Oracle® WebLogic Communication Services

Developer's Guide

11*g* Release 1 (11.1.1)

E13807-01

May 2009

ORACLE®

Oracle WebLogic Communication Services Developer's Guide, 11*g* Release 1 (11.1.1)

E13807-01

Contents

Preface

This preface contains the following sections:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

This guide is intended for developers and programmers who want to use Oracle WebLogic Communication Services to develop, package, deploy, and test applications.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see the following documents in the Oracle WebLogic Communication Services set:

- *Oracle WebLogic Communication Services Administrator's Guide*
- *Oracle WebLogic Communication Services Installation Guide*

Conventions

The following text conventions are used in this document:

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

Part I

Introduction

This part contains introductory information.

Part I contains the following chapter:

- Chapter 1, "Overview of SIP Servlet Application Development"

1

Overview of SIP Servlet Application Development

This chapter describes SIP servlet application development in the following sections:

- Section 1.1, "What is a SIP Servlet?"
- Section 1.2, "Differences from HTTP Servlets"

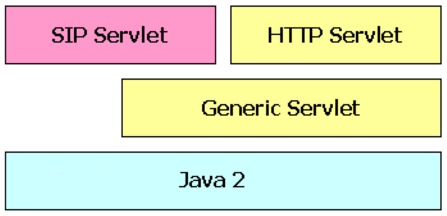
1.1 What is a SIP Servlet?

The SIP Servlet API is standardized as JSR289 of JCP (Java Community Process).

Note: In this document, the term "SIP Servlet" is used to represent the API, and "SIP servlet" is used to represent an application created with the API.

Java Servlets are for building server-side applications, HttpServlets are subclasses of Servlet and are used to create Web applications. SIP Servlet is defined as the generic servlet API with SIP-specific functions added.

Figure 1–1 Servlet API and SIP Servlet API



SIP Servlets are very similar to HTTP Servlets, and HTTP servlet developers can quickly adapt to the programming model. The service level defined by both HTTP and SIP Servlets is very similar, and you can easily design applications that support both HTTP and SIP. Listing 1 shows an example of a simple SIP servlet.

Example 1–1 SimpleSIPServlet.java

```
package oracle.example.simple;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.sip.*;

public class SimpleSIPServlet extends SipServlet {
    protected void doMessage(SipServletRequest req)
       throws ServletException, IOException
    {
       SipServletResponse res = req.createResponse(200);
       res.send();
    }
}
```

The above example shows a simple SIP servlet that sends back a 200 OK response to the SIP MESSAGE request. As you can see from the list, SIP Servlet and HTTP Servlet have many things in common:

1. Servlets must inherit the base class provided by the API. HTTP servlets must inherit HttpServlet, and SIP servlets must inherit SipServlet.
2. Methods doXxx must be overridden and implemented. HTTP servlets have doGet/doPost methods corresponding to GET/POST methods. Similarly, SIP servlets have doXxx methods corresponding to the method name (in the above example, the MESSAGE method). Application developers override and implement necessary methods.
3. The lifecycle and management method (init, destroy) of SIP Servlet are exactly the same as HTTP Servlet. Manipulation of sessions and attributes is also the same.
4. Although not shown in the example above, there is a deployment descriptor called sip.xml for a SIP servlet, which corresponds to web.xml. Application developers and service managers can edit this file to configure applications using multiple SIP servlets.

However, there are several differences between SIP and HTTP servlets. A major difference comes from protocols. The next section describes these differences as well as features of SIP servlets.

1.2 Differences from HTTP Servlets

This section describes differences between SIP Servlets and HTTP Servlets.

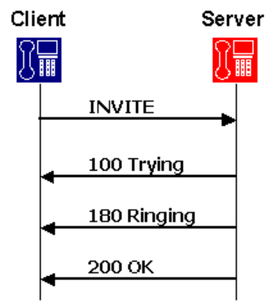
1.2.1 Multiple Responses

You might notice from Example 1–1 that the doMessage method has only one argument. In HTTP, a transaction consists of a pair of request and response, so arguments of a doXxx method specify a request (HttpServletRequest) and its response (HttpServletResponse). An application takes information such as parameters from the request to execute it, and returns its result in the body of the response.

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
```

For SIP, more than one response may be returned to a single request.

Figure 1–2 Example of Request and Response in SIP



The above figure shows an example of a response to the INVITE request. In this example, the server sends back three responses 100, 180, and 200 to the single INVITE request. To implement such sequence, in SIP Servlet, only a request is specified in a doXxx method, and an application generates and returns necessary responses in an overridden method.

Currently, SIP Servlet defines the following doXxx methods:

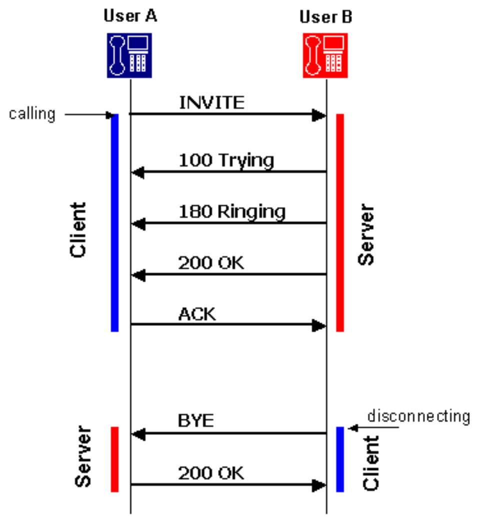
```
protected void doInvite(SipServletRequest req);
protected void doAck(SipServletRequest req);
protected void doOptions(SipServletRequest req);
protected void doBye(SipServletRequest req);
protected void doCancel(SipServletRequest req);
protected void doSubscribe(SipServletRequest req);
protected void doNotify(SipServletRequest req);
protected void doMessage(SipServletRequest req);
protected void doInfo(SipServletRequest req);
protected void doPrack(SipServletRequest req);
```

1.2.2 Receiving Responses

One of the major features of SIP is that roles of a client and server are not fixed. In HTTP, Web browsers always send HTTP requests and receive HTTP responses: They never receive HTTP requests and send HTTP responses. In SIP, however, each terminal needs to have functions of both a client and server.

For example, both of two SIP phones must call to the other and disconnect the call.

Figure 1–3 Relationship between Client and Server in SIP



The above example indicates that a calling or disconnecting terminal acts as a client. In SIP, roles of a client and server can be changed in one dialog. This client function is called UAC (User Agent Client) and server function is called UAS (User Agent Server), and the terminal is called UA (User Agent). SIP Servlet defines methods to receive responses as well as requests.

