic. 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!");
    }
}

```

4.4.2.2 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 4–12](#)):

Example 4–12 Publish the Callback Service

```

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

```

4.4.2.3 Stop a Dynamically Published Endpoint

To stop a dynamically published endpoint, call the `stop()` method on the `Endpoint` object returned from `Endpoint.publish()` ([Example 4–13](#)).

Example 4–13 Stop a Dynamically Published Endpoint

```

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

```

4.4.2.4 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.

4.5 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.

4.5.1 Registering an Access Point

The client application can create and register an access point, specifying that it wants to receive incoming messages sent to a particular address.

The client application can then invoke the receive method to fetch the pending messages. 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.

`AccessPoint` represents one or more device addresses to receive incoming messages.

You can 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(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(AccessPointType.NUMBER_RANGE,
        DeliveryType.SMS, "9000,9999");
messagingClient.registerAccessPoint(accessPointRangeAddress);
```

4.5.2 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.

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.

It performs a nonblocking call, so if no message is currently available, the method returns null.

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

4.5.3 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 Java EE servlet module. However, a servlet-based listener implementation is also acceptable.

4.5.3.1 Creating a Listener Programmatically

Listeners are purely programmatic. 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.

4.5.3.2 Default Message Listener

The client application typically sets a default message listener ([Example 4-14](#)). This listener is invoked for any delivery statuses for messages sent by this client application that do not have an associated listener. 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 4-14 Default Message Listener

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

4.5.3.3 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 4–15). 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 4–15 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);
```

4.5.4 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.

You can 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. When a message is received, it is passed through the filter chain in order; if the message matches a filter's criterion, the filter's action is taken immediately. If no filters match the message, the default action is to accept the message and deliver it to the application.

For example, to reject a message with the subject "spam":

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

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

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

4.6 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.
- The `ApplicationInstanceName` configuration parameter enables you to distinguish instances from one another.
- Application sessions are instance-specific. You can set the session flag on a message to ensure that any reply is received by the instance that sent the message.

- 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.

4.7 Using UMS Web Service API to Specify Message Resends

UMS allows you to schedule a message resend when the message send attempt fails. You can specify the maximum number of message resends by calling the `setMaxResend` method as shown 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*.

4.8 Configuring Security

The following sections discuss security considerations:

- [Section 4.8.1, "Client and Server Security"](#)
- [Section 4.8.2, "Listener or Callback Security"](#)

4.8.1 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 4-2](#) 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 4-1](#) for more details about configuring SAML security in a UMS web service client.

4.8.2 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 4–16](#) illustrates how to establish a secure callback endpoint using username token security:

Example 4–16 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);
```

4.9 Threading Model

Instances of the WS 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.

4.10 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>.

This section describes the following topics:

- [Section 4.10.1, "Overview"](#)
- [Section 4.10.2, "Running the Pre-Built Sample"](#)
- [Section 4.10.3, "Testing the Sample"](#)
- [Section 4.10.4, "Creating a New Application Server Connection"](#)

4.10.1 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.

4.10.1.1 Provided Files

The following files are included in the sample application:

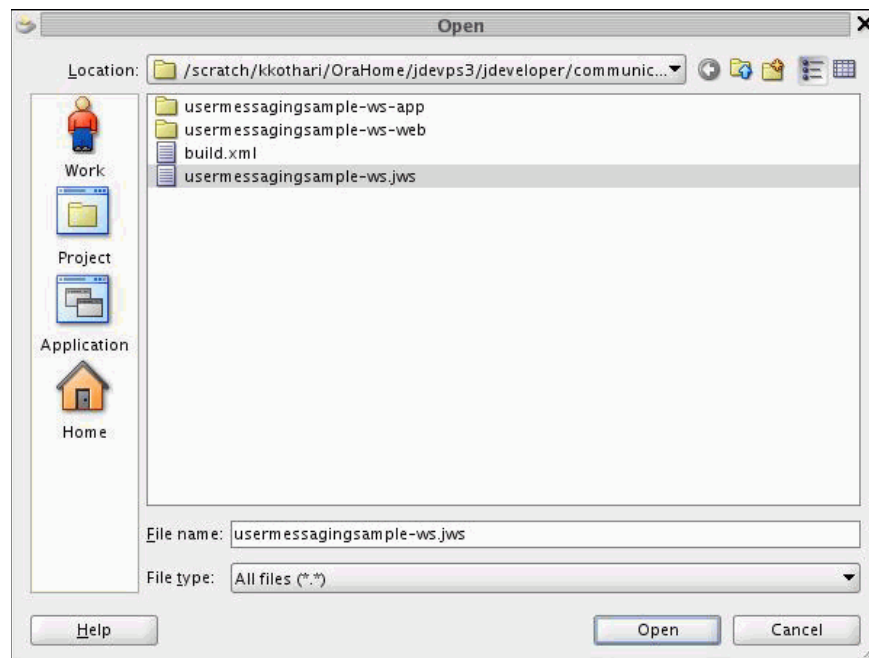
- usermessagingsample-ws-src.zip – the archive containing the source code and Oracle JDeveloper project files.
- usermessagingsample-ws.ear - the pre-built sample application that can be deployed to the container.

4.10.2 Running the Pre-Built Sample

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

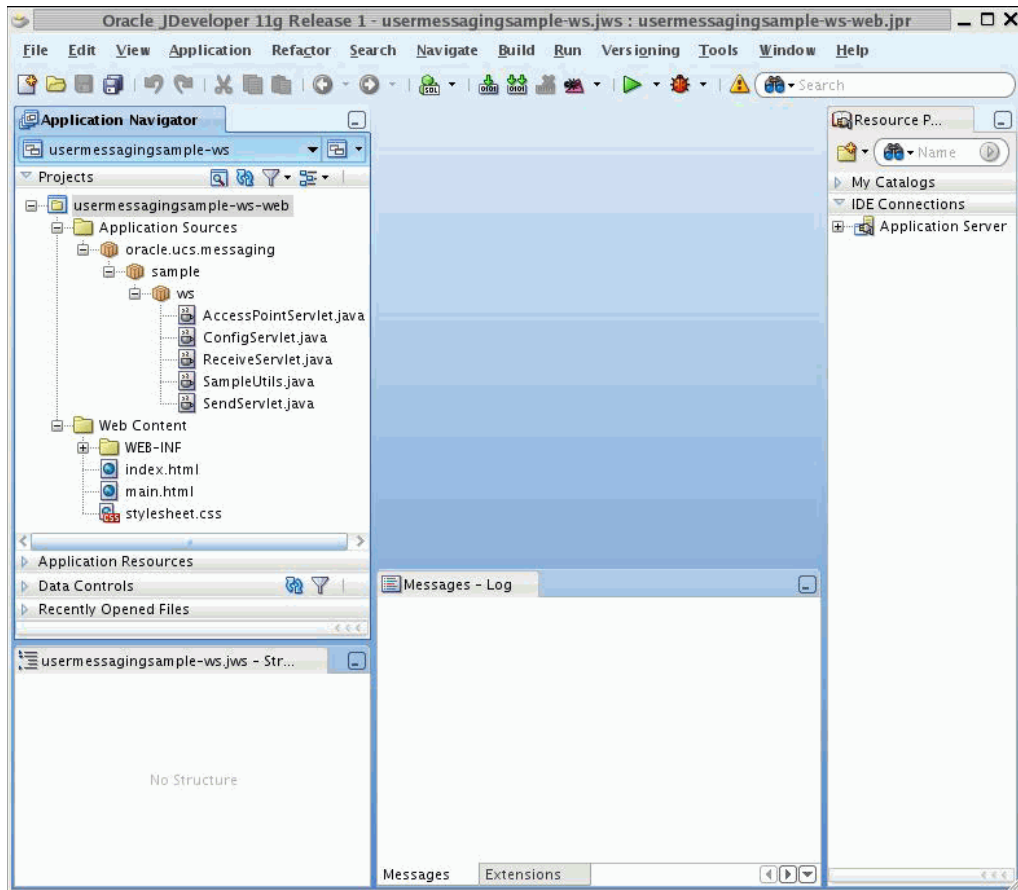
1. Extract "usermessagingsample-ws-src.zip" and open **usermessagingsample-ws.jws** in Oracle JDeveloper.

Figure 4–1 Opening the Project in Oracle JDeveloper



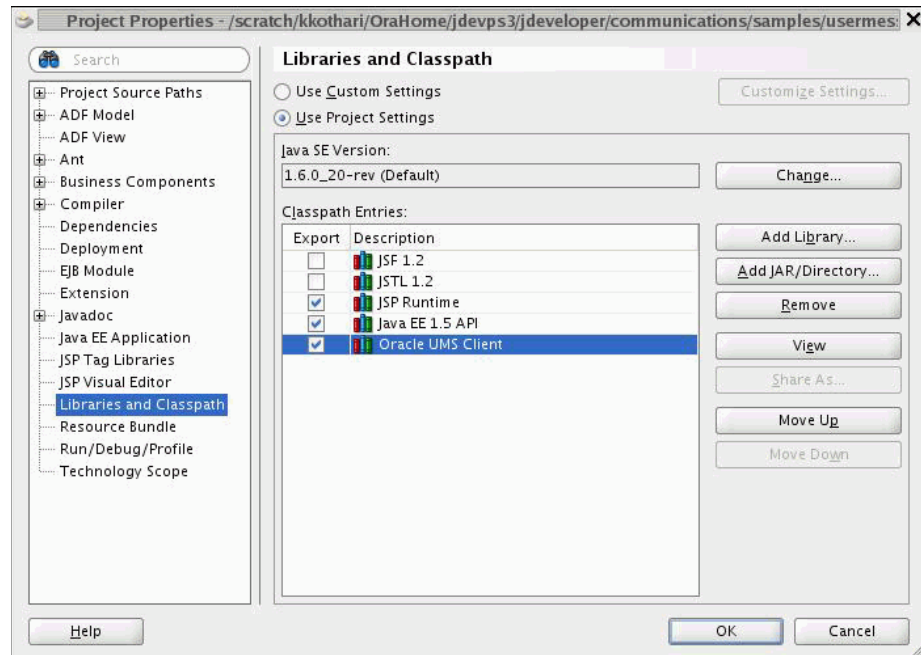
In the Oracle JDeveloper main window the project appears.

Figure 4-2 Oracle JDeveloper Main Window

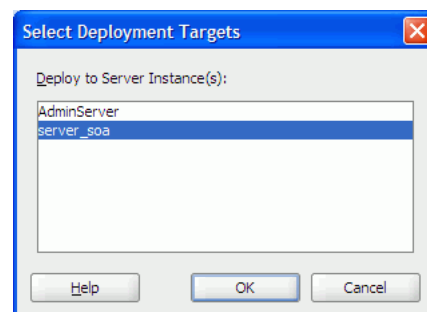


The application contains one web module. All of the source code for the application is in place.

2. Satisfy the build dependencies for the sample application by ensuring the "Oracle UMS Client" library is used by the web module.
 1. In the Application Navigator, right-click web module `usermessagingsample-ws-war`, and select **Project Properties**.
 2. In the left pane, select **Libraries and Classpath**.

Figure 4–3 Adding a Library

3. Click **OK**.
3. Create an Application Server Connection by right-clicking the project in the navigation pane and selecting **New**. Follow the instructions in [Section 4.10.4, "Creating a New Application Server Connection"](#).
4. Deploy the project by selecting the **usermessasgingsample-ws** project, **Deploy**, **usermessasgingsample-ws**, to, and **SOA_server** ([Figure 4–4](#)).

Figure 4–4 Deploying the Project

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.

4.10.3 Testing the Sample

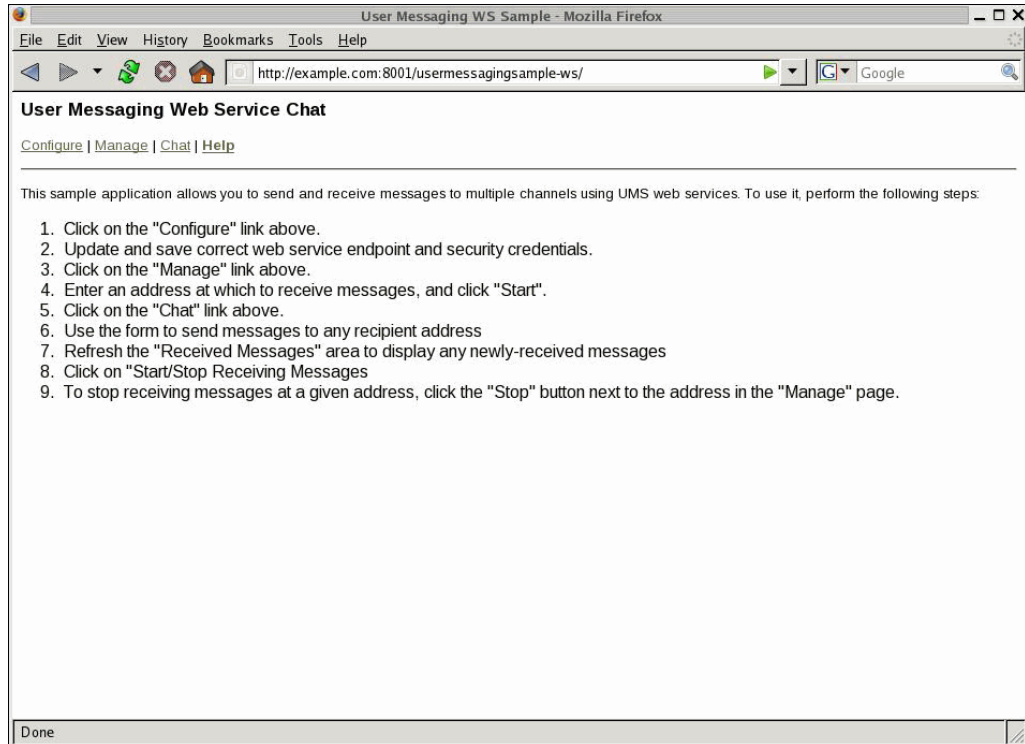
Perform the following steps to run and test the sample:

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 (Figure 4-5). This page contains navigation tabs and instructions for the application.

Figure 4-5 Messaging Web Services Sample Web Page



3. Click **Configure** and enter the following values (Figure 4-6):
 - 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).

Figure 4–6 Configuring the Web Service Endpoints and Credentials

User Messaging WS Sample - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://example.com:8001/usermessagingsample-ws/config

User Messaging Web Service Chat

[Configure](#) | [Manage](#) | [Chat](#) | [Help](#)

Update Web Service Endpoints and Credentials:

Web Service endpoint:

Username:

Password:

Policies:

4. Click **Save**.
5. Click **Manage**.
6. Enter an address and optional keyword at which to receive messages (Figure 4–7).

Figure 4–7 Registering an Access Point

User Messaging WS Sample - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://example.com:8001/usermessagingsample-ws/accesspoints

User Messaging Web Service Chat

[Configure](#) | [Manage](#) | [Chat](#) | [Help](#)

Register an address at which to receive messages:

Address: Keyword:

Current Access Points:

Done

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.

4.10.4 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 [Section A.3, "Creating a New Application Server Connection"](#) to create an Application Server Connection.

Parlay X Web Services Multimedia Messaging API

This chapter describes the Parlay X Multimedia Messaging web service that is available with Oracle User Messaging Service and how to use the Parlay X Web Services Multimedia Messaging API to send and receive messages through Oracle User Messaging Service.

Note: The Parlay X Multimedia Messaging API (described in this chapter) is deprecated. Use the User Messaging Service Java API instead, as described in [Chapter 3, "Sending and Receiving Messages using the User Messaging Service Java API"](#).

This chapter includes the following sections:

- [Section 5.1, "Introduction to Parlay X Messaging Operations"](#)
- [Section 5.2, "Send Message Interface"](#)
- [Section 5.3, "Receive Message Interface"](#)
- [Section 5.4, "Oracle Extension to Parlay X Messaging"](#)
- [Section 5.5, "Parlay X Messaging Client API and Client Proxy Packages"](#)
- [Section 5.6, "Sample Chat Application with Parlay X APIs"](#)

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>

Note: Oracle User Messaging Service also ships with a Java client library that implements the Parlay X API.

5.1 Introduction to Parlay X Messaging Operations

The following sections describe the semantics of each of the supported operations along with implementation-specific details for the Parlay X Gateway. The following tables, describing input/output message parameters for each operation, are taken directly from the Parlay X specification.

Oracle User Messaging Service implements a subset of the Parlay X 2.1 Multimedia Messaging specification. Specifically Oracle User Messaging Service supports the `SendMessage` and `ReceiveMessage` interfaces. The `MessageNotification` and `MessageNotificationManager` interfaces are not supported.

5.2 Send Message Interface

The `SendMessage` interface enables you to send a message to one or more recipient addresses by using the `sendMessage` operation, or get the delivery status for a previously sent message by using the `getMessageDeliveryStatus` operation. The following requirements apply:

- A recipient address must conform to the address format requirements of Oracle User Messaging Service (in addition to being a valid URI). The general format is *delivery_type:protocol_specific_address*, such as `email:user@domain`, `sms:5551212` or `im:user@jabberdomain`.
- Certain characters are not allowed in URIs; if it is necessary to include them in an address they can be encoded or escaped. Refer to the JavaDoc for `java.net.URI` for details on how to create a properly encoded URI.
- While the WSDL specifies that sender addresses can be any string, Oracle User Messaging Service requires that they be valid Messaging addresses.
- Oracle User Messaging Service requires that you specify sender addresses on a per-delivery type basis. So for a sender address to apply to a recipient of a given delivery type, say EMAIL, the sender address must also have delivery type of EMAIL. Since this operation allows multiple recipient addresses but only one sender address, the sender address only applies to the recipients with the same delivery type.
- Oracle User Messaging Service does not support the `MessageNotification` interface, and therefore do not produce delivery receipts, even if a `receiptRequest` is specified. In other words, the `receiptRequest` parameter is ignored.

5.2.1 sendMessage Operation

Table 5–1 describes message descriptions for the `sendMessageRequest` input in the `sendMessage` operation.