```
protected void doProvisionalResponse(SipServletResponse res);
protected void doSuccessResponse(SipServletResponse res);
protected void doRedirectResponse(SipServletResponse res);
protected void doErrorResponse(SipServletResponse res);
```

These doXxx response methods are not the method name of the request. They are named by the type of the response as follows:

- doProvisionalResponse—A method invoked on the receipt of a provisional response (or 1xx response).
- doSuccessResponse—A method invoked on the receipt of a success response.
- doRedirectResponse—A method invoked on the receipt of a redirect response.
- doErrorResponse—A method invoked on the receipt of an error response (or 4xx, 5xx, 6xx responses).

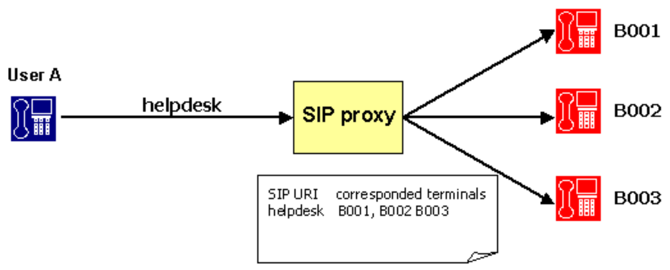
Existence of methods to receive responses indicates that in SIP Servlet requests and responses are independently transmitted an application in different threads. Applications must explicitly manage association of SIP messages. An independent request and response makes the process slightly complicated, but enables you to write more flexible processes.

Also, SIP Servlet allows applications to explicitly create requests. Using these functions, SIP servlets can not only wait for requests as a server (UAS), but also send requests as a client (UAC).

1.2.3 Proxy Functions

Another function that is different from the HTTP protocol is "forking." Forking is a process of proxying one request to multiple servers simultaneously (or sequentially) and used when multiple terminals (operators) are associated with one telephone number (such as in a call center).

Figure 1–4 Proxy Forking

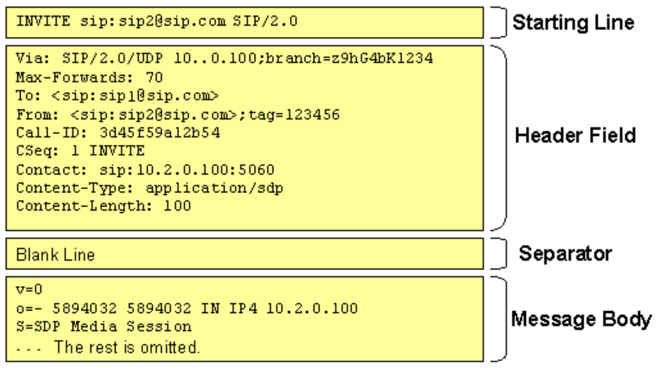


SIP Servlet provides a utility to proxy SIP requests for applications that have proxy functions.

1.2.4 Message Body

As the figure below, the structure of SIP messages is the same as HTTP.

Figure 1–5 SIP Message Example



```
INVITE sip:sip2@sip.com SIP/2.0
```

```
Via: SIP/2.0/UDP 10..0.100;branch=z9hG4bK1234
Max-Forwards: 70
To: <sip:sip1@sip.com>
From: <sip:sip2@sip.com>;tag=123456
Call-ID: 3d45f59a12b54
CSeq: 1 INVITE
Contact: sip:10.2.0.100:5060
Content-Type: application/sdp
Content-Length: 100
```

```
Blank Line
```

```
v=0
o=- 5894032 5894032 IN IP4 10.2.0.100
S=SDP Media Session
--- The rest is omitted.
```

HTTP is basically a protocol to transfer HTML files and images. Contents to be transferred are stored in the message body. HTTP Servlet defines stream manipulation-based API to enable sending and receiving massive contents.

1.2.4.1 Servlet Request

```
ServletInputStream getInputStream()
BufferedReader     getReader()
```

1.2.4.2 Servlet Response

```
ServletOutputStream getOutputStream()
PrintWriter         getWriter()
int  getBufferSize()
void setBufferSize(int size)
void resetBuffer()
void flushBuffer()
```

In SIP, however, only low-volume contents are stored in the message body since SIP is intended for real-time communication. Therefore, above methods are provided only for compatibility, and their functions are disabled.

In SIP, contents stored in the body include:

- SDP (Session Description Protocol)—A protocol to define multimedia sessions used between terminals. This protocol is defined in RFC2373.
- Presence Information—A message that describes presence information defined in CPIM.
- IM Messages—IM (instant message) body. User-input messages are stored in the message body.

Since the message body is in a small size, processing it in a streaming way increases overhead. SIP Servlet re-defines API to manipulate the message body on memory as follows:

1.2.4.3 SipServletMessage

```
void   setContent(Object content, String contentType)
Object getContent()
byte[] getRawContent()
```

1.2.5 Role of a Servlet Container

The following sections describe major functions provided by OWLCS as a SIP servlet container:

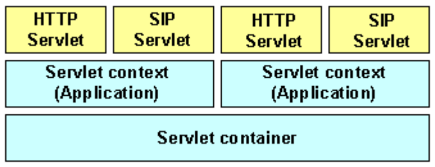
- Application Management—Describes functions such as application management by servlet context, lifecycle management of servlets, application initialization by deployment descriptors.
- SIP Messaging—Describes functions of parsing incoming SIP messages and delivering to appropriate SIP servlets, sending messages created by SIP servlets to appropriate UAS, and automatically setting SIP header fields.
- Utility Functions—Describes functions such as sessions, factories, and proxying that are available in SIP servlets.

1.2.5.1 Application Management

Like HTTP servlet containers, SIP servlet containers manage applications by servlet context (see Figure 6). Servlet contexts (applications) are normally archived in a WAR format and deployed in each application server.

Note: The method of deploying in application servers varies depending on your product. Refer to the documentation of your application server.

Figure 1–6 Servlet Container and Servlet Context



A servlet context for a converged SIP and Web application can include multiple SIP servlets, HTTP servlets, and JSPs.

OWLCS can deploy applications using the same method as the application server you use as the platform. However, if you deploy applications including SIP servlets, you need a SIP specific deployment descriptor (sip.xml) defined by SIP servlets. The table below shows the file structure of a general converged SIP and Web application.

Table 1–1 File Structure Example of Application

File	Description
WEB-INF/	Place your configuration and executable files of your converged SIP and Web application in the directory. You cannot directly refer to files in this directory on Web (servlets can do this).
WEB-INF/web.xml	The Java EE standard configuration file for the Web application.
WEB-INF/sip.xml	The SIP Servlet-defined configuration files for the SIP application.
WEB-INF/classes/	Store compiled class files in the directory. You can store both HTTP and SIP servlets in this directory.
WEB-INF/lib/	Store class files archived as Jar files in the directory. You can store both HTTP and SIP servlets in this directory.
*.jsp, *.jpg	Files comprising the Web application (for example JSP) can be deployed in the same way as Java EE.

Information specified in the sip.xml file is similar to that in the web.xml except <servlet-mapping> setting that is different from HTTP servlets. In HTTP you specify a servlet associated with the file name portion of URL. But SIP has no concept of the file name. You set filter conditions using URI or the header field of a SIP request. The following example shows that a SIP servlet called "register" is assigned all REGISTER methods.

Example 1–2 Filter Condition Example of sip.xml

```
<servlet-mapping>
  <servlet-name>registrar</servlet-name>
  <pattern>
    <equal>
      <var>request.method</var>
      <value>REGISTER</value>
    </equal>
  </pattern>
</servlet-mapping>
```

Once deployed, lifecycle of the servlet context is maintained by the servlet container. Although the servlet context is normally started and shutdown when the server is started and shutdown, the system administrator can explicitly start, stop, and reload the servlet context.

1.2.5.2 SIP Messaging

SIP messaging functions provided by a SIP servlet container are classified under the following types:

- Parsing received SIP messages.
- Delivering parsed messages to the appropriate SIP servlet.
- Sending SIP servlet-generated messages to the appropriate UA
- Automatically generating a response (such as "100 Trying").
- Automatically managing the SIP header field.

All SIP messages that a SIP servlet handles are represented as a SipServletRequest or SipServletResponse object. A received message is first parsed by the parser and then translated to one of these objects and sent to the SIP servlet container.

A SIP servlet container receives the following three types of SIP messages, for each of which you determine a target servlet.

- First SIP Request—When the SIP servlet container received a request that does not belong to any SIP session, it uses filter conditions in the sip.xml file (described in the previous section) to determine the target SIP servlet. Since the container creates a new SIP session when the initial request is delivered, any SIP requests received after that point are considered as subsequent requests.

  > **Note:** Filtering should be done carefully. In OWLCS, when the received SIP message matches multiple SIP servlets, it is delivered only to any one SIP servlet.
  >
  > The use of additional criteria such as request parameters can be used to direct a request to a servlet.

- Subsequent SIP Request—When the SIP Servlet container receives a request that belongs to any SIP session, it delivers the request to a SIP Servlet associated with that session. Whether the request belongs to a session or not is determined using dialog ID.

  Each time a SIP Servlet processes messages, a lock is established by the container on the call ID. If a SIP Servlet is currently processing earlier requests for the same call ID when subsequent requests are received, the SIP Servlet container queues the subsequent requests. The queued messages are processed only after the Servlet has finished processing the initial message and has returned control to the SIP Servlet container.

  This concurrency control is guaranteed both in a single containers and in clustered environments. Application developers can code applications with the understanding that only one message for any particular call ID gets processed at a given time.

- SIP Response—When the received response is to a request that a SIP servlet proxied, the response is automatically delivered to the same servlet since its SIP session had been determined. When a SIP servlet sends its own request, you must first specify a servlet that receives a response in the SIP session. For example, if the

SIP servlet sending a request also receives the response, the following handler setting must be specified in the SIP session.