Table 5–1 *sendMessage Input Message Descriptions*

Part Name	Part Type	Optional	Description
addresses	xsd:anyURI[0..unbounded]	No	Destination address for this Message.
senderAddress	xsd:string	Yes	Message sender address. This parameter is not allowed for all 3rd party providers. The Parlay X server must handle this according to a SLA for the specific application and its use can therefore result in a <code>PolicyException</code> .
subject	xsd:string	Yes	Message subject. If mapped to SMS, this parameter is used as the senderAddress, even if a separate senderAddress is provided.

Table 5–1 (Cont.) sendMessage Input Message Descriptions

Part Name	Part Type	Optional	Description
priority	MessagePriority	Yes	Priority of the message. If not present, the network assigns a priority based on the operator policy.Charging to apply to this message.
charging	common:ChargingInformation	Yes	Charging to apply to this message.
receiptRequest	common:SimpleReference	Yes	Defines the application endpoint, interface name and correlator that is used to notify the application when the message has been delivered to a terminal or if delivery is impossible.

[Table 5–2](#) describes `sendMessageResponse` output messages for the `sendMessage` operation.

Table 5–2 sendMessageResponse Output Message Descriptions

Part Name	Part Type	Optional	Description
result	xsd:string	No	This correlation identifier is used in a <code>getMessageDeliveryStatus</code> operation invocation to poll for the delivery status of all sent messages.

5.2.2 getMessageDeliveryStatus Operation

The `getMessageDeliveryStatus` operation gets the delivery status for a previously sent message. The input "requestIdentifier" is the "result" value from a `sendMessage` operation. This is the same identifier that is referred to as a Message ID in other Messaging documentation.

[Table 5–3](#) describes the `getMessageDeliveryStatusRequest` input messages for the `getMessageDeliveryStatus` operation.

Table 5–3 getMessageDeliveryStatusRequest Input Message Descriptions

Part Name	Part Type	Optional	Description
registrationIdentifier	xsd:string	No	Identifier related to the delivery status request.

[Table 5–4](#) describes the `getMessageDeliveryStatusResponse` output messages for the `getMessageDeliveryStatus` operation.

Table 5–4 getMessageDeliveryStatusResponse Output Message Descriptions

Part Name	Part Type	Optional	Description
result	DeliveryInformation [0..unbounded]	Yes	An array of status of the messages that were previously sent. Each array element represents a sent message, its destination address and its delivery status.

5.3 Receive Message Interface

The ReceiveMessage interface has three operations. The `getReceivedMessages` operation polls the server for any messages received since the last invocation of `getReceivedMessages`. Note that `getReceivedMessages` does not necessarily return any message content; it generally only returns message metadata.

The other two operations, `getMessage` and `getMessageURIs`, are used to retrieve message content.

5.3.1 getReceivedMessages Operation

This operation polls the server for any received messages. Note the following requirements:

- The registration ID parameter is a string that identifies the endpoint address for which the application wants to receive messages. See the discussion of the ReceiveMessageManager interface for more details.
- The Parlay X specification says that if the registration ID is not specified, all messages for this application should be returned. However, the WSDL says that the registration ID parameter is mandatory. Therefore our implementation treats the empty string ("") as the "not-specified" value. If you call `getReceivedMessages` with the empty string as your registration ID, you get all messages for this application. Therefore the empty string is not an allowed value of registration ID when calling `startReceiveMessages`.
- According to the Parlay X specification, if the received message content is "pure ASCII text", then the message content is returned inline within the MessageReference object, and the messageIdentifier (Message ID) element is null. Our implementation treats any content with Content-Type "text/plain", and with encoding "us-ascii" as "pure ASCII text" for the purposes of this operation. As per the MIME specification, if no encoding is specified, "us-ascii" is assumed, and if no Content-Type is specified, "text/plain" is assumed.
- The priority parameter is currently ignored.

Table 5–5 describes the `getReceivedMessagesRequest` input messages for the `getReceivedMessages` operation.

Table 5–5 *getReceivedMessagesRequest Input Message Descriptions*

Part Name	Part Type	Optional	Description
registrationIdentifier	xsd:string	No	Identifies the off-line provisioning step that enables the application to receive notification of Message reception according to the specified criteria.
priority	MessagePriority	Yes	The priority of the messages to poll from the Parlay X gateway. All messages of the specified priority and higher are retrieved. If not specified, all messages shall be returned, that is, the same as specifying "Low."

Table 5–6 describes the `getReceivedMessagesResponse` output messages for the `getReceivedMessages` operation.

Table 5–6 *getReceivedMessagesResponse Output Message Descriptions*

Part Name	Part Type	Optional	Description
registrationIdentifier	xsd:string	No	Identifies the off-line provisioning step that enables the application to receive notification of Message reception according to the specified criteria.
priority	MessagePriority	Yes	The priority of the messages to poll from the Parlay X gateway. All messages of the specified priority and higher are retrieved. If not specified, all messages shall be returned. This is equal to specifying Low.

5.3.2 getMessage Operation

The `getMessage` operation retrieves message content, using a message ID from a previous invocation of `getReceivedMessages`. There is no SOAP body in the response message; the content is returned as a single SOAP attachment.

Table 5–7 describes the `getMessageRequest` input messages for the `getMessage` operation.

Table 5–7 *getMessageRequest Input Message Descriptions*

Part Name	Part Type	Optional	Description
messageRefIdentifier	xsd:string	No	The identity of the message.

There are no `getMessageResponse` output messages for the `getMessage` operation.

5.3.3 getMessageURIs Operation

The `getMessageURIs` retrieves message content as a list of URIs. Note the following requirements:

- These URIs are HTTP URLs that can be dereferenced to retrieve the content.
- If the inbound message has a Content-Type of "multipart", then there are multiple URIs returned, one per subpart. If the Content-Type is not "multipart", then a single URI are returned.
- Per the Parlay X specification, if the inbound messages a body text part, defined as "the message body if it is encoded as ASCII text", it is returned inline within the MessageURI object. For the purposes of our implementation, you define this behavior as follows:
 - If the message's Content-Type is "text/*" (any text type), and if the charset parameter is "us-ascii", then the content is returned inline in the MessageURI object. There are no URIs returned since there is no content other than what is returned inline.
 - If the message's Content-Type is "multipart/" (any multipart type), and if the first body part's Content-Type is "text/" with charset "us-ascii", then that part is returned inline in the MessageURI object, and there are no URIs returned corresponding to that part.

- Per the MIME specification, if the charset parameter is omitted, the default value of "us-ascii" is assumed. If the Content-Type header is not specified for the message, then a Content-Type of "text/plain" is assumed.

Table 5–8 describes the `getMessageURIsRequest` input messages for the `getMessageURIs` operation.

Table 5–8 *getMessageURIsRequest Input Message Descriptions*

Part Name	Part Type	Optional	Description
messageRefIdentifier	xsd:string	No	The identity of the message to retrieve.

Table 5–9 describes the `getMessageURIsResponse` output messages for the `getMessageURIs` operation.

Table 5–9 *getMessageURIsResponse Output Message Descriptions*

Part Name	Part Type	Optional	Description
result	MessageURI	No	Contains the complete message, consisting of the textual part of the message, if such exists, and a list of file references for the message attachments, if any.

5.4 Oracle Extension to Parlay X Messaging

The Parlay X Messaging specification leaves certain parts of the messaging flow undefined. The main area that is left undefined is the process for binding a client to an address for synchronous receiving (through the `ReceiveMessage` interface).

Oracle User Messaging Service includes an extension interface to Parlay X to support this process. The extension is implemented as a separate WSDL in an Oracle XML namespace to indicate that it is not an official part of Parlay X. Clients can choose to not use this additional interface or use it in some modular way such that their core messaging logic remains fully compliant with the Parlay X specification.

5.4.1 ReceiveMessageManager Interface

`ReceiveMessageManager` is the Oracle-specific interface for managing client registrations for receiving messages. Clients use this interface to start and stop receiving messages at a particular address. (This is analogous to the concept of registering/unregistering access points in the Messaging API).

5.4.1.1 startReceiveMessage Operation

Invoking this operation allows a client to bind itself to a given endpoint for receiving messages. Note the following requirements:

- An endpoint consists of an address and an optional "criteria", defined by the Parlay X specification as the first white space-delimited token of the message subject or content.
- In addition to the endpoint information, the client also specifies a "registration ID" when invoking this operation; this ID is just a unique string which can be used later to refer to this particular binding in the `stopReceiveMessage` and `getReceivedMessages` operations.

- If an endpoint is already registered by another client application, or the registration ID is already being used, a Policy Error results.
- Certain characters are not allowed in URIs; if it is necessary to include them in an address they can be encoded/escaped. See the javadoc for `java.net.URI` for details on how to create a properly encoded URI. For example, when registering to receive XMPP messages you must specify an address such as `IM:jabber|user@example.com`, however the pipe (`|`) character is not allowed in URIs, and must be escaped before submitting to the server.
- There is no guarantee that the server can actually receive messages at a given endpoint address. That depends on the overall configuration of Oracle User Messaging Service, particularly the Messaging drivers that are deployed in the system. No error is indicated if a client binds to an address where the server cannot receive messages.

The `startReceiveMessage` operation has the following inputs and outputs:

[Table 5–10](#) describes the `startReceiveMessageRequest` input messages for the `startReceiveMessage` operation.

Table 5–10 *startReceiveMessageRequest Input Message Descriptions*

Part Name	Part Type	Optional	Description
registrationIdentifier	xsd:string	No	A registration identifier.
messageServiceActivationNumber	xsd:anyURI	No	Message Service Activation Number.
criteria	xsd:string	Yes	Descriptive string.

There are no `startReceiveMessageResponse` output messages for the `startReceiveMessage` operation.

5.4.1.2 stopReceiveMessage Operation

Invoking this operation removes the previously-established binding between a client and a receiving endpoint. The client specifies the same registration ID that was supplied when `startReceiveMessage` was called to identify the endpoint binding that is being broken. If there is no corresponding registration ID binding known to the server for this application, a Policy Error results.

[Table 5–11](#) describes the `stopReceiveMessageRequest` input messages for the `stopReceiveMessage` operation.

Table 5–11 *stopReceiveMessageRequest Input Message Descriptions*

Part Name	Part Type	Optional	Description
registrationIdentifier	xsd:string	No	A registration identifier.

There are no `stopReceiveMessageResponse` output messages for the `stopReceiveMessage` operation.

5.5 Parlay X Messaging Client API and Client Proxy Packages

While it is possible to assemble a Parlay X Messaging Client using only the Parlay X WSDL files and a web service assembly tool, prebuilt web service stubs and interfaces are provided for the supported Parlay X Messaging interfaces. Due to difficulty in

assembling a web service with SOAP attachments in the style mandated by Parlay X, Oracle recommends the use of the provided API rather than starting from WSDL.

For a complete listing of the classes available in the Parlay X Messaging API, see the Messaging JavaDoc. The main entry points for the API are through the following client classes:

- `oracle.sdp.parlayx.multimedia_messaging.send.SendMessageClient`
- `oracle.sdp.parlayx.multimedia_messaging.receive.ReceiveMessageClient`
- `oracle.sdp.parlayx.multimedia_messaging.extension.receive_manager.ReceiveMessageManager`

Each client class allows a client application to invoke the operations in the corresponding interface. Additional web service parameters such as the remote gateway URL and any required security credentials, are provided when an instance of the client class is constructed. See the Javadoc for more details. The security credentials are propagated to the server using standard WS-Security headers, as mandated by the Parlay X specification.

The general process for a client application is to create one of the client classes above, set the necessary configuration items (endpoint, username, password), then invoke one of the business methods (for example, `SendMessageClient.sendMessage()`, and so on). For examples of how to use this API, see the Messaging samples on Oracle Technology Network (OTN), and specifically `usermessagingsample-parlayx-src.zip`.

5.6 Sample Chat Application with Parlay X APIs

This chapter describes how to create, deploy and run the sample chat application with Parlay X APIs provided with Oracle User Messaging Service on OTN.

This chapter contains the following sections:

- [Section 5.6.1, "Overview"](#)
- [Section 5.6.2, "Running the Pre-Built Sample"](#)
- [Section 5.6.3, "Testing the Sample"](#)
- [Section 5.6.4, "Creating a New Application Server Connection"](#)

5.6.1 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 standards-based Parlay X Web Service APIs to interact with a User Messaging server. The sample application includes web service proxy code for each of three web service interfaces: the `SendMessage` and `ReceiveMessage` services defined by Parlay X, and the `ReceiveMessageManager` service which is an Oracle extension to Parlay X. 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.

5.6.1.1 Provided Files

The following files are included in the sample application:

- Project – the directory containing the archived Oracle JDeveloper project files.
- Readme.txt.
- Release notes

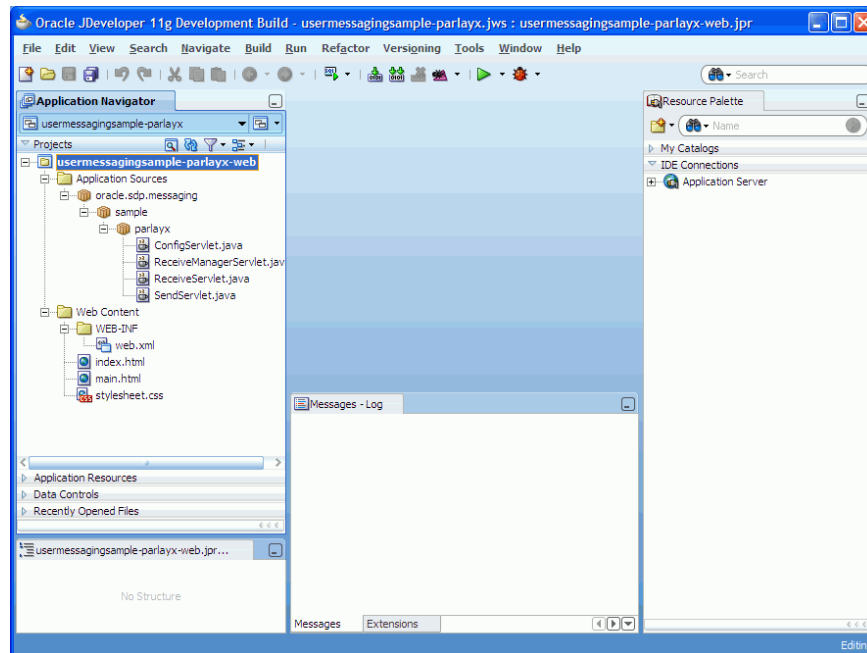
5.6.2 Running the Pre-Built Sample

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

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

In the Oracle JDeveloper main window the project appears.

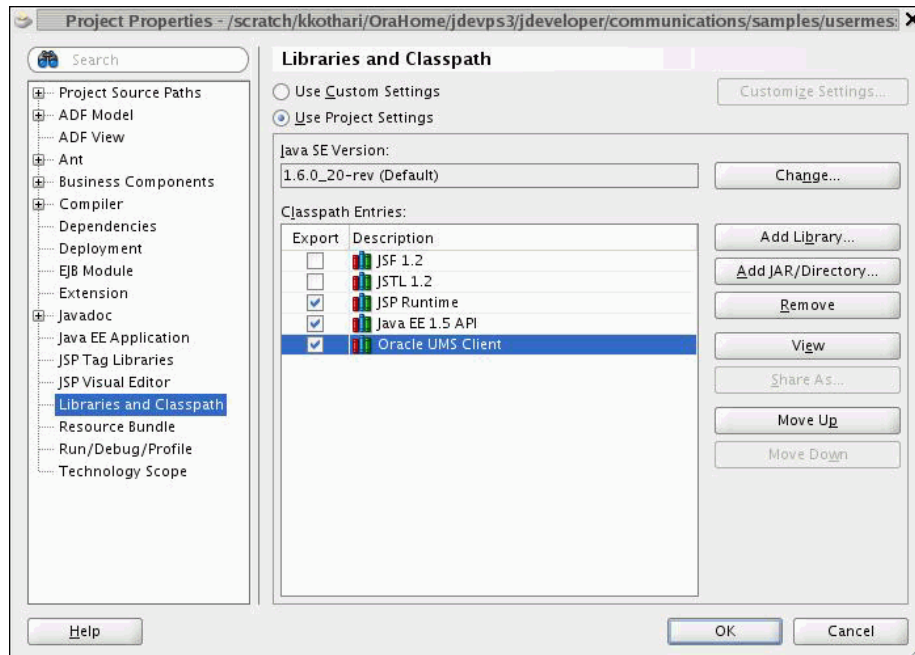
Figure 5–1 Oracle JDeveloper Main Window



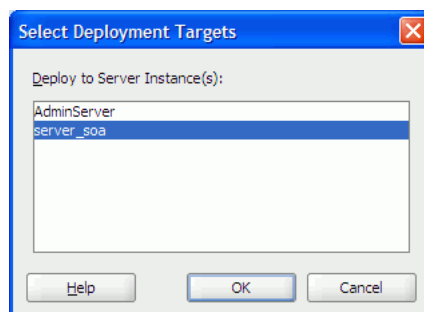
2. In Oracle JDeveloper, select **File > Open...**, then navigate to the directory above and open workspace file "usermessagingsample-parlayx.jws".

This opens the precreated JDeveloper application for the Parlay X sample application. The application contains one web module. All of the source code for the application is in place. You must configure the parameters that are specific to your installation.

3. Satisfy the build dependencies for the sample application by ensuring the "Oracle UMS Client" library is used by the web module.
 1. In the Application Navigator, right-click web module `usermessagingsample-parlayx-war`, and select **Project Properties**.
 2. In the left pane, select **Libraries and Classpath**.

Figure 5–2 Adding a Library

3. Click **OK**.
4. Create an Application Server Connection by right-clicking the project in the navigation pane and selecting **New**. Follow the instructions in [Section 5.6.4, "Creating a New Application Server Connection"](#).
5. Deploy the project by selecting the **usermessasingsample-parlayx** project, **Deploy, usermessasingsample-parlayx, to, and SOA_server** ([Figure 5–3](#)).

Figure 5–3 Deploying the Project

6. Verify that the message **Build Successful** appears in the log.
7. Enter the default revision and click **OK**.
8. 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 configure a default device for the user receiving the message in User Communication Preferences, as described in the following sections.

Note: Refer to *Administering Oracle SOA Suite and Oracle Business Process Management Suite* for more information.

5.6.3 Testing the Sample

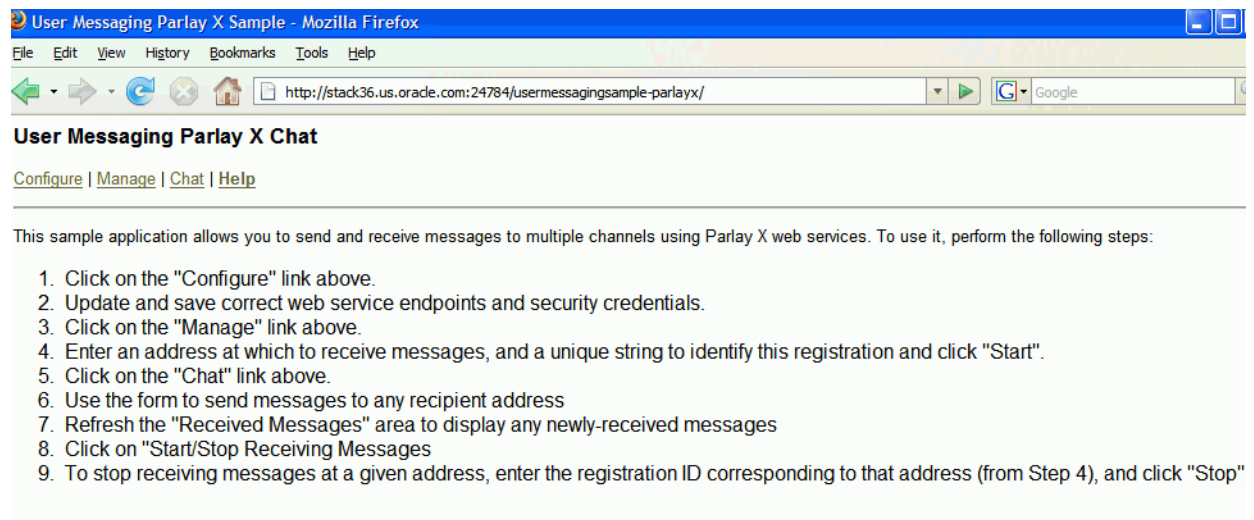
Perform the following steps to run and test the sample:

1. Open a web browser.
2. Navigate to the URL of the application as follows, and log in:

`http://host:port/usermessagingsample-parlayx/`

The Messaging Parlay X Sample web page appears (Figure 5–4). This page contains navigation tabs and instructions for the application.

Figure 5–4 Messaging Parlay X Sample Web Page



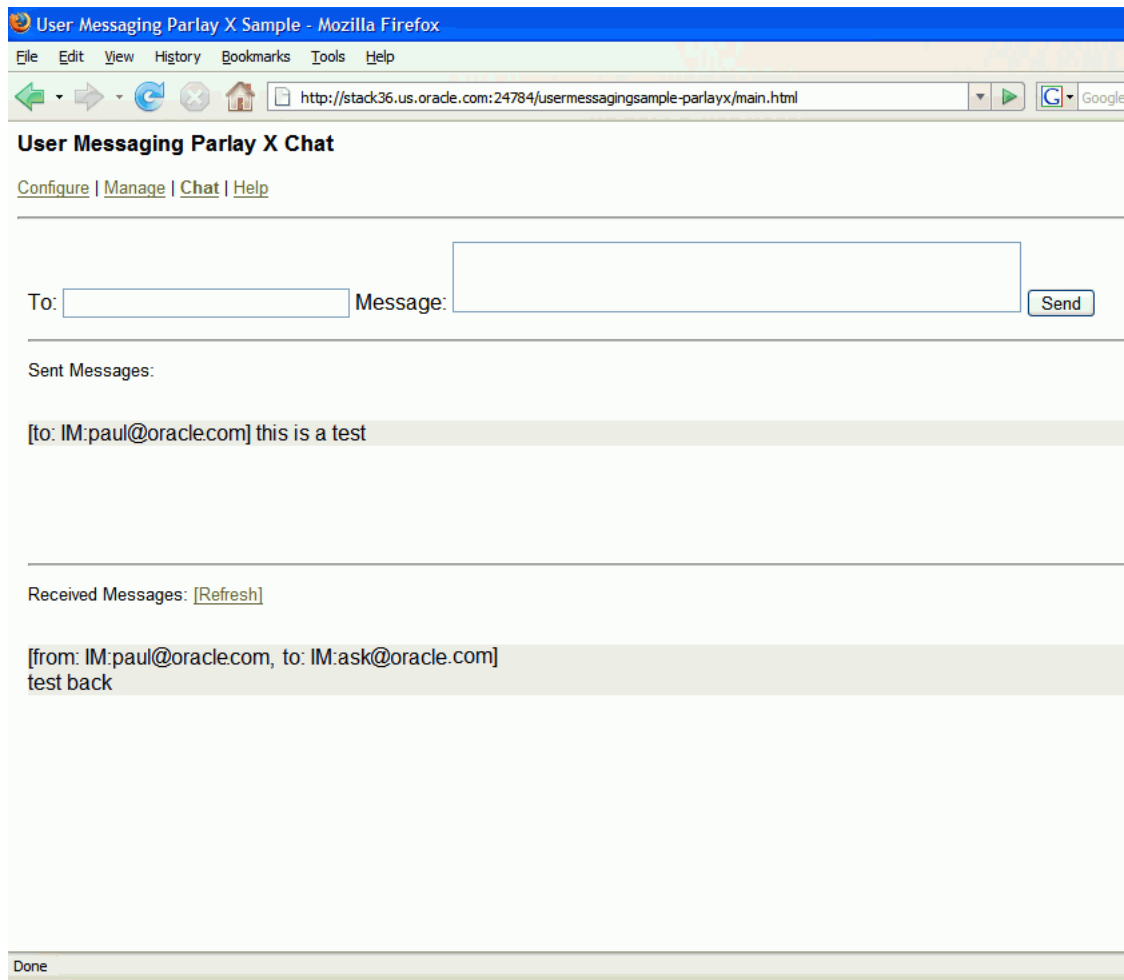
3. Click **Configure** and enter the following values (Figure 5–5):
 - Specify the Send endpoint. For example, `http://localhost:port/sdpmessaging/parlayx/SendMessageService`
 - Specify the Receive endpoint. For example, `http://localhost:port/sdpmessaging/parlayx/ReceiveMessageservice`
 - Specify the Receive Manager endpoint. For example, `http://localhost:port/sdpmessaging/parlayx/ReceiveMessageMessageService`
 - Specify the Username and Password.
 - Specify a Policy (required if the User Messaging Service instance has WS security enabled).

Figure 5–5 Configuring the Web Service Endpoints and Credentials

4. Click **Save**.
5. Click **Manage**.
6. Enter a Registration ID to specify the registration and address at which to receive messages (Figure 5–6). You can also use this page to stop receiving messages at an address.

Figure 5–6 Specifying a Registration ID

7. Click **Start**.
Verify that the message Registration operation succeeded appears.
8. Click **Chat** (Figure 5–7).
9. Enter recipients in the **To:** field in the format illustrated in Figure 5–7.
10. Enter a message.
11. Click **Send**.
12. Verify that the message is received.

Figure 5–7 Running the Sample

5.6.4 Creating a New Application Server Connection

Perform the steps in [Section A.3, "Creating a New Application Server Connection"](#) to create an Application Server Connection.

User Communication Preferences

This chapter describes the User Communication Preferences (UCP). It describes how to work with communication channels and to create contact rules using messaging filters. This chapter also discusses how to manage communication preferences from a web interface by managing channels and filters. It also provides information for system administrators about configuring User Communication Preferences; and for developers about integrating their applications with User Communication Preferences.

Note: To learn about the architecture and components of Oracle User Messaging Service, see *Administering Oracle User Messaging Service*.

This chapter includes the following sections:

- [Section 6.1, "Introduction to User Communication Preferences"](#)
- [Section 6.2, "Managing User Preferences"](#)
- [Section 6.3, "Administering User Communication Preferences"](#)
- [Section 6.4, "Integrating UCP Web User Interface"](#)
- [Section 6.5, "Java Application Interface"](#)

6.1 Introduction to User Communication Preferences

User Communication Preferences allows a user who has access to multiple channels to control how, when, and where they receive messages. Users define filters, or delivery preferences, that specify which channel a message should be delivered to, and under what circumstances. Information about a user's channels and filters are stored in any database supported for use with Oracle Fusion Middleware. Since preferences are stored in a database, this information is shared across all instances of User Communication Preferences in a domain.

UCP does not provide services for message delivery, rather provides user interface and APIs to access and manage a user's channels and delivery preferences. When a message is addressed to a user, UMS acquires the user's delivery preferences from UCP services and sends the message according to the user's preferences. For an application developer, User Communication Preferences provide increased flexibility. By sending messages through UMS, an application is indirectly using UCP service. Applications can also directly access UCP services by calling UCP APIs to access and manage a user's preferences and by integrating with UCP using task flow library to provide web user interface.

6.1.1 Terminology

User Communication Preferences defines the following terminology:

- **Channel:** a combination of delivery type and address. For instance, a cell phone number *6503334444* can be used in two channels, *SMS:6503334444* and *VOICE:6503334444*, where *SMS* and *VOICE* are delivery types and *6503334444* is the delivery address.
- **Channel address:** one of the addresses that a channel can communicate with.
- **Filter:** a message delivery preference rule that controls how, when, and where a user receives messages.
- **System term:** a pre-defined business term where the fact for the term is automatically supplied by UCP service.
- **Business term:** a named attribute for messages, such as a subject. The fact for a business term can be extracted from messages or supplied by applications and used to compare with a specified value in a filter condition to select the filter.
- **Fact:** actual value for a business term extracted from messages or supplied by applications.
- **Condition:** a combination of a business term, an operator and a specified value. The fact about a message is used to compare against the value to evaluate the *truth* of the condition.
- **Action:** the action to be taken if the specified conditions in a filter are true, such as *do not send message*, *send to first available channel*, or *send to all selected channels*.

6.2 Managing User Preferences

This section provides information about managing user preferences using web user interface. This section discusses the following topics:

- [Managing Communication Channels](#)
- [Managing Filters](#)
- [Configuring Preference Settings](#)

6.2.1 Managing Communication Channels

A communication channel defines an address (such as a phone number) and a type (such as a short message service or SMS) for message delivery.

User Communication Preferences (UCP) creates a few channels automatically for a user based on the user profile settings in the Identity Store. These channels, called as IDM channels, can be used for message delivery. A *POPUP* or *WORKLIST* channel is automatically created when you deploy the corresponding driver. This channel will be removed when the driver is undeployed. The address value for this channel is the user's login ID. Users cannot modify these channels by using the UCP UI.

Note: For messaging drivers you cannot specify a channel address with spaces. As the channel address value is the login ID, do not use spaces when you create the login ID for the user.

Alternately, users can create, modify, and delete user-defined channels. These are called *USER* channels. Any channel that a user creates is associated with that user's

system ID. In Oracle User Communication Preferences, channels represent both physical channels, such as mobile phones, and also email client applications running on desktops, and are configurable in the UCP UI.

6.2.1.1 Creating a Channel

To create a USER channel, perform the following tasks:

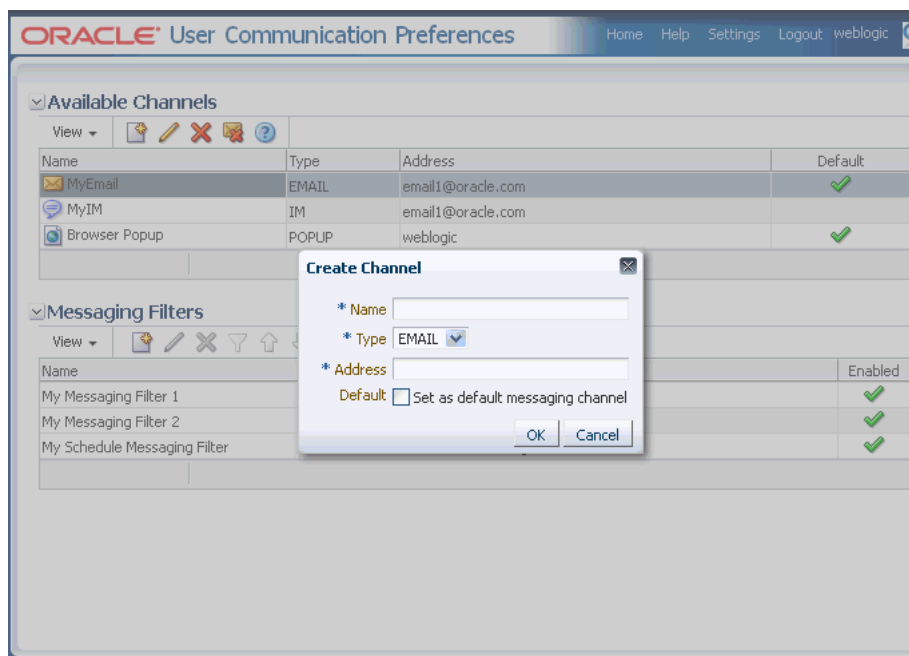
1. Click the **Create** icon as shown in [Figure 6–1](#), located in the toolbar under Available Channels.

Figure 6–1 The Create Icon



The Create Channel dialog box appears as shown in [Figure 6–2](#).

Figure 6–2 Creating a Channel



2. Enter a name for the channel in the **Name** field.
3. Select the delivery type from the **Type** drop-down list.
4. Enter the address appropriate to the delivery type you selected.
5. Select the **Default** check box if you want to set this channel as a default channel. You can have multiple default channels.
6. Click **OK** to create the channel. The new channel appears in the *Available Channels* section. The *Available Channels* page enables you to modify or delete the channel.

6.2.1.2 Modifying a Channel

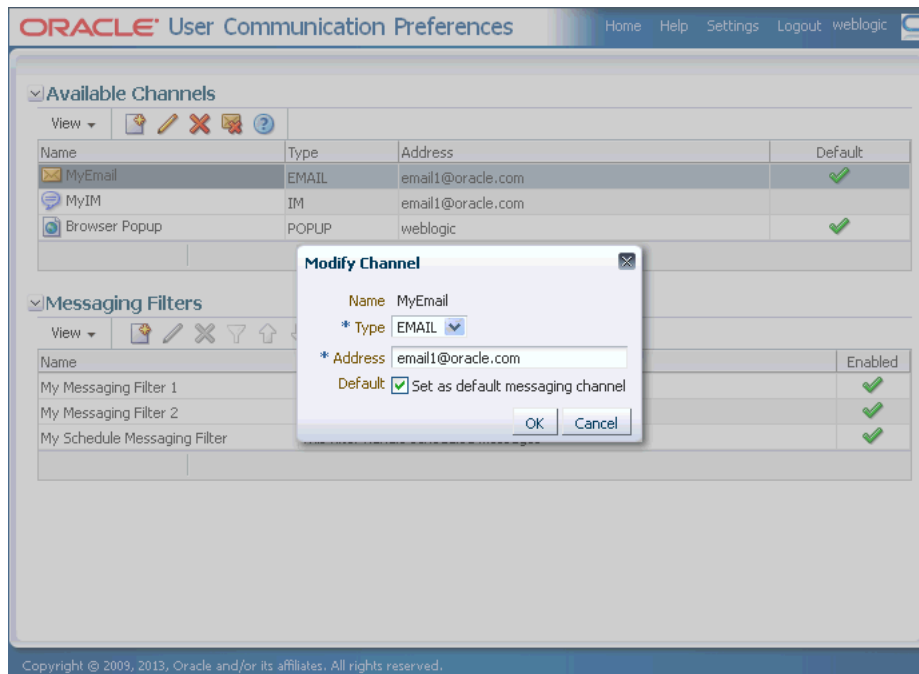
To modify a USER channel, select it from the list of Channels in the *Available Channels* section, and click the **Modify** icon as shown in [Figure 6–3](#).

Figure 6–3 The Modify Icon



The Modify Channel page appears as shown in [Figure 6–4](#). This page enables you to change the channel properties described in [Section 6.2.1.1, "Creating a Channel"](#).

Figure 6–4 Modifying a Channel



6.2.1.3 Deleting a Channel

To delete a USER channel, select the channel from the list of channels in the *Available Channels* section, and click the **Delete** icon as shown in [Figure 6–5](#).

Figure 6–5 The Delete Icon



Certain channels are based on information retrieved from your user profile in the identity store. These are called IDM channels. Users are not allowed to modify or delete such channels through this interface. The only operation that can be performed on such a channel is to make it the default channel. IDM channel addresses are managed through Identity Management System.

6.2.1.4 Setting a Default Channel

You can configure one or more channels as default channels. You can set a channel as default messaging channel either during channel creation or after channel creation directly from the list of channels.

To set a channel as default, select that channel from the list of channels, and click the **default** icon as shown in [Figure 6–6](#). A checkmark appears next to the selected channel, designating it as a default means of receiving messages. Repeat this procedure to add additional default channels, if required.

Figure 6–6 The Default Icon

To remove the default setting of a channel already set as default, select that channel from the list of channels, and click the icon shown in [Figure 6–7](#).

Figure 6–7 The Remove Default Icon

Note: The Business Email channel that is automatically created from the Identity Store attribute is set as a default channel. You cannot remove the default setting of the Business Email channel unless another default channel is set.

For more information about configuring LDAP settings, see *Configuring User Messaging Service Access to LDAP User Profile in Administering Oracle User Messaging Service*.

6.2.2 Managing Filters

A messaging filter defines rules on how to handle incoming messages addressed to a user. The **Messaging Filters** section on the User Communications Preferences page ([Figure 6–8](#)) enables the users to build filters that specify not only the type of messages they want to receive, but also the channel through which to receive these messages.

A filter is composed of two primary sections, condition (or the **If** section) and action (or the **Then** section). For each incoming message, the filters are evaluated to determine the appropriate filter that must be selected for handling the message. The condition section determines the circumstances under which a filter is selected; while the action section specifies how the message is handled if the filter is selected.

Figure 6–8 Messaging Filters

The screenshot shows the Oracle User Communication Preferences interface. At the top, there is a navigation bar with 'Home', 'Help', 'Settings', 'Logout', and 'weblogic'. Below this, there are two main sections:

- Available Channels:** This section contains a table with the following data:

Name	Type	Address	Default
Browser Popup	POPUP	weblogic	<input checked="" type="checkbox"/>
- Messaging Filters:** This section is currently empty, displaying 'No data to display'.