```
SipServletRequest req = getSipFactory().createRequest(appSession, ...);
req.getSession().setHandler(getServletName());
```

Normally, in SIP a "session" means a real-time session by RTP/RTSP. On the other hand, in HTTP Servlet a "session" refers to a way of relating multiple HTTP transactions. In this document, session-related terms are defined as follows:

Table 1–2 Session-Related Terminology

Realtime Session	A realtime session established by RTP/RTSP.
HTTP Session	A session defined by HTTP Servlet. A means of relating multiple HTTP transactions.
SIP Session	A means of implementing the same concept as in HTTP session in SIP. SIP (RFC3261) has a similar concept of "dialog," but in this document this is treated as a different term since its lifecycle and generation conditions are different.
Application Session	A means for applications using multiple protocols and dialogs to associate multiple HTTP sessions and SIP sessions. Also called "AP session."

OWLCS automatically execute the following response and retransmission processes:

- Sending "100 Trying"—When WebLogic Communications Server receives an INVITE request, it automatically creates and sends "100 Trying."
- Response to CANCEL—When WebLogic Communications Server receives a CANCEL request, it executes the following processes if the request is valid.
  1. Sends a 200 response to the CANCEL request.
  2. Sends a 487 response to the INVITE request to be cancelled.
  3. Invokes a doCancel method on the SIP servlet. This allows the application to abort the process within the doCancel method, eliminating the need for explicitly sending back a response.
- Sends ACK to an error response to INVITE—When a 4xx, 5xx, or 6xx response is returned for INVITE that were sent by a SIP servlet, WebLogic Communications Server automatically creates and sends ACK. This is because ACK is required only for a SIP sequence, and the SIP servlet does not require it.

  When the SIP servlet sends a 4xx, 5xx, or 6xx response to INVITE, it never receives ACK for the response.
- Retransmission process when using UDP—SIP defines that sent messages are retransmitted when low-trust transport including UDP is used. WebLogic Communications Server automatically do the retransmission process according to the specification.

Mostly, applications do not need to explicitly set and see header fields In HTTP Servlet since HTTP servlet containers automatically manage these fields such as Content-Length and Content-Type. SIP Servlet also has the same header management function.

In SIP, however, since important information about message delivery exists in some fields, these headers are not allowed to change by applications. Headers that can not be changed by SIP servlets are called "system headers." The table below lists system headers:

Table 1–3 System Headers

Header Name	Description
Call-ID	Contains ID information to associate multiple SIP messages as Call.
From, To	Contains Information on the sender and receiver of the SIP request (SIP, URI, etc.). tag parameters are given by the servlet container.
CSeq	Contains sequence numbers and method names.
Via	Contains a list of servers the SIP message passed through. This is used when you want to keep track of the path to send a response to the request.
Record-Route, Route	Used when the proxy server mediates subsequent requests.
Contact	Contains network information (such as IP address and port number) that is used for direct communication between terminals. For a REGISTER message, 3xx, or 485 response, this is not considered as the system header and SIP servlets can directly edit the information.

1.2.5.3 Utility Functions

SIP Servlet defines the following utilities that are available to SIP servlets:

1. SIP Session, Application Session
2. SIP Factory
3. Proxy

1.2.5.3.1 SIP Session, Application Session As stated before, SIP Servlet provides a "SIP session" whose concept is the same as a HTTP session. In HTTP, multiple transactions are associated using information like Cookie. In SIP, this association is done with header information (Call-ID and tag parameters in From and To). Servlet containers maintain and manage SIP sessions. Messages within the same dialog can refer to the same SIP session. Also, For a method that does not create a dialog (such as MESSAGE), messages can be managed as a session if they have the same header information.

SIP Servlet has a concept of an "application session," which does not exist in HTTP Servlet. An application session is an object to associate and manage multiple SIP sessions and HTTP sessions. It is suitable for applications such as B2BUA.

1.2.5.3.2 SIP Factory A SIP factory (SipFactory) is a factory class to create SIP Servlet-specific objects necessary for application execution. You can generate the following objects:

Table 1–4 Objects Generated with SipFactory

Class Name	Description
URI, SipURI, Address	Can generate address information including SIP URI from String.
SipApplicationSession	Creates a new application session. It is invoked when a SIP servlet starts a new SIP signal process.
SipServletRequest	Used when a SIP servlet acts as UAC to create a request. Such requests can not be sent with Proxy.proxyTo. They must be sent with SipServletRequest.send.

SipFactory is located in the servlet context attribute under the default name. You can take this with the following code.