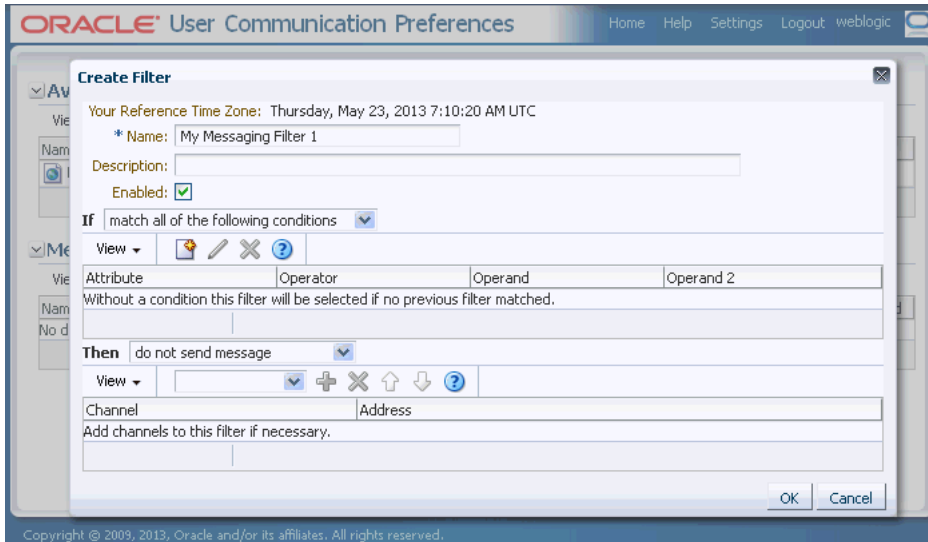
At the bottom of the page, there is a copyright notice: 'Copyright © 2009, 2013, Oracle and/or its affiliates. All rights reserved.'

6.2.2.1 Creating a Filter

To create a filter, perform the following tasks:

1. Click the **Create** icon as shown in [Figure 6-1](#), located in the toolbar under Messaging Filters. The Create Filter page appears as shown in [Figure 6-9](#).

Figure 6-9 *Creating a Filter*



2. Enter a name for the filter in the **Name** field.
3. If needed, enter a description of the filter in the **Description** field.
The checkbox allows you to temporarily enable or disable a filter.
4. Select whether messages must meet all of the conditions or any of the conditions by selecting either the **match all of the following conditions** or **match any of the following conditions** options.
5. Create a filter condition in the **If** section as follows:
 - a. Click the **Create** icon located in the toolbar. The Create Condition dialog box appears.
 - b. Select the message's attribute from the **Attribute** drop-down list that lists the available Business terms. Refer to [Table 6-1](#) for a list of these attributes.
 - c. Combine the selected attribute with one of the comparison operators from the **Operator** drop-down list.
 - d. Enter an appropriate value in the **Operand** field. This is the value that the fact for the selected attribute is used to compare with, using the selected operator.
For instance, if you select the Date attribute, select one of the comparison operators and then select the appropriate dates from the date chooser. If you choose a range operation such as *is between*, then two operand fields will appear for entering lower and upper limit value.
 - e. Click **OK** to add the condition to the table.
6. Repeat the above mentioned steps to add more filter conditions. To delete a filter condition, select the condition from the list of conditions in the table, and click the **Delete** icon as shown in [Figure 6-5](#).

7. When a message is addressed to a user, the **If** section of the user's filters are evaluated against the facts in the message. After a filter is selected in the **If** section, the **Then** section determines how the message will be handled. The **Then** section consists of action type and a list of channels. Select one of the following actions:
 - **do not send message** -- Select this option to block the receipt of any messages that meet the filter conditions.
 - **send to first available channel (Failover in the order)** -- Select this option to send messages matching the filter criteria to a preferred channel (set using the up and down arrows) or to the next available channel.
 - **send to all selected channels** -- Select this option to send messages to every listed channel.
8. If you have selected the action type to send messages, then you must select channels from the drop-down list in the toolbar to add to the action channel list for this filter. After selecting a channel, click **Add** as shown in [Figure 6-10](#). To delete a channel, click **Delete** as shown in [Figure 6-5](#).

Figure 6-10 The Add Icon



9. If needed, use the up and down arrows to prioritize channels. If available, the top-most channel receives messages meeting the filter criteria if you select **send to first available channel**.
10. Click **OK** to create the filter, or **Cancel** to discard the filter.

6.2.2.2 Modifying a Filter

To modify a filter, select the filter from the list of messaging filters, and click **Modify** as shown in [Figure 6-3](#). The Modify Filter page appears. Except the filter name, this page enables you to add or change the filter properties described in [Section 6.2.2.1, "Creating a Filter"](#).

6.2.2.3 Deleting a Filter

To delete a filter, select the filter from the list of messaging filters, and click **Delete** as shown in [Figure 6-5](#).

6.2.2.4 Disabling a Filter

You can temporarily enable or disable a filter. A filter that is disabled will be skipped during the message processing. You can disable or enable a filter in either of the following ways:

- Select the filter from the list of messaging filters. If the selected filter is enabled, the disable filter icon appears in the toolbar as shown in [Figure 6-11](#). Click this icon to disable the filter.

Figure 6-11 The Disable Filter Icon



If the selected filter is disabled, the enable filter icon appears in the toolbar as shown in [Figure 6-12](#). Click this icon to enable the filter.

Figure 6–12 The Enable Filter Icon



- You can also enable or disable a filter from the filter creation or modification dialog box.

6.2.2.5 Organizing Filters

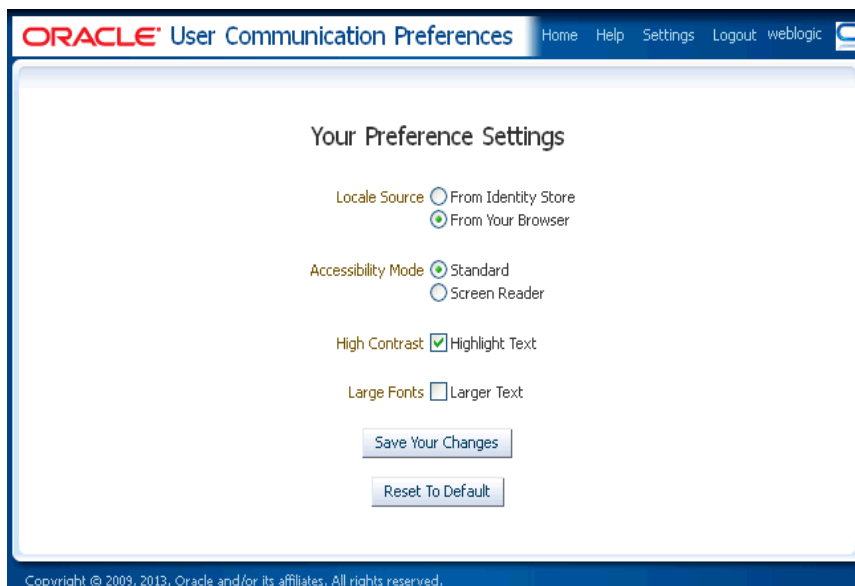
You can prioritize the order of the filters in the list by selecting the filter and moving them up or down in the list, using the up or down arrow icons in the toolbar. During message processing, filters are evaluated in the order from top to bottom in the list until a filter matching the condition is found. If a matching filter is not found, the default channel is used for message delivery with the **send to all selected channels** action type.

6.2.3 Configuring Preference Settings

You can configure your preference settings by accessing the **Settings** tab located in the upper right area of the UCP UI. You can set the following parameters:

- **Locale Source:** Select **From Identity Store** or **From Your Browser**.
- **Accessibility Mode:** Select **Standard** or **Screen Reader**.
- Select the **Highlight Text** checkbox if you want the text to be displayed in high contrast.
- Select the **Larger Text** checkbox if you want the text to be displayed in large fonts.
- Click **Save Your Changes** in order to save the changes you made, or click **Reset to Default** to restore default settings.

Figure 6–13 Configuring Settings



6.3 Administering User Communication Preferences

This section provides information about configuring profiles and managing user data using WebLogic Scripting Tool (WLST). It also provides information about business terms that are used during profile configuration.

6.3.1 About Business Terms

As mentioned earlier in this document, each filter condition is defined against a Business Term. UCP supports the Business Terms as listed in the table below. A business term consists of a name, a data type, and an optional description.

[Table 6–1](#) lists the pre-defined business terms supported by User Communication Preferences.

Table 6–1 Pre-defined Business Terms for User Communication Preferences

Business Term	Data Type
Organization	String
Priority	String
Application	String
Application Type	String
Expiration Date	Date
From	String
To	String
Customer Name	String
Customer Type	String
Status	String
Amount	Number (Decimal)
Due Date	Date
Process Type	String
Expense Type	String
Total Cost	Number (Decimal)
Processing Time	Number (Decimal)
Order Type	String
Service Request Type	String
Group Name	String
Source	String
Classification	String
Duration	Number (Decimal)
User	String
Role	String
Subject	String
Service Name	String
Process Name	String

Table 6–1 (Cont.) Pre-defined Business Terms for User Communication Preferences

Business Term	Data Type
System Code	String
Error Code	String
Occurrence Count	Number (Decimal)

UCP supports two System Terms listed in [Table 6–2](#). System Terms are pre-defined business terms. Administrators cannot extend the system terms. System Terms are available for defining conditions though they are not managed here. The facts for System Terms are automatically obtained based on the current time and user's time zone. Thus, unlike other Business Terms, during message processing, applications do not need to supply facts for System Terms.

Table 6–2 System Terms Supported by User Communication Preferences

System Term	Data Type	Supported Values
Date	Date	Date is accepted as a <code>java.util.Date</code> object or string representing the number of milliseconds.
Time	Time	A 4-digit integer to represent time of the day in HHMM format. First 2-digit is the hour in 24-hour format. Last 2-digit is minutes.

Note: The Server Properties page in the Oracle Enterprise Manager enables you to manage business terms (add or remove business terms). However, do *not* use this page to add or remove business terms as this feature will soon be deprecated. You are allowed to use a subset of business terms during profile configuration as discussed in [Section 6.3.2, "Configuring Profiles by using Oracle Enterprise Manager"](#).

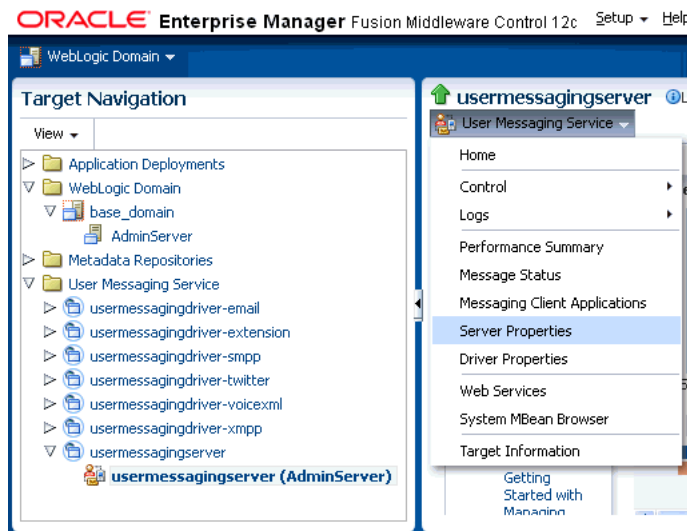
6.3.2 Configuring Profiles by using Oracle Enterprise Manager

Multiple applications may consume a single instance of UCP services. However, not all applications might consume the same set of UCP features. To meet various requirements of different applications, UCP features are virtualized into profiles. This enables each application to target to a specific profile that encapsulates a subset of UCP features.

Each profile is identified by a profile ID. Oracle UCP service provides APIs for a client application to target to a profile, by specifying a profile ID. After upgrading, legacy applications consuming old UCP APIs, without profile ID, will target to a default profile sharing with other applications that targets to the default profile. It is recommended to migrate legacy applications to use latest APIs so that each application can target to a specific profile without sharing with other applications. For more information about UCP API, refer to *User Messaging Service Java API Reference*.

You can manage messaging preferences profiles using Oracle Enterprise Manager Fusion Middleware Control. This interface lists the existing configured profiles, and allows the user to add, modify, or remove profiles. To configure preference profiles, perform the following tasks:

1. Open the User Messaging Server home page in Oracle Enterprise Manager and select **Server Properties** from the drop down menu, as shown in the following figure.



The Server Properties page appears as shown in the following figure. All existing profiles are listed in the **Messaging Preference Profiles** section.

 The screenshot displays the 'Server Properties' configuration page for the 'usermessagingserver' target. The page includes an 'Apply' and 'Revert' button. Under the 'Messaging Preference Profiles' section, there is a table listing existing profiles. The table has columns for ID, Name, Description, and Default.

ID	Name	Description	Default
applcore	ApplCore	This profile should be used for ApplCore applications	
defaultId	Default	This profile is also applied if no id is specified	✓
soa	SOA	This profile should be used for SOA applications	
webcenter	WebCenter	This profile should be used for WebCenter applications	

2. To add a new profile, click the Add icon in the Messaging Preference Profiles section. A dialog box appears as shown in the following figure. You must specify the configuration details to add a new profile.

Click **OK** to save the new profile.

You can define the profile features by selecting a **Locale Source**, availability of **IDM** and/or **User Channels** and a subset of Business Terms. UCP renders the web user interface based on the locale from three different locale sources, namely Client Browser, Identity Store and System Default. The Locale Source field provides a choice from two sources - Locale from Client Browser and, Locale from Identity Store. This determines the locale lookup sequence for the UI rendering. If Locale from Client Browser is selected, then the lookup sequence will be (1) Locale from Client Browser, (2) Locale from Identity Store and (3) System Default Locale. Otherwise, the sequence will be (1) Locale from Identity Store, (2) Locale from Client Browser and (3) System Default Locale. In each sequence, UCP renders the UI using the first supported locale.

In the above figure, the check box for User Channel defines the availability of user defined channels for that particular profile. If the check box is selected, then the users are allowed to create User Channels from the UCP web user interface. The check box for IDM Channel determines the availability of auto-synced channels from Identity Store. Channels are the properties of users. These check boxes determine the visibility of channels for each profile.

Each Profile encapsulates a subset of Business Terms listed under Selected Business Terms as shown in the above figure. In this example, only three terms - **From**, **To**, and **Subject**, are selected. To add more terms, select them from the **Available Business Terms** list on the left and click the right arrow (>) to move them to the Selected Business Terms list. Only selected Business Terms are available for users to define conditions for filters in a particular profile. This means that, on defining filter conditions only the selected Business Terms will be seen in the Attribute drop-down list in UCP web user interface.

3. To modify an existing profile, click the **Edit** icon in the Messaging Preference Profiles section. A dialog box appears as shown in the following figure. You must specify the configuration details to edit the profile.

Click **OK** to save the profile.

4. To remove the messaging preference profile, select the profile from the list of profiles and click **Remove**.
5. Click **Apply** to apply changes in the User Messaging Server instance. Restart the server for the changes to take effect.

6.3.3 Managing User Data using WLST Commands

UCP provides a command line scripting tool, Oracle Weblogic Scripting Tool (WLST), for downloading user preferences data from the UCP repository to the specified XML file, or for uploading user preferences data from an XML file into the UCP repository.

For information about how to use WLST for uploading or downloading user preferences data, see `manageUserCommunicationPrefs` in *Oracle Fusion Middleware Core Components WLST Command Reference*. For information about how to get started with WLST, refer to section *Getting Started Using the Oracle Weblogic Scripting Tool (WLST)* in the *Oracle Fusion Middleware Administering Oracle Fusion Middleware*.

A user may be deleted from the identity store. If a new user is subsequently created with the same name as the old deleted user, then the new user will have access to the User Communication Preferences (UCP) data of the old user of the same name. Such data includes communication channels, message delivery preferences, and messaging filters configured by the old user. To prevent such access by the new user with the same name as the old deleted user, perform the following steps after deleting a user from the Identity Store:

1. Using the following WLST command, download the UCP data for all users into an XML file, for example *userdata.xml*:

```
wls:/offline> manageUserCommunicationPrefs(operation='download',
filename='userdata.xml', url='t3://<hostname>:<portnumber>',
username='<adminusername>', password='<adminpassword>')
```

2. Make a backup copy of the XML file. Edit the *userdata.xml* file to delete all data for the old deleted user. Each user data section is organized in the `<user guid=username>` element in the XML file. Find the user element with *guid* for the old user, and delete all data of the user from the user element. The following example shows how the user element should appear after deleting all data of a user, for example *david*:

```
<user guid="david">
```

```
<devices>
</devices>
</user>
```

3. Save the *userdata.xml* file and upload the modified file using the following WLST command:

```
wls:/offline> manageUserCommunicationPrefs(operation='upload',
filename='userdata.xml', merge='overwrite', url='t3://<hostname>:<portnumber>',
username='<adminusername>', password='<adminpassword>')
```

6.4 Integrating UCP Web User Interface

With UCP services, an end user can manage his communication preferences from a web interface by managing his channels and filters. To ensure that the end users are provided control over their communication preferences, the application developers must ensure the integration of UCP web user interface to their application. UCP provides an ADF task flow library which makes it easy to add a region in your web application so that end users can access their preference directly from your application.

This section discusses how to implement the web user interface using an ADF task flow library and targeting a specific profile. The ADF task flow library makes it easy to integrate the web interface with a new application or an existing application. For more information about ADF task flows, refer to *Creating ADF Task Flows in Developing Fusion Web Applications with Oracle Application Development Framework*. To integrate your UCP instance with web user interface, perform the tasks in this section.

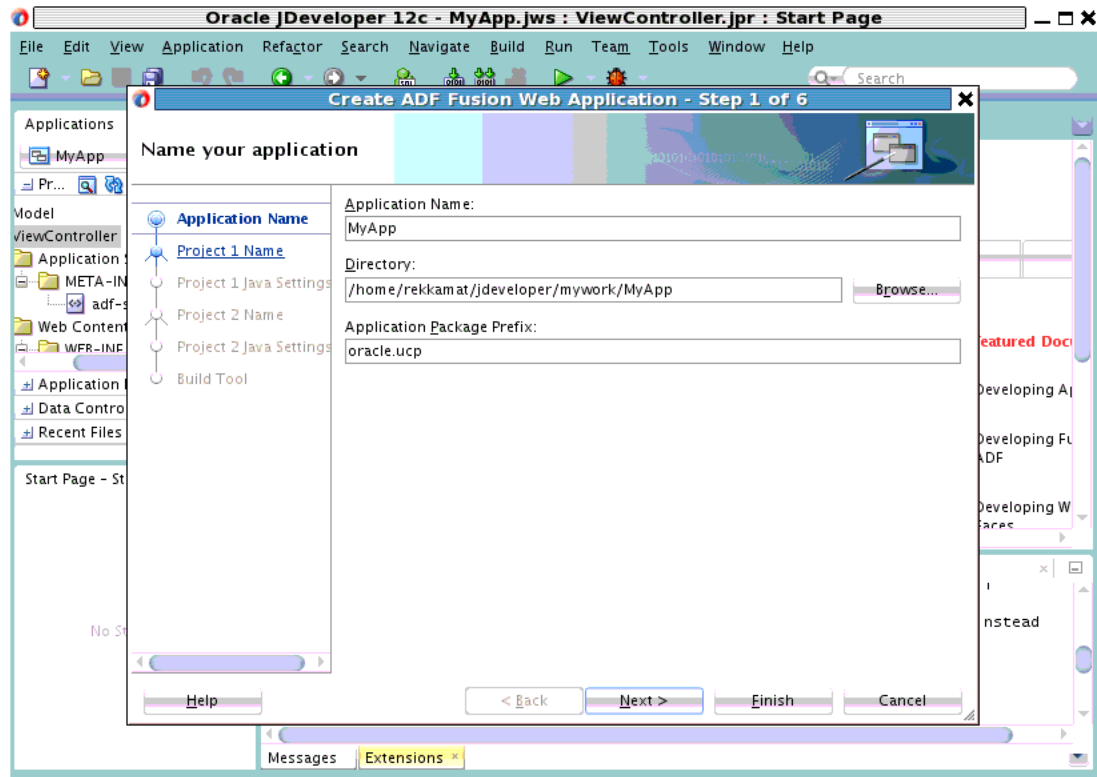
6.4.1 Integrate ADF Web Application with UCP

This section describes how to create an ADF web application, and how to add a page or a region to an existing web application using the ADF task flow library in Oracle JDeveloper.

6.4.1.1 Create a New ADF Application

You can integrate UCP with a new application or an existing application. To create a new ADF web application in Oracle JDeveloper, perform the following tasks:

1. Open the **Oracle JDeveloper 12c** wizard. Select **New Application** from the Application Navigator on the left pane. A New Gallery window appears.
2. In the New Gallery window, select **ADF Fusion Web Application** from the list of items, and click **OK**. The **Create ADF Fusion Web Application** window appears as shown in the following figure.

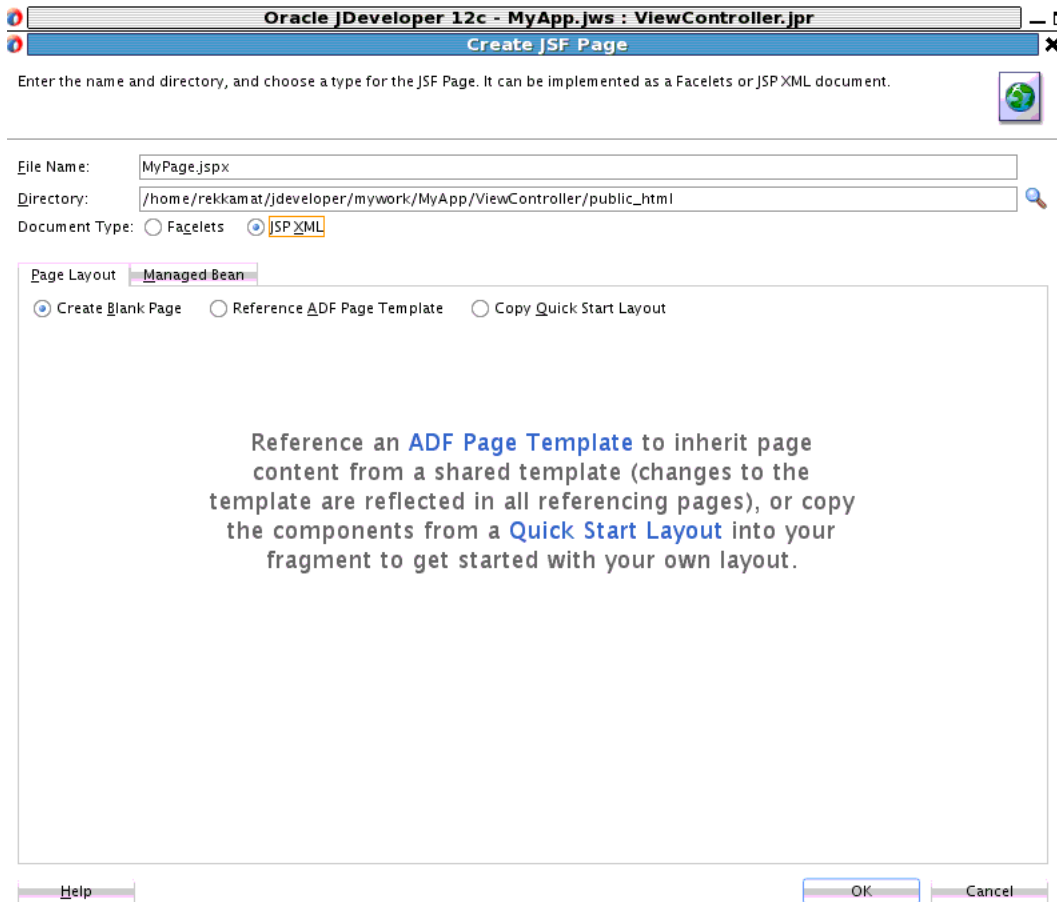


3. In this window, enter an application name, for example, *MyApp*. Also, enter the application package prefix, for example, *oracle.ucp*. Click **Finish** to create a new ADF application.

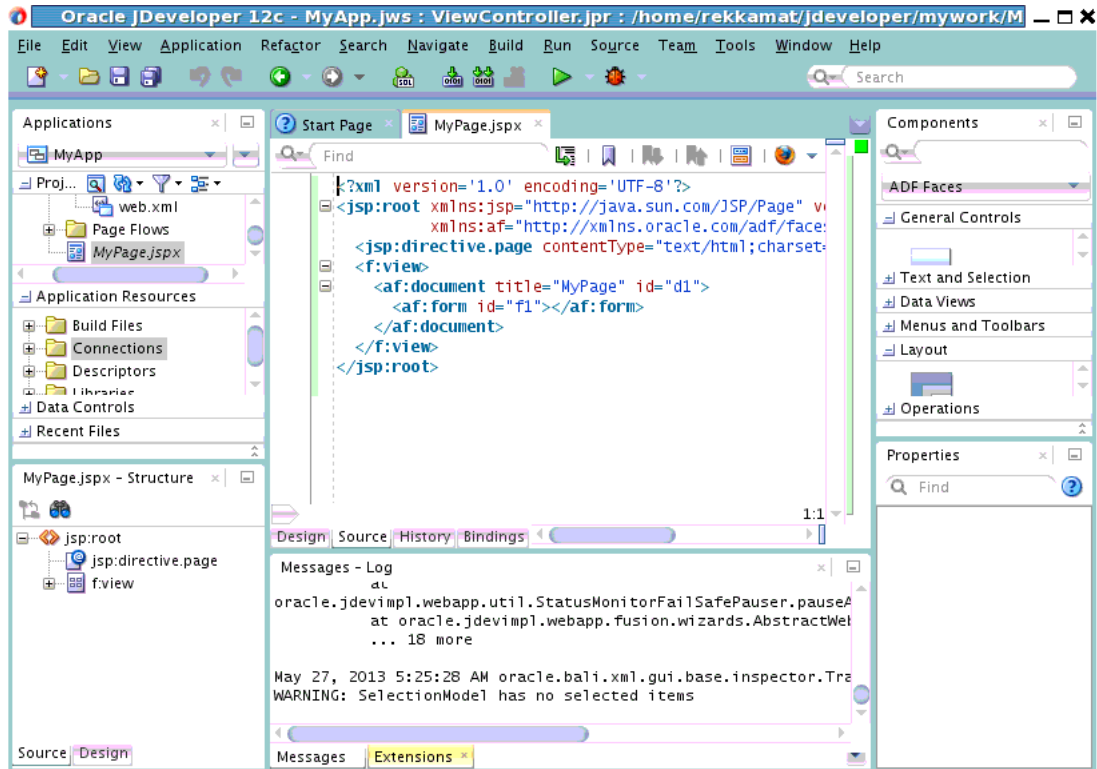
6.4.1.2 Create an ADF Web Page

The bounded task flow in the UCP library can be added in any region to provide seamless integration to your application. In this section, we will create a demo ADF web page to host a region for the UCP task flow. To create a new ADF web page, perform the following tasks:

1. In the left window pane, right-click **ViewController** to display the context menu.
2. In the ViewController context menu, from the **New** menu, select **Page**. The **Create JSF Page** window appears as shown in the following figure.



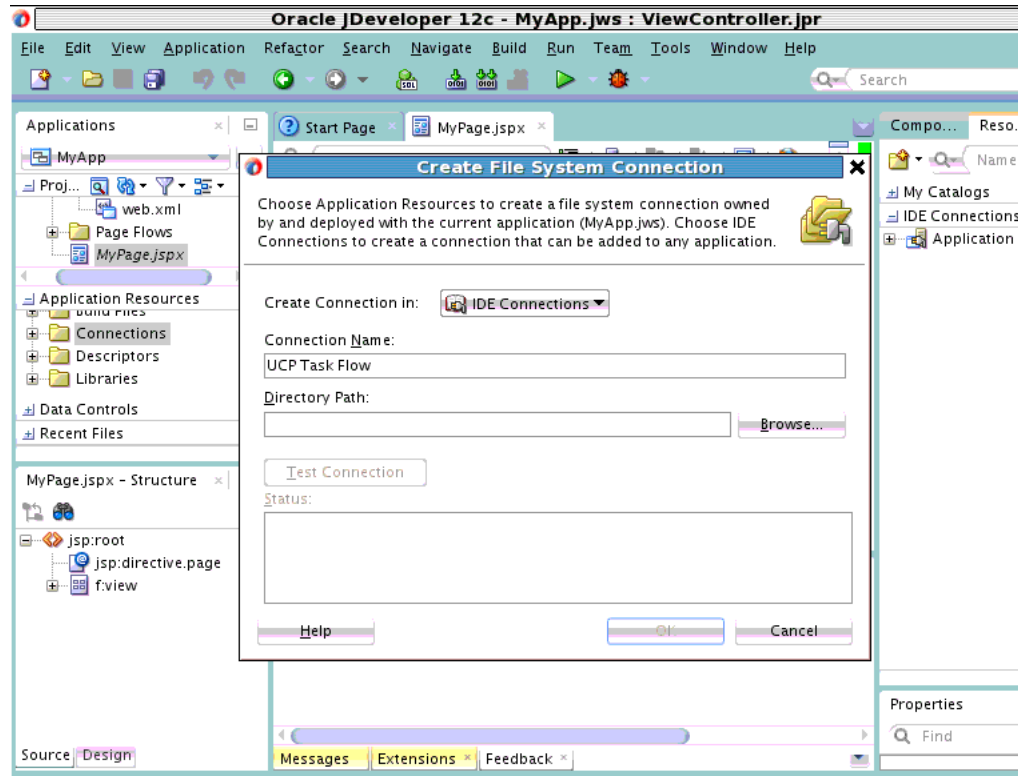
3. In this wizard, enter the file name, for example, *MyPage.jspx*. From the Document Type menu, select **JSP XML**.
4. From the Page Layout menu, select **Create Blank Page**. Click **OK**. The new *MyPage.jspx* is created.
5. Open the newly created page by double-clicking **MyPage.jspx** under the *ViewController* node in the navigation pane. In the *MyPage.jspx* window, select **Source** from toolbar at the bottom to view the source code of the page as shown in the following figure.



6.4.1.3 Connect UCP Task Flow Library

If you have deployed UMS to Weblogic Server, then the UCP task flow library is already deployed to the server. However, the UCP task flow library may not be accessible from your JDeveloper instance. Therefore, it is important to connect the task flow library to your application. You must first create a file system connection from your JDeveloper instance to the library. To achieve this, perform the following steps:

1. Click the **Window** tab located in the toolbar on the top pane to display a drop-down list. From the list, select **Resources** to add the Resources tab to the right pane of the JDeveloper window.
2. Under the Resources tab, click the folder icon to display a drop-down list. In the list, navigate to **IDE Connections**, and select **File System**. The Create File System Connection wizard appears as shown in the following figure.

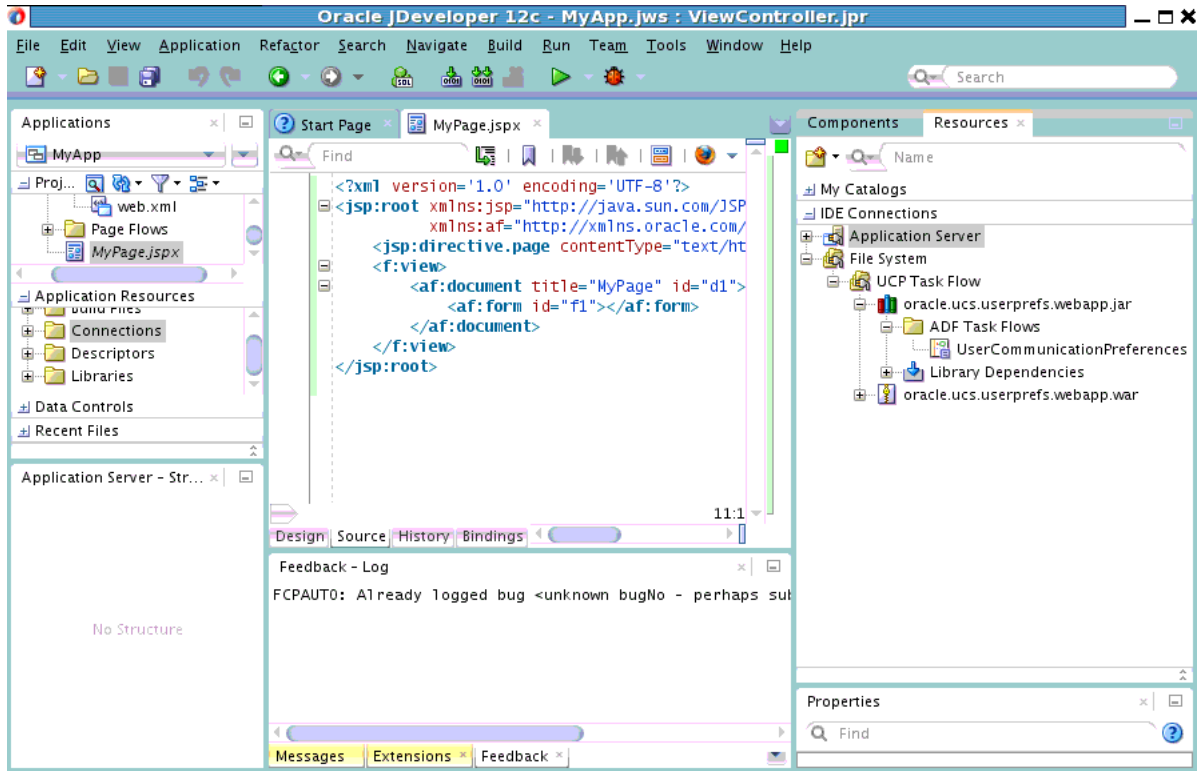


3. Enter the connection name, for example, UCP Task Flow.
4. Click **Browse** to select the folder that contains the UCP task flow library. This folder is located in the `oracle_common` directory and the path would look like the following:


```
oracle_common/communications/modules/oracle.ucs.userprefs.webapp_xx.x.x
```

 where `xx.x.x` is the version number, for example, 12.1.2.

Navigate to this folder and click **Select**.
5. Click **Ok**. The File System Connection is created. To verify, expand the **File System** navigation tree in the right pane. On expanding the **ADF Task Flows** folder, you will see `UserCommunicationPreferences` as shown in the following figure.

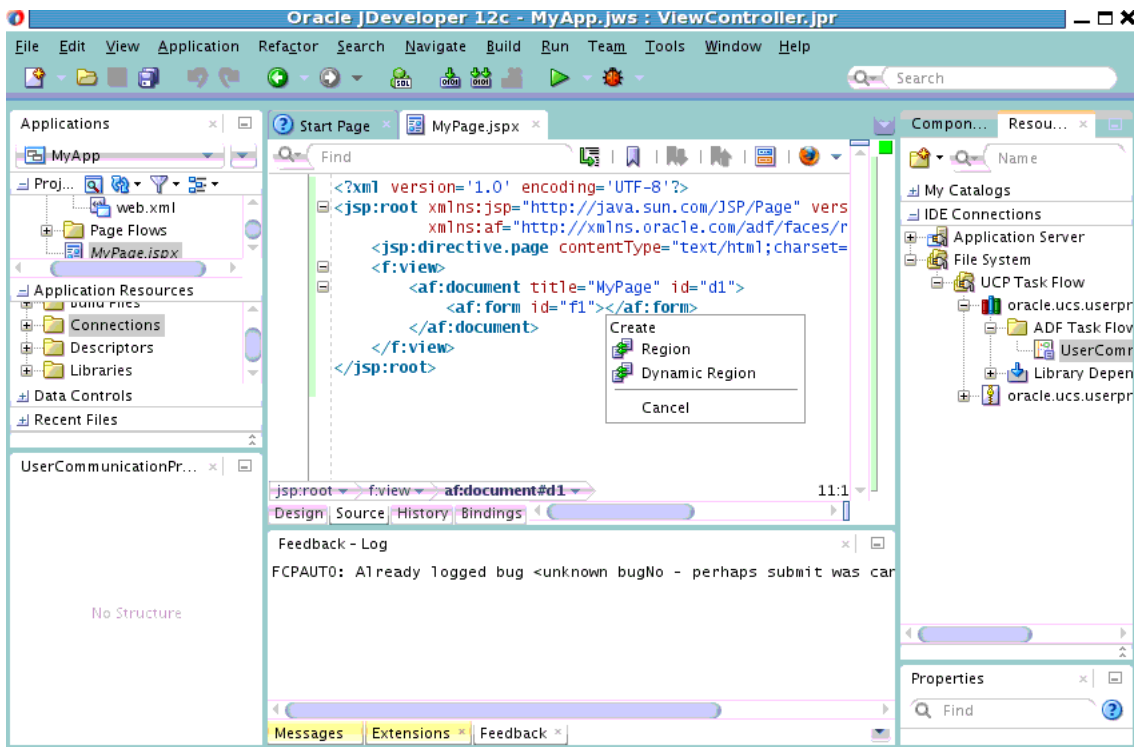


6.4.1.4 Add a Region in the New Page

With the support of an ADF task flow library, you can add a region in your web application so that the end users can access their preference directly from your application. To achieve this, perform the following tasks:

1. Click the **Source** tab of the newly created page, `MyPage.jspx`.
2. From the File System navigation tree in the right pane, drag the **UserCommunicationPreferences** task flow and drop it inside the `<af:form>` `</af:form>` tag in the page source to create a region in this tag.