```
ServletContext context = getServletContext();
SipFactory factory =
    (SipFactory) context.getAttribute("javax.servlet.sip.SipFactory");
```

1.2.5.3.3 Proxy Proxy is a utility used by a SIP servlet to proxy a request. In SIP, proxying has its own sequences including forking. You can specify the following settings in proxying with Proxy:

- Recursive routing (recurse)—When the destination of proxying returns a 3xx response, the request is proxied to the specified target.
- Record-Route setting—Sets a Record-Route header in the specified request.
- Parallel/Sequential (parallel)—Determines whether forking is executed in parallel or sequentially.
- stateful—Determines whether proxying is transaction stateful. This parameter is not relevant because stateless proxy mode is deprecated in JSR289.
- Supervising mode—In the event of the state change of proxying (response receipts), an application reports this.

Part II

Developing and Programming SIP Applications

This part describes programming guidelines and procedures for SIP applications.

Part II contains the following chapters:

2 Developing Converged Applications

This chapter describes how to develop converged HTTP and SIP applications with OWLCS, in the following sections:

- Section 2.1, "Overview of Converged Applications"
- Section 2.2, "Assembling and Packaging a Converged Application"
- Section 2.3, "Working with SIP and HTTP Sessions"
- Section 2.4, "Using the Converged Application Example"

2.1 Overview of Converged Applications

In a *converged application*, SIP protocol functionality is combined with HTTP or Java EE components to provide a unified communication service. For example, an online push-to-talk application might enable a customer to initiate a voice call to ask questions about products in their shopping cart. The SIP session initiated for the call is associated with the customer's HTTP session, which enables the employee answering the call to view customer's shopping cart contents or purchasing history.

You must package converged applications that utilize Java EE components (such as EJBs) into an application archive (.EAR file). Converged applications that use SIP and HTTP protocols must be packaged in a single SAR or WAR file containing both a `sip.xml` and a `web.xml` deployment descriptor file.You can optionally package the SIP and HTTP Servlets of a converged application into separate SAR and WAR components within a single EAR file.

The HTTP and SIP sessions used in a converged application can be accessed programmatically through a common application session object. The SIP Servlet API also helps you associate HTTP sessions with an application session.

2.2 Assembling and Packaging a Converged Application

The SIP Servlet specification fully describes the requirements and restrictions for assembling converged applications. The following statements summarize the information in the SIP Servlet specification:

- Use the standard SIP Servlet directory structure for converged applications.
- Store all SIP Servlet files under the `WEB-INF` subdirectory; this ensures that the files are not served up as static files by an HTTP Servlet.
- Include deployment descriptors for both the HTTP and SIP components of your application. This means that both `sip.xml` and `web.xml` descriptors are

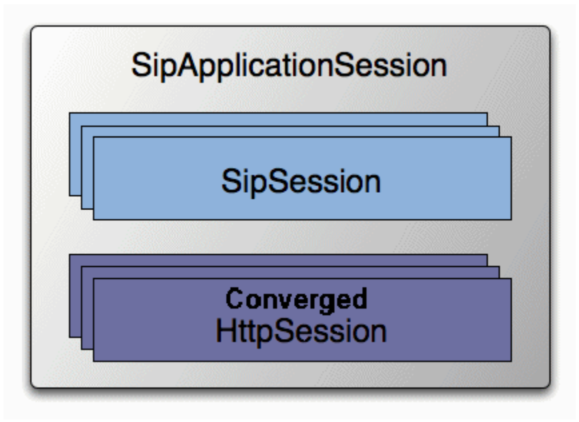
required. A `weblogic.xml` deployment descriptor may also be included to configure Servlet functionality in the OWLCS container.

- Observe the following restrictions on deployment descriptor elements:
  - The `distributable` tag must be present in both `sip.xml` and `web.xml`, or it must be omitted entirely.
  - `context-param` elements are shared for a given converged application. If you define the same `context-param` element in `sip.xml` and in `web.xml`, the parameter must have the same value in each definition.
  - If either the `display-name` or `icons` element is required, the element must be defined in both `sip.xml` and `web.xml`, and it must be configured with the same value in each location.

2.3 Working with SIP and HTTP Sessions

As shown in Figure 2–1, each converged application deployed to the OWLCS container has a unique `SipApplicationSession`, which can contain one or more `SipSession` and `ConvergedHttpSession` objects.

Figure 2–1 Sessions in a Converged Application



The API provided by `javax.servlet.SipApplicationSession` enables you to iterate through all available sessions in a given `SipApplicationSession`. It also provides methods to encode a URL with the unique application session when developing converged applications.

In prior releases, OWLCS extended the basic SIP Servlet API to provide methods for:

- Creating new HTTP sessions from a SIP Servlet
- Adding and removing HTTP sessions from `SipApplicationSession`
- Obtaining `SipApplicationSession` objects using either the call ID or session ID
- Encoding HTTP URLs with session IDs from within a SIP Servlet

This functionality is now provided directly as part of the SIP Servlet API version 1.1, and the proprietary API (`com.bea.wcp.util.Sessions`) is now deprecated. Table 2–1 lists the SIP Servlet APIs to use in place of now deprecated methods. See the SIP Servlet v1.1 API JavaDoc for more information.

Table 2–1 Deprecated com.bea.wcp.util.Sessions Methods

Deprecated Method (in com.bea.wcp.util.Sessions)	Replacement Method	Description
getApplicationSession	javax.servlet.sip.SipSessionsUtil. getApplicationSession	Obtains the `SipApplicationSession` object with a specified session ID.
getApplicationSessionsByCallId	None.	Obtains an Iterator of `SipApplicationSession` objects associated with the specified call ID.
createHttpSession	None.	Applications can instead cast an `HttpSession` into `ConvergedHttpSession`.
setApplicationSession	javax.servlet.sip.ConvergedHttpSession. getApplicationSession	Associates an HTTP session with an existing `SipApplicationSession`.
removeApplicationSession	None.	Removes an HTTP session from an existing `SipApplicationSession`.
getEncodeURL	javax.servlet.sip.ConvergedHttpSession. encodeURL	Encodes an HTTP URL with the `jsessionid` of an existing HTTP session object.

Note: The `com.bea.wcp.util.Sessions` API is provided only for backward compatibility. Use the SIP Servlet APIs for all new development. OWLCS does not support converged applications that mix the `com.bea.wcp.util.Sessions` API and JSR 289 convergence APIs.

Specifically, the deprecated `Sessions.getApplicationSessionsByCallId(String callId)` method cannot be used with v1.1 SIP Servlets that use the session key-based targeting method for associating an initial request with an existing SipApplicationSession object. See Section 15.11.2 in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for more information about this targeting mechanism.

2.3.1 Modifying the SipApplicationSession

When using a replicated domain, OWLCS automatically provides concurrency control when a SIP Servlet modifies a `SipApplicationSession` object. In other words, when a SIP Servlet modifies the `SipApplicationSession` object, the SIP container automatically locks other applications from modifying the object at the same time.

Non-SIP applications, such as HTTP Servlets, must themselves ensure that the application call state is locked before modifying it. This is also required if a single SIP Servlet needs to modify other call state objects, such as when a conferencing Servlet joins multiple calls.

To help application developers manage concurrent access to the application session object, OWLCS extends the standard `SipApplicationSession` object with `com.bea.wcp.sip.WlssSipApplicationSession`, and adds two interfaces, com.bea.wcp.sip.WlssAction and com.bea.wcp.sip.WlssAsynchronous Action, to encapsulate tasks performed to modify the session. When these APIs are used, the SIP container ensures that all business logic contained within the WlssAction and WlssAsynchronousAction object is executed on a locked copy of the associated SipApplicationSession instance. The sections that follow describe each interface.

2.3.1.1 Synchronous Access

Applications that need to read and update a session attribute in a transactional and synchronous manner must use the WlssAction API. WlssAction obtains an explicit lock on the session for the duration of the action. Example 2–1, "Example Code using WlssAction API" shows an example of using this API.

Example 2–1 Example Code using WlssAction API