A context menu appears as shown in the following figure.



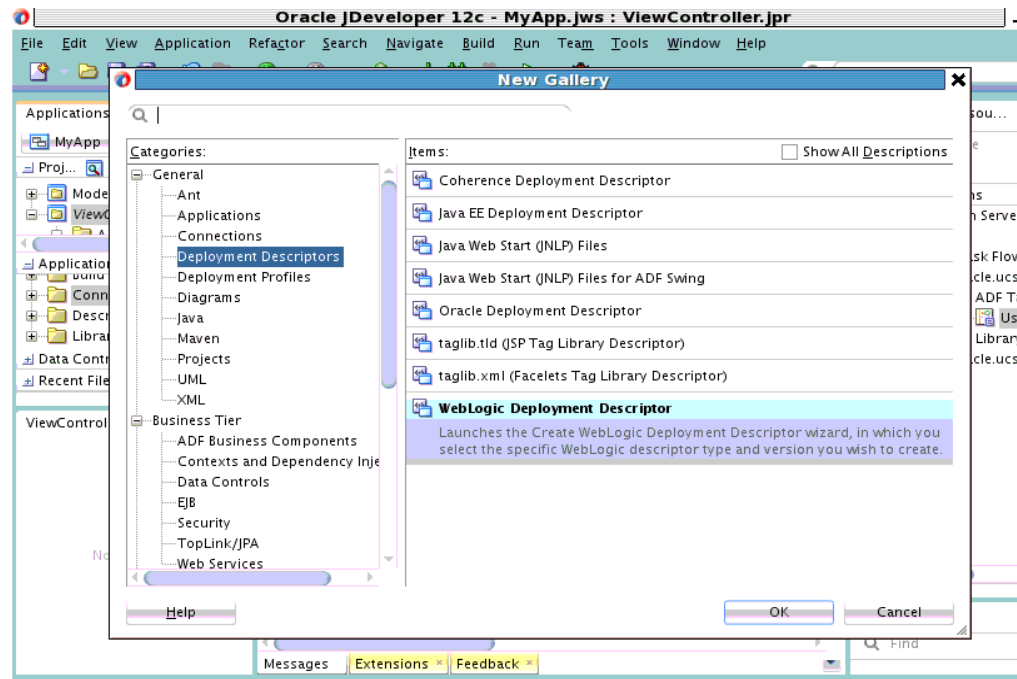
3. Select **Region** from the context menu. A confirmation window appears.
4. Click **Add Library**. The Edit Task Flow Binding wizard appears.
5. In the Input Parameters table, enter a value for `profileId`, for example, `soa`. The value for the Profile ID determines that the created user interface accesses only user filters and channels in this profile. This should be the same profile that your application is targeting. Click **OK**. A region has been added to the page.

If you have an existing ADF web application that you want to integrate with UCP by including a user preference page, then you can add a region to your existing or new ADF page, in the same manner as mentioned in this section.

6.4.1.5 Reference UCP Libraries

Since UMS and UCP task flow libraries are deployed in WebLogic server, you do not need these libraries in your application. You must reference these libraries from the `weblogic.xml` file. To achieve this, perform the following tasks:

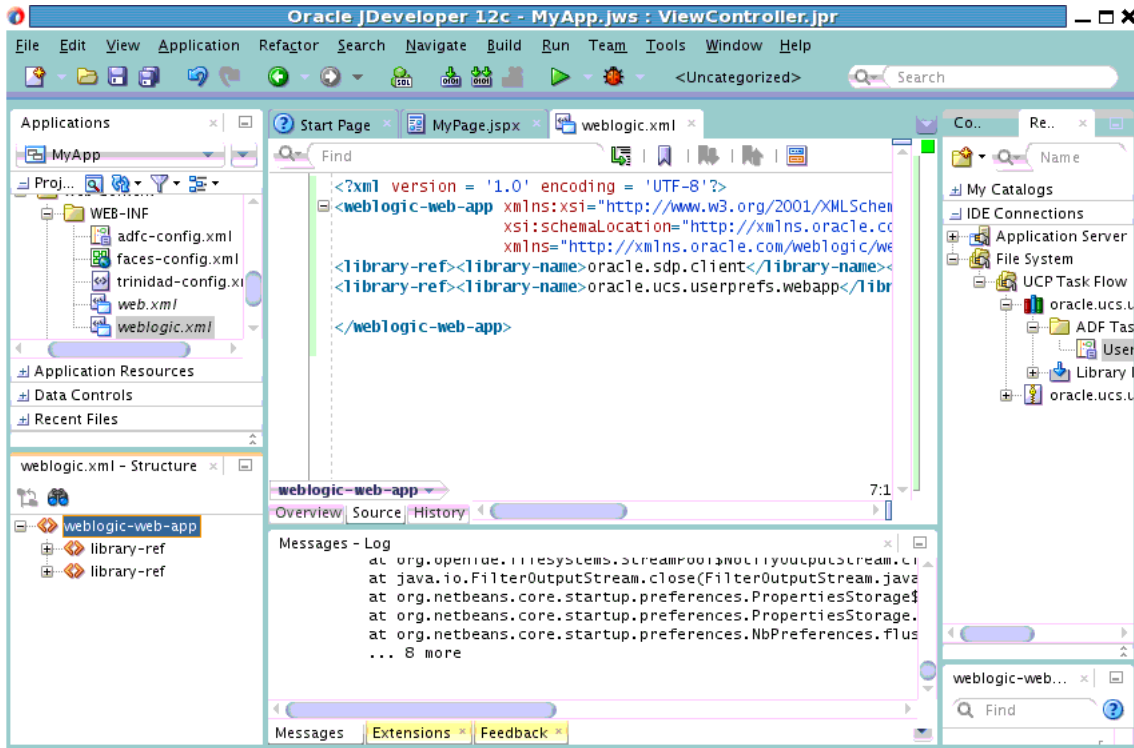
1. Right-click the **ViewController** node located in the navigation pane. A context menu appears.
2. From the context menu, select **New**, and click **From Gallery**. The New Gallery wizard appears.
3. In the navigation pane, expand the **General** node and select **Deployment Descriptors**. From the list of items on the right pane, select **Weblogic Deployment Descriptor**. Click **OK**. See the following figure.



The Create WebLogic Deployment Descriptor wizard appears.

4. From the list of deployment descriptors, select **weblogic.xml**, and click **Next**. In the next screen, select **12.1.1** or the newer version as the deployment descriptor version, and click **Next**. In the Summary page, click **Finish**
5. Open the newly created page by double-clicking **weblogic.xml** under the ViewController node in the navigation pane. In the weblogic.xml window, select **Source** from toolbar at the bottom to view the source code of the page.
6. In the weblogic.xml file, you will reference two libraries, that is, the UMS library (oracle.sdp.client) and the UCP library (oracle.ucs.userprefs.webapp). Add the following library references as shown in the following figure:

```
<library-ref><library-name>oracle.sdp.client</library-name></library-ref>
<library-ref><library-name>oracle.ucs.userprefs.webapp</library-name></library-ref>
```



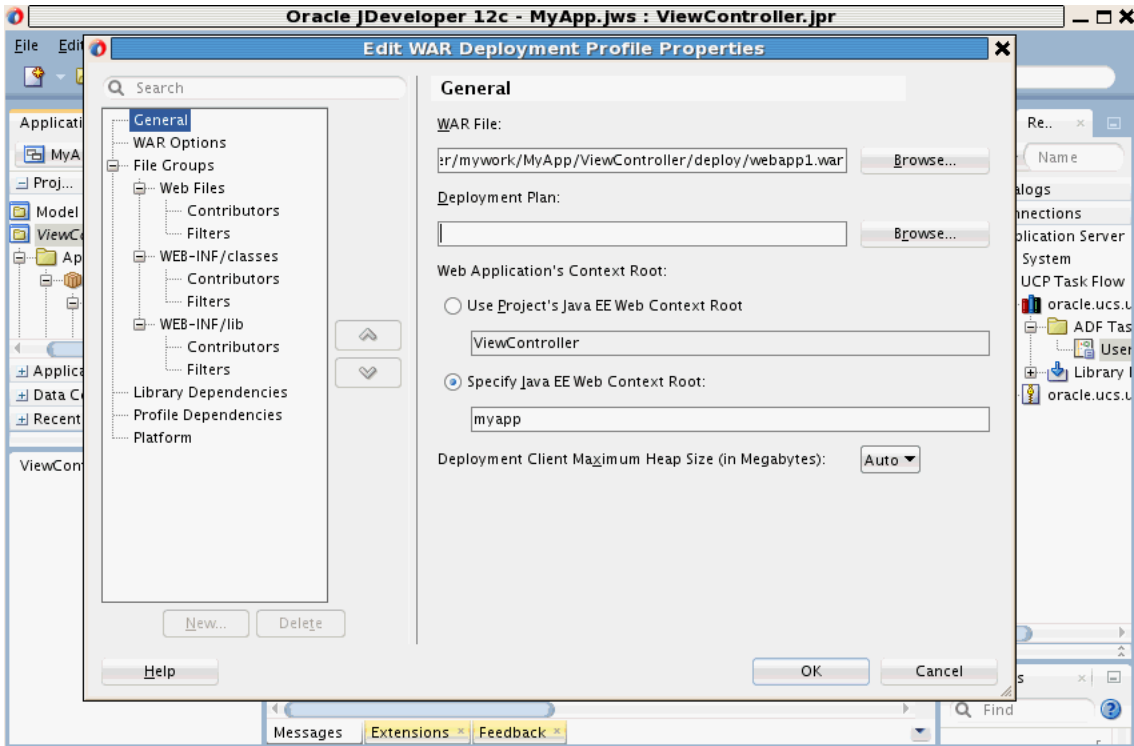
6.4.1.6 Manage Project Deployment Profile

Since you have already referenced the two libraries in the `weblogic.xml` file, you do not need to package these libraries in your application. This can be managed through the project deployment profile. To achieve this, perform the following tasks:

1. You must first create a project deployment profile. Right-click the **ViewController** node from the navigation pane. A context menu appears.
2. From the context menu, click **Deploy**, and select **New Deployment Profile**. The Create Deployment Profile wizard appears.
3. From the Profile Type drop-down list, select **WAR File**, and click **OK**.

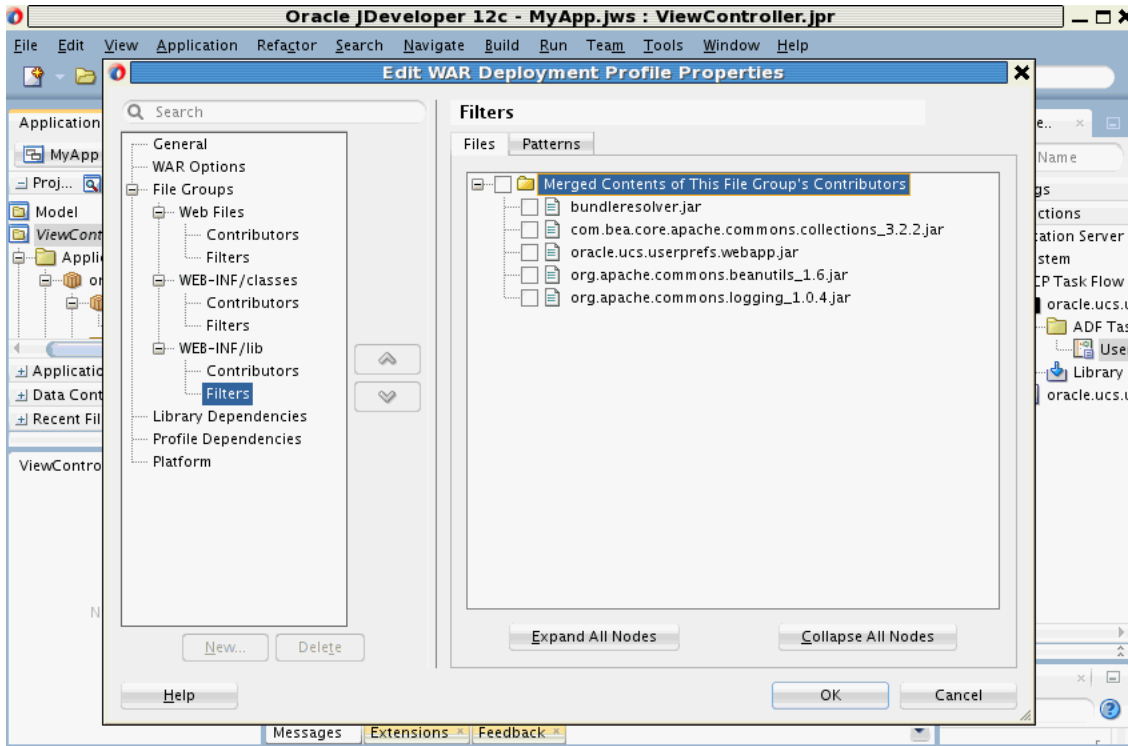
The project deployment profile has been created. You must, now, ensure that the libraries are not packaged in your application.

4. Under the General tab, select the **Specify Java EE Web Context Root** option, and enter your application context root, for example, `myapp` as shown in the following figure.



- From the navigation tree, select **Filters** listed under **WEB-INF/lib**. Deselect all the `.jar` files listed in the right pane as shown in the following figure. Click **OK** to save changes to your deployment profile.

This ensures that the libraries are not packaged in your application, as you have already referenced the important libraries in the `weblogic.xml` file in the previous section.



6.4.1.7 Create Application Deployment Profile

To create a new application deployment profile, perform the following tasks:

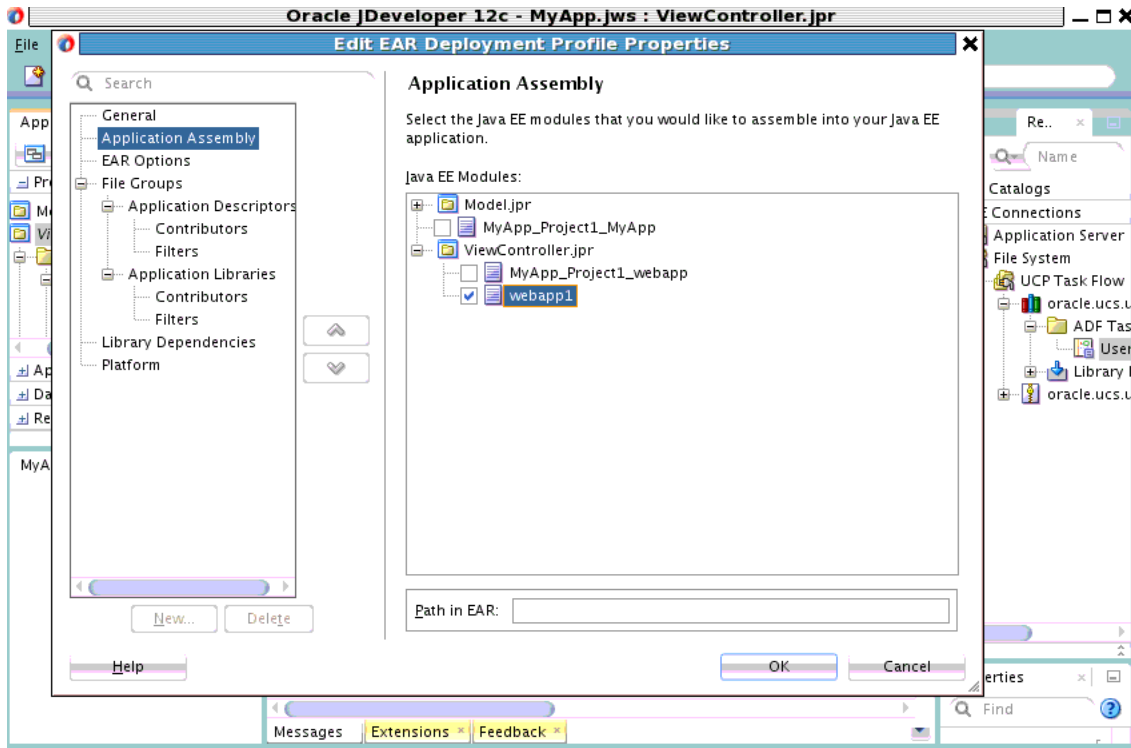
1. From the **Applications** drop-down list in the left pane, select **Deploy** and click **New Deployment Profile** from the context menu.

The Create Deployment Profile wizard appears.

2. From the Profile Type drop-down list, select **EAR File**, and enter a name for your application deployment profile, for example, *MyApp*.

Click **OK** to create the application deployment profile. The Edit EAR Deployment Profile Properties wizard appears.

3. From the navigation tree, select **Application Assembly**.
4. In the right pane, expand the **ViewController.jpr** node and select **webapp1** to include it in the application as shown in the following figure.



5. In the toolbar, click **Save All** to save your application.

6.4.2 Deploy Your Application

You must, now, deploy your application to the WebLogic application server. If your application server has not been configured already, then you must first configure your application server connection.

6.4.2.1 Deploy Application

To deploy your application, perform the following tasks:

1. From the **Applications** drop-down list, select **Deploy** and click **MyApp** from the context menu. The Deploy MyApp wizard appears.
2. In the Deployment Action screen, select **Deploy to Application Server** and click **Next**.
3. In the Select Server screen, select the server from the list of Application servers, and click **Finish** to deploy your application.

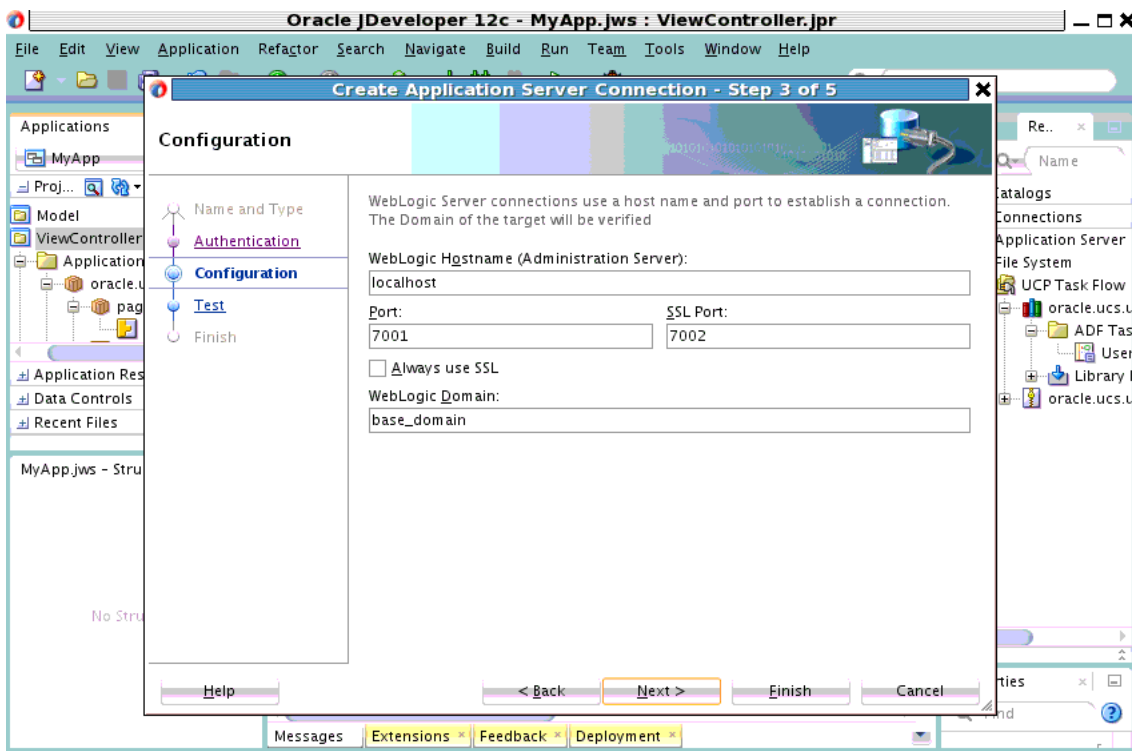
If the server is not in the list of application servers, then follow instructions in [Section 6.4.2.2, "Configure Application Server Connection"](#) to configure your server connection.

6.4.2.2 Configure Application Server Connection

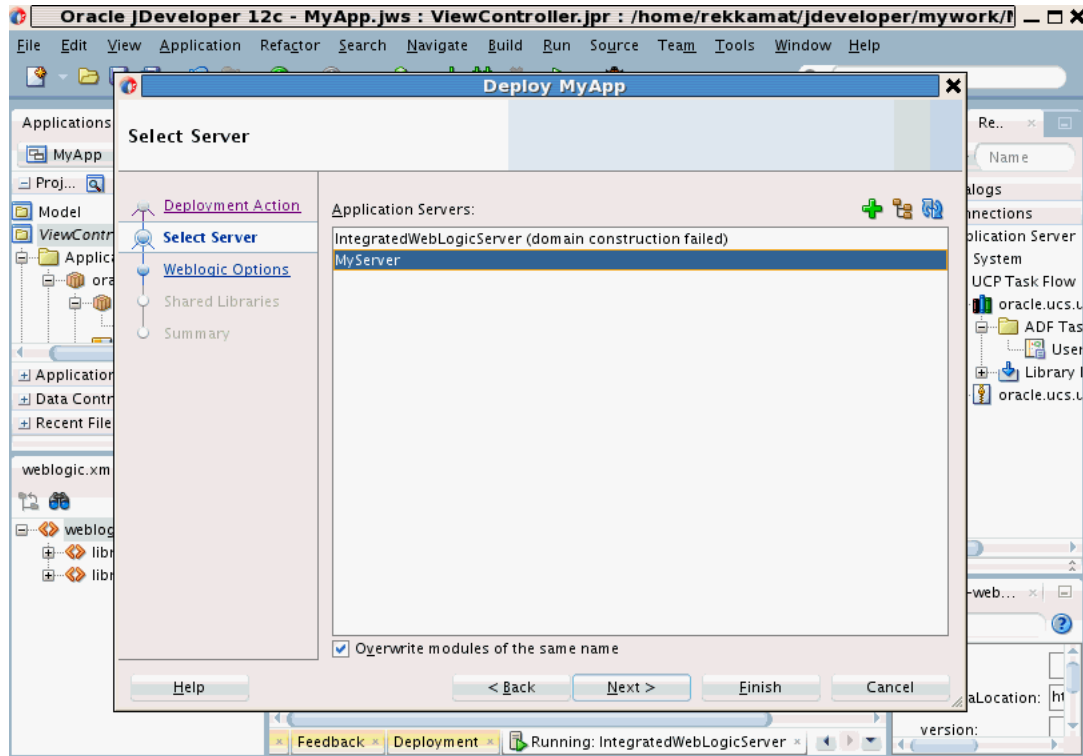
If you have not configured your application server connection, then you must first configure it by performing the following tasks:

1. From the **Applications** drop-down list, select **Deploy** and click **MyApp** from the context menu. The Deploy MyApp wizard appears.

2. In the Deployment Action screen, select **Deploy to Application Server** and click **Next**.
3. In the Select Server screen, click the plus icon located at the top right corner.
The Create Application Server Connection wizard appears.
4. In the **Connection Name** field, enter your server connection name, for example, *MyServer*. Click **Next**.
5. In the Authentication screen, enter your application server's admin **Username** and **Password**. Click **Next**.
6. In the Configuration screen, specify the host name, port numbers and, domain name as shown in the following figure.
Click **Next**.



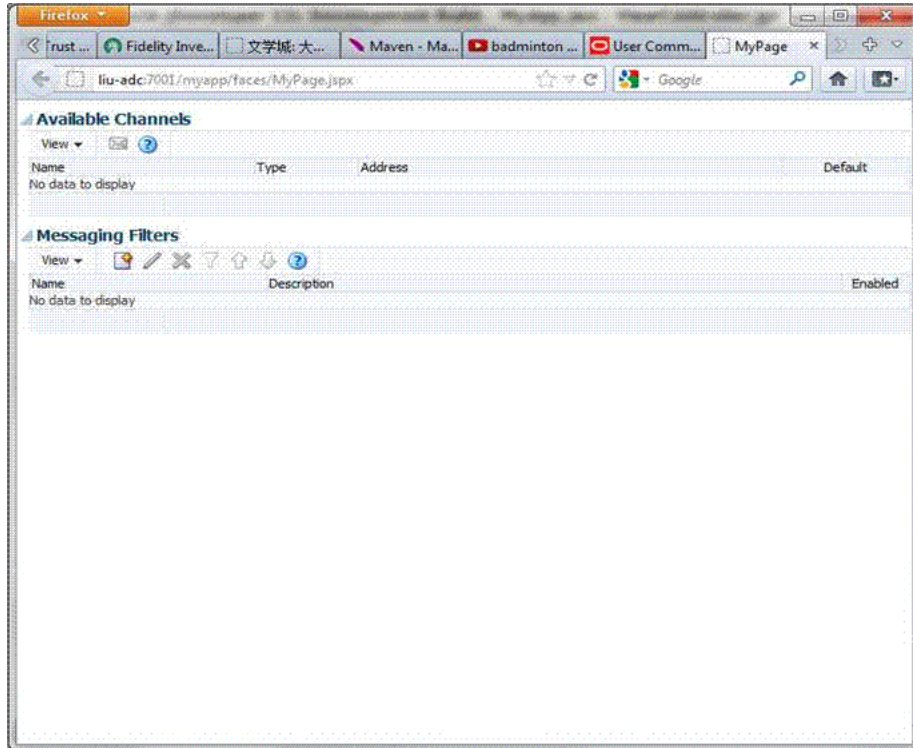
7. On the Test screen, verify your connection by clicking **Test Connection**. If the test is successful, then click **Finish**. You should be able to see your application server in the list of application servers as shown in the following figure.



8. To deploy your application to the newly configured application server, select the server from the list and click **Finish**. You will receive a confirmation for the successful deployment of your application.

6.4.3 Verify Your Application

You can access your application to manage user preferences. To verify your application, open a browser and point the URL to your newly created page, for example, <http://localhost:7001/myapp/faces/MyPage.jspx>. You should see your communication preference web interface as shown in the following figure.



Note: When you access your application for the first time on your browser, you might see a blank page. This is expected behavior if your application does not include authentication. For the simplicity of this demo, we have not included authentication. But it is recommended that you protect your web application with proper authentication.

To test your demo application without authentication, you can try the following workaround. If there is another Oracle SSO protected web application, you can first log in to that application, and then access your application from the same browser. As a result, you will see the channels and filters of the authenticated user, instead of seeing a blank page.

6.5 Java Application Interface

UCP services can be consumed directly or indirectly from your application. To consume UCP services indirectly, you must call UMS API that, in turn, will send messages to the users. UMS will deliver messages to the best channels according to the user's preferences. If you want to consume UCP services directly, then you must call UCP Java API. For instance, if you want to control the message format depending on the delivery type of the channel that a user prefers, then you are required to consume the UCP services directly. Or for instance, if you want to build a new application to send messages to the users of your application on certain events, then you can programmatically create a default email channel for each user using the UCP API to ensure that each user receives the message initially upon the deployment of your application. A user may change his preferences later if he does not want the email as his default channel.

UCP Java APIs provide program interface for implementing applications to consume UCP services. UCP also provides Java APIs for managing channels and filters. The

Java interface, `oracle.ucs.userprefs.UserCommunicationPreference` is the primary interface for clients to access UCP services. The `oracle.ucs.userprefs` package contains the User Communication Preferences API classes. For more information, refer to *User Messaging Service Java API Reference*. This is the root interface for managing user's preference objects such as addresses, filtersets, etc. For more information about APIs and interfaces for UCP services, refer to *User Messaging Service Java API Reference*.

6.5.1 Obtain Delivery Preferences

A user's delivery preferences can be obtained by invoking the Java method `getDeliveryPreference(String guid, String profileID, Map<String, Object> facts)`, where `guid` is case-sensitive. The `profileID` parameter, introduced in 12c, is used to target user preferences for a specific profile. It is recommended that each application targets a specific profile. There is a default profile ID that can be acquired by applications by calling the `getDefaultProfileId` method. For applications that invoke the 11g APIs (without a Profile ID), the default profile is automatically used. Finally, applications calling this API must provide all the necessary facts so that UCP can select a matched filter in the target profile. The facts are in a `Map<String, Object>` of name/value pair where the name is the business term and the key in the map. For more information about using this API, refer to *User Messaging Service Java API Reference*.

The following code snippet demonstrates how to obtain the delivery preferences for `testUser1`:

```
// Obtain a UserCommunicationPreference object
UserCommunicationPreference ucp =
UserPrefsServicesFactory.createUserCommunicationPreference();

// Set target user ID
String userId = "testUser1";

// Specify the Profile ID, or default id from ucp.getDefaultProfileId()
String profileId = "soa";

// Add all facts into Hashtable facts. Facts for Date and Time are not needed.
Map <String, String> facts = new HashMap<String, Object>();
facts.put("Application", "BPEL"); // Add application name
facts.put("Due Date", new Date()); // Use current date
facts.put("Amount", new Double("678.00")); // Set number for 678

// Invoke getDeliveryPreference() function with userId, profileId and facts.
DeliveryPreference dp = ucp.getDeliveryPreference(userId, profileId, facts);

// Retrieve Action Type and delivery Channels from the returned DeliveryPreference
object.
ActionType at = dp.getActionType();//Get Action Type
Vector <DeviceAddress> channels = dp.getDevices();//Get delivery Channels
```

6.5.2 Manage Channels

UCP provides an automatic channel management feature to sync UCP repository with the Identity Store (usually, LDAP). When an email address is added to the Identity Store, an EMAIL channel corresponding to that email address is automatically created in the UCP repository. This channel is removed from the repository when the corresponding email address is deleted from the Identity Store. UCP automatically

creates channels for email, IM, business, home and mobile phones. These channels are called IDM Channels.

Since these channels are automatically managed by UCP, applications must not attempt to create or remove them. Applications are allowed only to set or unset these channels as default channels. However, applications can create, modify, or remove USER channels using APIs such as, `createDeviceAddress`, `getDeviceAddresses`, etc. A channel can be flagged as a default channel using the `setDefaultChannel` API.

The following code snippet demonstrates how to create a communication channel for a user, *testUser1* with a particular email address:

```
// Create an email channel for testUser1
DeviceAddress channel = ucp.createDeviceAddress("testUser1", // User ID
"myEmail", // Channel name
DeliveryType.EMAIL, // Delivery Type for email
"myemail@somecompany.com"); // Email address
ucp.save(channel); // without this line, the Channel will not be persisted in UCP
repository
```

The channel name must be unique for each user. The combination of Delivery Type and Delivery Address must also be unique for each user. Following are some sample code snippets that demonstrate how to manage a channel:

```
// Set the Channel as a default Channel
ucp.setDefaultAddress("testUser1", // User ID
"soa", // Profile ID
channel); // Channel to be flagged

// Unset a default Channel
ucp.removeDefaultAddress("testUser1", // User ID
"soa", // Profile ID
channel); // Channel to be unset

// Modify the Channel's address
channel.setAddress(newemail@somecompany.com);
ucp.save(channel); // without this line, the change will not be persisted in UCP
repository

// Remove the Channel
ucp.delete(channel);
```

6.5.3 Manage Filters

Filters are managed in a set for each profile. The following code snippet demonstrates how to create a messaging filter for a user, *testUser1* for the *soa* profile.

```
// Get, or create if not exist, user's Filter Set for Profile "soa"
FilterSet filterSet = ucp.getFilterSet("testUser1" // User ID
"soa"); // Profile ID

// Create a new Filter
Filter filter = filterSet.createFilter("Test Email Filter"); // Create a new
Filter named "Test Email Filter"
filter.setConditionType(ConditionType.MATCH_ANY); // Set the Condition Type to
logical OR

// Create a new Condition
Condition condition = filter.createCondition(); // Create a new Condition first
Map<String, BusinessRuleTerm> terms = ucp.getBusinessTerms();
```

```
BusinessRuleTerm term = terms.get("Subject");// Business Term for "Subject"
condition.setFilterTerm(term);
condition.setTermOperation(TermOperationType.Contains);
condition.setOperandOne("approved");// Set value "approved"
ArrayList<Condition> conditions = new ArrayList<Condition>();
conditions.add(condition);
filter.setConditions(conditions);// Add the Condition list to the Filter

// Set Action Type
Filter.setActionType(ActionType.SERIAL);// Set Action Type for SERIAL

// Get all the Channels for "soa" Profile
Set<DeviceAddress> allAddresses = ucp.getDeviceAddress("testUser1",// User ID
    "soa");// Profile ID
ArrayList<DeviceAddress> channels = new ArrayList(allAddresses);// Convert to a
List
filter.setDeviceAddressList(channels);// Add to the Filter as target Channels

// Add the Filter to the Filter Set
filterSet.addFilter(filter);

// Finally persist the FilterSet object
ucp.save(filterSet); // Required to persist the Filter
```

To deploy your application, you must reference the shared library `oracle.sdp.client` from your application's development descriptor. The application must be deployed in the same domain with the UCP service.

Though UCP provides Java APIs for applications to manage channels and filters, it is recommended that users manage their preferences through UCP web interface integrated in their web application using the UCP task flow library.

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.

This appendix includes the following sections:

- [Section A.1, "Using the UMS Client API to Build a Client Application"](#)
- [Section A.2, "Using the UMS Client API to Build a Client Echo Application"](#)
- [Section A.3, "Creating a New Application Server Connection"](#)

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>.

A.1 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.

A.1.1 Overview of Development

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

1. [Section A.1.2, "Configuring the Email Driver"](#)
2. [Section A.1.3, "Using JDeveloper 12c to Build the Application"](#)
3. [Section A.1.4, "Deploying the Application"](#)
4. [Section A.1.5, "Testing the Application"](#)

A.1.2 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*.

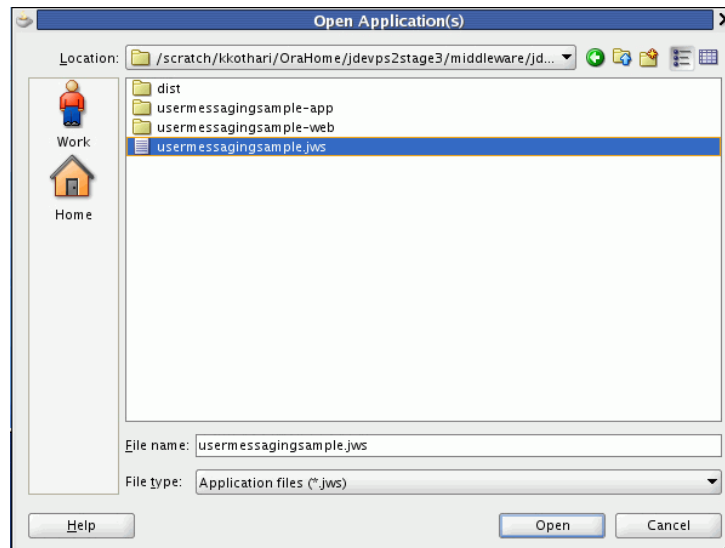
Note: This sample application is generic and can support outbound messaging through other channels when the appropriate messaging drivers are deployed and configured.