```
final SipApplicationSession appSession = ...;
WlssSipApplicationSession wlssAppSession = (WlssSipApplicationSession) appSession;
wlssAppSession.doAction(new WlssAction() {
       public Object run() throws Exception {
         // Add all business logic here.
         appSession.setAttribute("counter", latestCounterValue);
         sipSession.setAttribute("currentState", latestAppState);
         // The SIP container ensures that the run method is invoked
         // while the application session is locked.
         return null;
       }
});
```

Because the WlssAction API obtains an exclusive lock on the associated session, deadlocks can occur if you attempt to modify other application session attributes within the action.

2.3.1.2 Asynchronous Access

Applications that need to update a different SipApplicationSession while in the context of a locked SipApplicationSession can perform asynchronous updates using the WlssAsynchronousAction API. This API reduces contention when multiple applications might need to update attributes in the same SipApplicationSession at the same time. Example 2–2, "Example Code using WlssAsynchronousAction API" shows an example of using this API.

To compile applications using this API, you need to include MIDDLEWARE_HOME/server/lib/wlss/wlssapi.jar, and MIDDLEWARE_HOME/server/lib/wlss/sipservlet.jar.

Example 2–2 Example Code using WlssAsynchronousAction API

```
SipApplicationSession sas1 = req.getSipApplicationSession(); //
SipApplicationSession1 is already locked by the container
  // Obtain another SipApplicationSession to schedule work on it
  WlssSipApplicationSession wlssSipAppSession =
SipSessionsUtil.getApplicationSessionById(conferenceAppSessionId);
  // The work is done on the application session asynchronously
  appSession.doAsynchronousAction(new WlssAsynchronousAction() {
    Serializable run(SipApplicationSession appSession) {
      // Add all business logic here.
      int counter = appSession.getAttribute("counter");
      ++ counter;
      appSession.setAttribute("counter", counter);
```

```
            return null;
        }
    });
```

Performing the work on appSession in an asynchronous manner prevents nested locking and associated deadlock scenarios.

2.4 Using the Converged Application Example

OWLCS includes a sample converged application that uses the `com.bea.wcp.util.Sessions` API. All source code, deployment descriptors, and build files for the example can be installed in *`OWLCS_HOME`*`\samples\sipserver\examples\src\convergence`. See the `readme.html` file in the example directory for instructions about how to build and run the example.

3 SIP Protocol Programming

This chapter describes programming SIP applications and contains the following sections:

- Section 3.1, "Using Compact and Long Header Formats for SIP Messages"
- Section 3.2, "Using Content Indirection in SIP Servlets"
- Section 3.3, "Generating SNMP Traps from Application Code"

3.1 Using Compact and Long Header Formats for SIP Messages

This section describes how to use the OWLCS `SipServletMessage` interface and configuration parameters to control SIP message header formats

3.1.1 Overview of Header Format APIs and Configuration

Applications that operate on wireless networks may want to limit the size of SIP headers to reduce the size of messages and conserve bandwidth. JSR 289 provides the `SipServletMessage.setHeaderForm()` method, which enables application developers to set a long or compact format for the value of a given header.

One feature of the `SipServletMessage` API provided in JSR 289 is the ability to set long or compact header formats for the entire SIP message using the `setHeaderForm` method.

In addition to `SipServletMessage`, OWLCS provides a container-wide configuration parameter that can control SIP header formats for all system-generated headers. This system-wide parameter can be used along with `SipServletMessage.setHeaderForm` and `SipServletMessage.setHeader` to further customize header formats.

3.1.2 Summary of Compact Headers

Table 3–1 defines the compact header abbreviations described in the SIP specification (`http://www.ietf.org/rfc/rfc3261.txt`). Specifications that introduce additional headers may also include compact header abbreviations.

Table 3–1 Compact Header Abbreviations

Header Name (Long Format)	Compact Format
Call-ID	i
Contact	m

Table 3–1 (Cont.) Compact Header Abbreviations

Header Name (Long Format)	Compact Format
Content-Encoding	e
Content-Length	l
Content-Type	c
From	f
Subject	s
Supported	k
To	t
Via	v

3.1.3 Assigning Header Formats with WlssSipServletMessage

A pair of getter/setter methods, `setHeaderForm` and `getHeaderForm`, are used to assign or retrieve the header formats used in the message. These methods assign or return a `HeaderForm` object, which is a simple Enumeration that describes the header format:

- `COMPACT`—Forces all headers in the message to use compact format. This behavior is similar to the container-wide configuration value of "force compact," as described in use-compact-form in the *Configuration Reference Manual*.
- `LONG`—Forces all headers in the message to use long format. This behavior is similar to the container-wide configuration value of "force long," as described in use-compact-form in the *Configuration Reference Manual*.
- `DEFAULT`—Defers the header format to the container-wide configuration value set in use-compact-form.

`SipServletResponse.setHeaderForm` can be used in combination with `SipServletMessage.setHeader` and the container-level configuration parameter, use-compact-form.

3.1.4 Summary of API and Configuration Behavior

Header formats can be specified at the header, message, and SIP Servlet container levels. Table 3–1 shows the header format that results when adding a new header with `SipServletMessage.setHeader`, given different container configurations and message-level settings with `SipServletMessage.setHeaderForm`.

Table 3–2 API Behavior when Adding Headers

SIP Servlet Container Header Configuration (`use-compact-form` Setting)	.SIPServletMessage setHeaderForm Setting	SipServletMessage. setHeader Value	Resulting Header
COMPACT	DEFAULT	"Content-Type"	"Content-Type"
COMPACT	DEFAULT	"c"	"c"
COMPACT	COMPACT	"Content-Type"	"c"
COMPACT	COMPACT	"c"	"c"
COMPACT	LONG	"Content-Type"	"Content-Type"

Table 3–2 (Cont.) API Behavior when Adding Headers

COMPACT	LONG	"c"	"Content-Type"
LONG	DEFAULT	"Content-Type"	"Content-Type"
LONG	DEFAULT	"c"	"c"
LONG	COMPACT	"Content-Type"	"c"
LONG	COMPACT	"c"	"c"
LONG	LONG	"Content-Type"	"Content-Type"
LONG	LONG	"c"	"Content-Type"
FORCE_COMPACT	DEFAULT	"Content-Type"	"c"
FORCE_COMPACT	DEFAULT	"c"	"c"
FORCE_COMPACT	COMPACT	"Content-Type"	"c"
FORCE_COMPACT	COMPACT	"c"	"c"
FORCE_COMPACT	LONG	"Content-Type"	"Content-Type"
FORCE_COMPACT	LONG	"c"	"Content-Type"
FORCE_LONG	DEFAULT	"Content-Type"	"Content-Type"
FORCE_LONG	DEFAULT	"c"	"Content-Type"
FORCE_LONG	COMPACT	"Content-Type"	"c"
FORCE_LONG	COMPACT	"c"	"c"
FORCE_LONG	LONG	"Content-Type"	"Content-Type"
FORCE_LONG	LONG	"c"	"Content-Type"

Table 3–1 shows the system header format that results when setting the header format with `WlssSipServletResponse.setUseHeaderForm` given different container configuration values.

Table 3–3 API Behavior for System Headers

SIP Servlet Container Header Configuration (`use-compact-form` Setting)	SipServletMessage. setHeaderForm Setting	Resulting Contact Header
COMPACT	DEFAULT	"m"
COMPACT	COMPACT	"m"
COMPACT	LONG	"Contact"
LONG	DEFAULT	"Contact"
LONG	COMPACT	"m"
LONG	LONG	"Contact"
FORCE_COMPACT	DEFAULT	"m"
FORCE_COMPACT	COMPACT	"m"
FORCE_COMPACT	LONG	"Contact"
FORCE_LONG	DEFAULT	"Contact"
FORCE_LONG	COMPACT	"m"

Table 3–3 (Cont.) API Behavior for System Headers

FORCE_LONG	LONG	"Contact"

3.2 Using Content Indirection in SIP Servlets

This section describes how to develop SIP Servlets that work with indirect content specified in the SIP message body.

3.2.1 Overview of Content Indirection

Data provided by the body of a SIP message can be included either directly in the SIP message body, or indirectly by specifying an HTTP URL and metadata that describes the URL content. Indirectly specifying the content of the message body is used primarily in the following scenarios:

- When the message bodies include large volumes of data. In this case, content indirection can be used to transfer the data outside of the SIP network (using a separate connection or protocol).
- For bandwidth-limited applications. In this case, content indirection provides enough metadata for the application to determine whether or not it must retrieve the message body (potentially degrading performance or response time).

OWLCS provides a simple API that you can use to work with indirect content specified in SIP messages.

3.2.2 Using the Content Indirection API

The content indirection API provided by OWLCS helps you quickly determine if a SIP message uses content indirection, and to easily retrieve all metadata associated with the indirect content. The basic API consists of a utility class, `com.bea.wcp.sip.util.ContentIndirectionUtil`, and an interface for accessing content metadata, `com.bea.wcp.sip.util`.

SIP Servlets can use the utility class to identify SIP messages having indirect content, and to retrieve an `ICParsedData` object representing the content metadata. The `ICParsedData` object has simple "getter" methods that return metadata attributes.

3.2.3 Additional Information

Complete details about content indirection are available in RFC 4483.

See the Oracle Fusion Middleware WebLogic Communication Services API Reference for additional documentation about the content indirection API.

3.3 Generating SNMP Traps from Application Code

This section describes how to use the OWLCS `SipServletSnmpTrapRuntimeMBean` to generate SNMP traps from within a SIP Servlet.

3.3.1 Overview

OWLCS includes a runtime MBean, `SipServletSnmpTrapRuntimeMBean,` that enables applications to easily generate SNMP traps. The OWLCS MIB contains seven new OIDs that are reserved for traps generated by an application. Each OID corresponds to a severity level that the application can assign to a trap, in order from the least severe to the most severe:

- Info
- Notice
- Warning
- Error
- Critical
- Alert
- Emergency

To generate a trap, an application simply obtains an instance of the `SipServletSnmpTrapRuntimeMBean` and then executes a method that corresponds to the desired trap severity level (`sendInfoTrap()`, `sendWarningTrap()`, `sendErrorTrap()`, `sendNoticeTrap()`, `sendCriticalTrap()`, `sendAlertTrap()`, and `sendEmergencyTrap()`). Each method takes a single parameter—the String value of the trap message to generate.

For each SNMP trap generated in this manner, OWLCS also automatically transmits the Servlet name, application name, and OWLCS instance name associated with the calling Servlet.

3.3.2 Requirement for Accessing SipServletSnmpTrapRuntimeMBean

In order to obtain a `SipServletSnmpTrapRuntimeMBean`, the calling SIP Servlet must be able to perform MBean lookups from the Servlet context. To enable this functionality, you must assign a OWLCS administrator `role-name` entry to the `security-role` and `run-as` role elements in the `sip.xml` deployment descriptor. Example 3–1 shows a sample `sip.xml` file with the required role elements highlighted.

Example 3–1 Sample Role Requirement in sip.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sip-app
   PUBLIC "-//Java Community Process//DTD SIP Application 1.0//EN"
   "http://www.jcp.org/dtd/sip-app_1_0.dtd">
<sip-app>
  <display-name>My SIP Servlet</display-name>
  <distributable/>
  <servlet>
    <servlet-name>myservlet</servlet-name>
    <servlet-class>com.mycompany.MyServlet</servlet-class>
    <run-as>
      <role-name>weblogic</role-name>
    </run-as>
  </servlet>
  <servlet-mapping>
    <servlet-name>myservlet</servlet-name>
    <pattern>
      <equal>
<var>request.method</var>
<value>INVITE</value>
      </equal>
    </pattern>
  </servlet-mapping>
  <security-role>
    <role-name>weblogic</role-name>
  </security-role>
</sip-app>
```

3.3.3 Obtaining a Reference to SipServletSnmpTrapRuntimeMBean

Any SIP Servlet that generates SNMP traps must first obtain a reference to the `SipServletSnmpTrapRuntimeMBean`. Example 3–2 shows the sample code for a method to obtain the MBean.