A.1.3 Using JDeveloper 12c to Build the Application

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

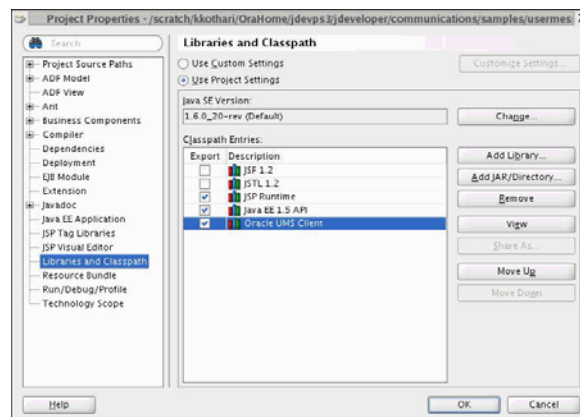
A.1.3.1 Opening the Project

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

Figure A-1 Oracle JDeveloper Open Application Window

In the Oracle JDeveloper main window, the project appears.

2. Satisfy the build dependencies for the sample application by ensuring the "Oracle UMS Client" library is used by the web module.
 1. In the Application Navigator, right-click web module **usermessagingsample-web**, and select **Project Properties**.
 2. In the left pane, select **Libraries and Classpath**.

Figure A-2 Verifying Libraries

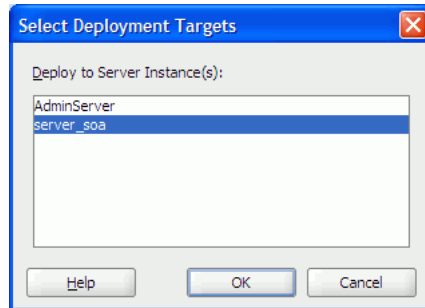
3. Click **OK**.
3. Explore the Java files under the **usermessagingsample-web** project to see how the messaging client APIs are used to send messages, get statuses, and synchronously receive messages. The MessagingClient instance is created in `SampleUtils.java` in the project.

A.1.4 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 [Section A.3, "Creating a New Application Server Connection."](#)
2. Deploy the application by selecting the **usermessagingsample** application, **Deploy**, **usermessagingsample**, **to**, and **SOA_server** ([Figure A-3](#)).

Figure A-3 Deploying the Project



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.

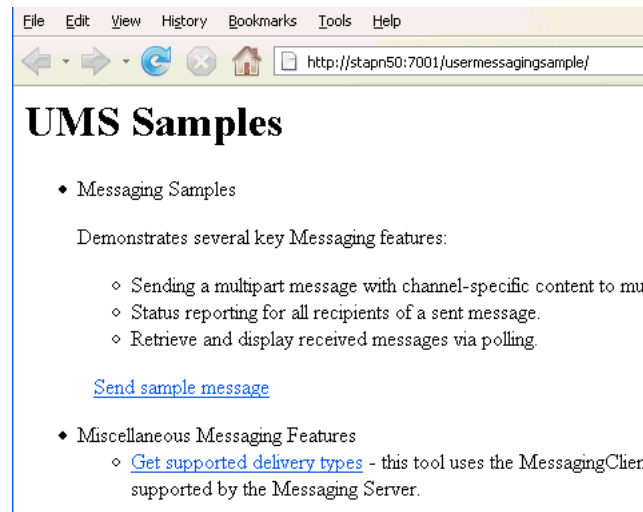
Note: Refer to *Administering Oracle User Messaging Service* for more information.

A.1.5 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 ([Figure A-4](#)).

Figure A-4 Testing the Sample Application

2. Click **Send sample message**. The Send Message page appears (Figure A-5).

Figure A-5 Addressing the Test Message

The screenshot shows the 'UMS Sample: Send Message' form with the following fields and values:

- Enter Sender Addresses (optional): [e.g. "IM:sender@example.com"]
Separate multiple addresses using comma.
Note: If you enter sender addresses they will also be registered as access points with the UMS. Replies sent by the recipient to one of these addresses will be routed to this application. In this sample, there is an option to poll the receiving queue to retrieve such received messages. (This assumes that the underlying Messaging Driver used is capable of and configured for two-way messaging.)
Field value: EMAIL:test@oracle.com
- Enter Recipient Addresses: [e.g. "IM:recipient@example.com"]
Separate multiple addresses using comma.
Field value: EMAIL:john.doe@oracle.com
- Enter a Subject (optional):
Field value: hello
- Channel specific payload 1
Select Delivery Types:
Dropdown menu: SMS (selected), EMAIL, IM, VOICE, TWO_WAY_PAGER
- Content Type:
Field value: text/plain; charset=UTF
- Message Content:
Field value: This is a sample message.

3. As an optional step, enter the sender address in the following format:

Email: *sender_address*.

For example, enter Email:sender@oracle.com.

4. Enter one or more recipient addresses. For example, enter Email:recipient@oracle.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.

- As an optional step, enter a subject line or content for the email.
- Click **Send**. The Message Status page appears, showing the progress of transaction (Message received by Messaging engine for processing in [Figure A-6](#)).

Figure A-6 Message Status

UMS Sample to Send Messages

Sending message:
Sent message with id = f622d4dd984464dd008fbc395a2e5afa

Checking Status: [Refresh](#)

UMS: Message Status

Status for message id: f622d4dd984464dd008fbc395a2e5afa

Gateway Message ID	Address	Status Type	Status Content	Reporting Driver	Date	Failover Status
Recipient #1: EMAIL:john.doe@oracle.com ...						
	EMAIL:john.doe@oracle.com		Message received by Messaging engine for processing.		Jan 20, 2009 2:22:04 PM PST	false

- 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.*, as illustrated in [Figure A-7](#).

Figure A-7 Checking the Message Status

UMS Sample: Send Message - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://stapn50:7001/usermessagingsample/sample/send

UMS Sample to Send Messages

Sending message:
Sent message with id = f624a335984464dd008fbc39f9471bfa

Checking Status: [Refresh](#)

UMS: Message Status

Status for message id: f624a335984464dd008fbc39f9471bfa

Gateway Message ID	Address	Status Type	Status Content	Reporting D
Recipient #1: EMAIL:john.doe@oracle.com ...				
:25@31221839-2-john.doe@oracle.com	EMAIL:john.doe@oracle.com	✓	Outbound message delivery to remote gateway succeeded.	Farm_base_domain/base_domain/AdminServer/usermessagingsdri

A.2 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.

A.2.1 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. [Section A.2.2, "Configuring the Email Driver"](#)
2. [Section A.2.3, "Using Oracle JDeveloper 12c to Build the Application"](#)
3. [Section A.2.4, "Deploying the Application"](#)
4. [Section A.2.5, "Testing the Application"](#)

A.2.2 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.

A.2.3 Using Oracle JDeveloper 12c 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:

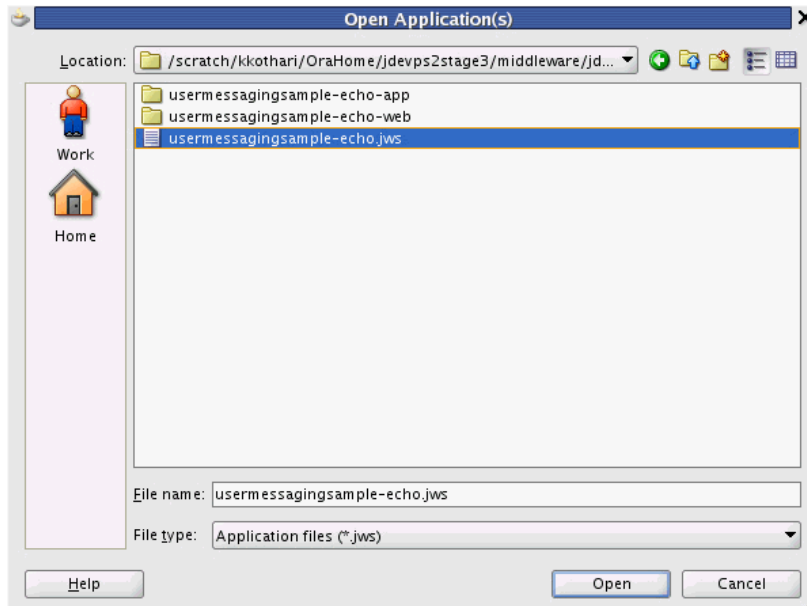
A.2.3.1 Opening the Project

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.

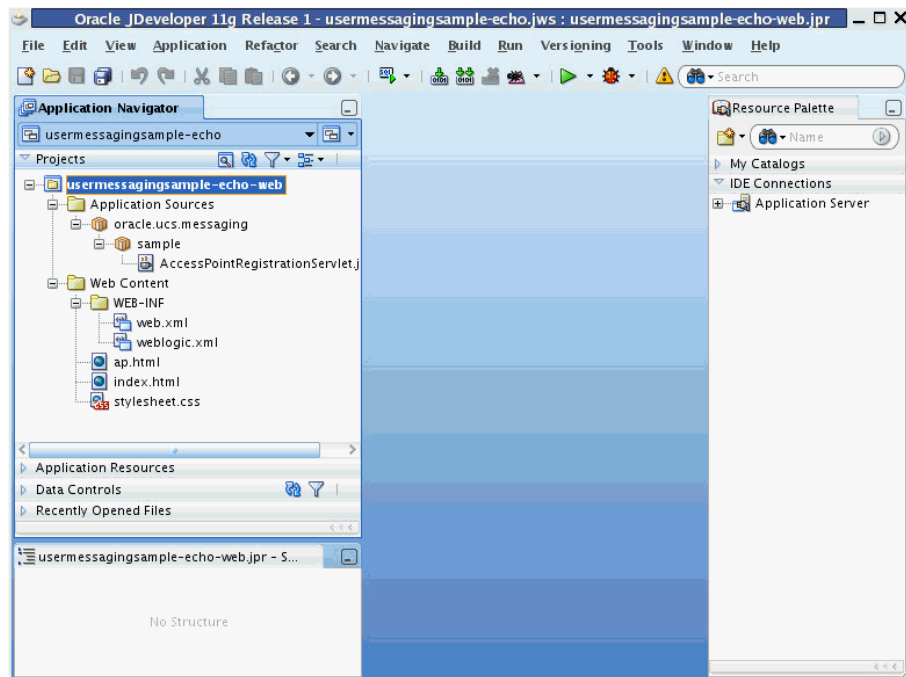
Note: If you choose to use a different directory, you must update the `oracle.sdp.messaging` library source path to `JDEV_HOME/communications/modules/oracle.sdp.messaging_12.1.2/sdpmessaging.jar`.

2. Open `usermessagingsample-echo.jws` (contained in the `.zip` file) in Oracle JDeveloper ([Figure A-8](#)).

Figure A-8 Opening the Project



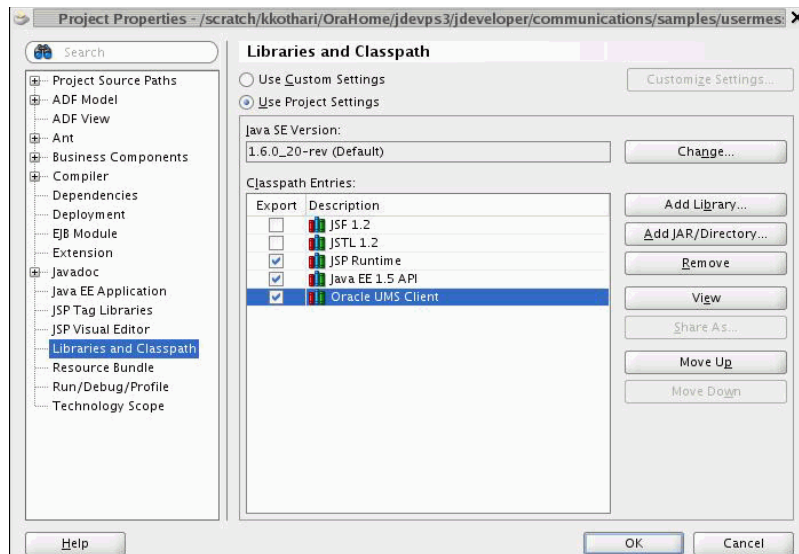
In the Oracle JDeveloper main window the project appears ([Figure A-9](#)).

Figure A-9 Oracle JDeveloper Main Window

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_11.1.1/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:

1. In the Application Navigator, right-click the module and select **Project Properties**.
2. In the left pane, select **Libraries and Classpath** (Figure A-10).

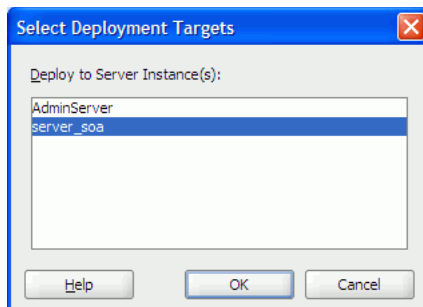
Figure A–10 Verifying Libraries

3. 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 points, and how the `EchoListener` is used to asynchronously receive messages.

A.2.4 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 [Section A.3, "Creating a New Application Server Connection."](#)
2. Deploy the application by selecting the **usermessagingsample-echo** application, **Deploy**, **usermessagingsample-echo**, **to**, and **SOA_server** ([Figure A–11](#)).

Figure A–11 Deploying the Project

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.

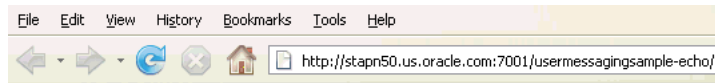
A.2.5 Testing the Application

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 (Figure A-12).

Figure A-12 Testing the Sample Application



UMS Samples

- Sample for Two Way Messaging

Perform the following steps:

1. Click on "Register/unregister Access Points".
2. Enter address of access point.
For example, `IM.myserver@example.com`.
3. Click on Submit.
4. Using your client send a message to the access point.
5. The sample application will receive the message and echo it back to you.

[Register/Unregister Access Points](#)

2. Click **Register/Unregister Access Points**. The *Access Point Registration* page appears (Figure A-13).

Figure A-13 Registering an Access Point

File Edit View History Bookmarks Tools Help

http://stapn50.us.oracle.com:7001/usermessagingsample-echo/ap.htm

UMS Sample App: Access Point Registration

Two Way Messaging Test requires the following steps:

1. Click on "Register/Unregister Access Points".
2. Enter address of access point, and optionally a keyword.
For example, IM:myserver@example.com
For Demo - IM:<ServerJabberID>
3. Click on Submit.
4. Using your IM client send a message to your buddy (ServerJabberID). If a keyword was specified, the first token of the message must match the keyword.
5. The sample application will receive the message and echo it back to you.

Use this form to register/unregister access point addresses for this Sample App.

Enter an Address:
[e.g. "IM:sender@example.com"
or "EMAIL:sender@example.com"]

Enter a keyword (optional):

Action:

Register
 Unregister

3. Enter the access point 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" in [Figure A-14](#)).

Figure A-14 Access Point Registration Status

File Edit View History Bookmarks Tools Help

http://stapn50.us.oracle.com:7001/usermessagingsample-echo/

UMS Sample: Access Point Registration

Registering access point:
EMAIL:myserver@example.com

Registered.

5. Send a message from your messaging client (for email, your email client) to the address you just registered as an access point 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.

A.3 Creating a New Application Server Connection

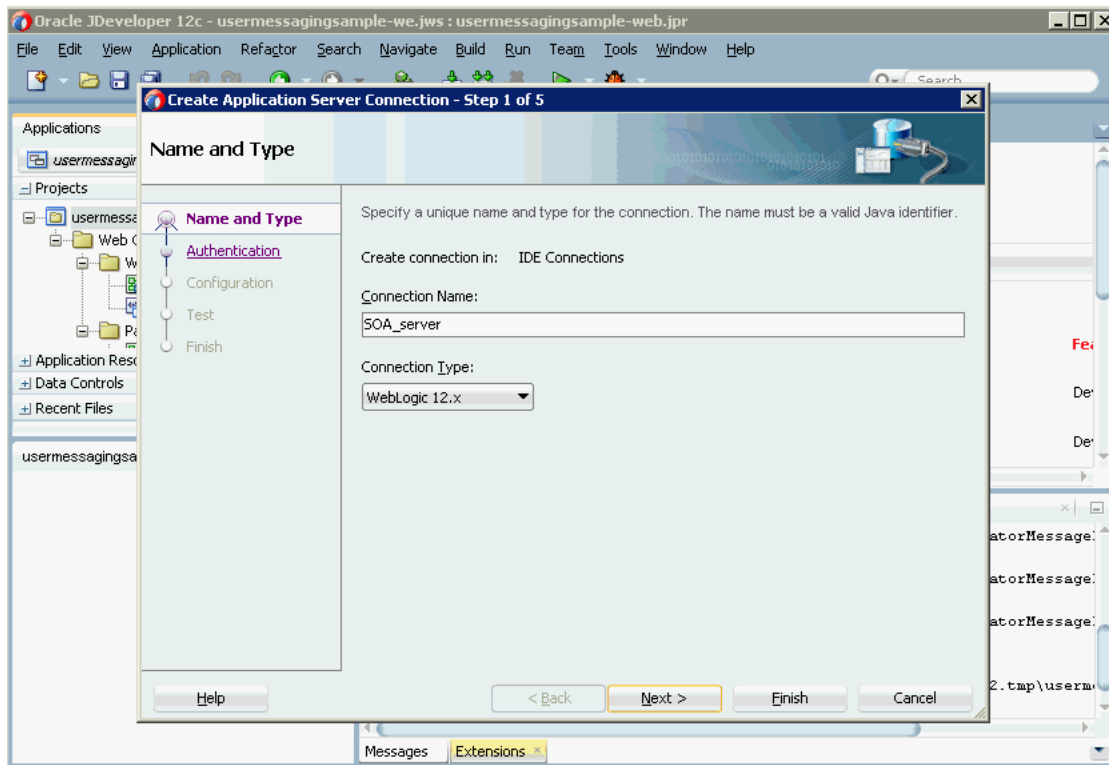
You define an application server connection in Oracle JDeveloper, and deploy and run the application. Perform the following steps to create an Application Server Connection.

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** as shown in [Figure A-15](#).
3. Select **WebLogic 12.x** from the Connection Type drop-down list.

Figure A-15 Create Application Server Connection



4. In the Authentication screen, enter your application server's admin **Username** and **Password**. Click **Next**.
5. In the Configuration screen, enter the 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.