Example 3–2 Sample Method for Accessing SipServletSnmpTrapRuntimeMBean

```
public SipServletSnmpTrapRuntimeMBean getServletSnmpTrapRuntimeMBean() {
    MBeanHome localHomeB = null;
    SipServletSnmpTrapRuntimeMBean ssTrapMB = null;

    try
    {
      Context ctx = new InitialContext();
      localHomeB = (MBeanHome)ctx.lookup(MBeanHome.LOCAL_JNDI_NAME);
      ctx.close();
    } catch (NamingException ne){
      ne.printStackTrace();
    }

    Set set = localHomeB.getMBeansByType("SipServletSnmpTrapRuntime");
    if (set == null || set.isEmpty()) {
      try {
        throw new ServletException("Unable to lookup type
'SipServletSnmpTrapRuntime'");
      } catch (ServletException e) {
        e.printStackTrace();
      }
    }
    ssTrapMB = (SipServletSnmpTrapRuntimeMBean) set.iterator().next();
    return ssTrapMB;
}
```

3.3.4 Generating an SNMP Trap

In combination with the method shown in Example 3–2, Example 3–3 demonstrates how a SIP Servlet would use the MBean instance to generate an SNMP trap in response to a SIP INVITE.

Example 3–3 Generating a SNMP Trap

```
public class MyServlet extends SipServlet {
  private SipServletSnmpTrapRuntimeMBean sipServletSnmpTrapMb = null;

  public MyServlet () {
  }

  public void init (ServletConfig sc) throws ServletException {
    super.init (sc);
    sipServletSnmpTrapMb = getServletSnmpTrapRuntimeMBean();
  }

  protected void doInvite(SipServletRequest req) throws IOException {
    sipServletSnmpTrapMb.sendInfoTrap("Rx Invite from " + req.getRemoteAddr() +
"with call id" + req.getCallId());
  }
}
```

4 Requirements and Best Practices for SIP Applications

This chapter describes requirements and best practices for developing applications for deployment to OWLCS. It contains the following sections:

4.1 Overview of Developing Distributed Applications for Oracle Communications Converged Application Server

In a typical production environment, SIP applications are deployed to a cluster of OWLCS instances that form the engine tier cluster. A separate cluster of servers in the SIP data tier provides a replicated, in-memory database of the call states for active calls. In order for applications to function reliably in this environment, you must observe the programming practices and conventions described in the sections that follow to ensure that multiple deployed copies of your application perform as expected in the clustered environment.

If you are porting an application from a previous version of OWLCS, the conventions and restrictions described below may be new to you, because the 2.0 and 2.1 versions of WebLogic SIP Server implementations did not support clustering. Thoroughly test and profile your ported applications to discover problems and ensure adequate performance in the new environment.

4.2 Applications Must Not Create Threads

OWLCS is a multi-threaded application server that carefully manages resource allocation, concurrency, and thread synchronization for the modules it hosts. To obtain the greatest advantage from the OWLCS architecture, construct your application modules according to the SIP Servlet and Java EE API specifications.

Avoid application designs that require creating new threads in server-side modules such as SIP Servlets:

- The SIP Servlet container automatically locks the associated call state when invoking the `do`*`xxx`* method of a SIP Servlet. If the `doxxx` method spawns additional threads or accesses a different call state before returning control, *deadlock scenarios and lost updates to session data can occur*.
- Applications that create their own threads do not scale well. Threads in the JVM are a limited resource that must be allocated thoughtfully. Your applications may break or cause poor OWLCS performance when the server load increases. Problems such as deadlocks and thread starvation may not appear until the application is under a heavy load.
- Multithreaded modules are complex and difficult to debug. Interactions between application-generated threads and WebLogic Server threads are especially difficult to anticipate and analyze.
- The `WlssSipApplicationSession.doAction()` method, described in "Use setAttribute() to Modify Session Data in “No-Call” Scope", does not provide synchronization for spawned Java threads. Any threads created within `doAction()` can execute another `doAction()` on the same `WlssSipApplicationSession`. Similarly, main threads that use `doAction()` to access a different `wlssSipApplicationSession` can lead to deadlocks, because the container automatically locks main threads when processing incoming SIP messages. "Use setAttribute() to Modify Session Data in “No-Call” Scope" describes a potential deadlock situation.

Caution: If your application must spawn threads, you must guard against deadlocks and carefully manage concurrent access to session data. At a minimum, never spawn threads inside the service method of a SIP Servlet. Instead, maintain a separate thread pool outside of the service method, and be careful to synchronize access to all session data.

4.3 Servlets Must Be Non-Blocking

SIP and HTTP Servlets must not block threads in the body of a SIP method because the call state remains locked while the method is invoked. For example, no Servlet method must actively wait for data to be retrieved or written before returning control to the SIP Servlet container.

4.4 Store all Application Data in the Session

If you deploy your application to more than one engine tier server (in a replicated OWLCS configuration) you must store all application data in the session as session attributes. In a replicated configuration, engine tier servers maintain no cached information; all application data must be de-serialized from the session attribute available in SIP data tier servers.

4.5 All Session Data Must Be Serializable

To support in-memory replication of SIP application call states, you must ensure that all objects stored in the SIP Servlet session are serializable. Every field in an object must be serializable or transient in order for the object to be considered serializable. If the Servlet uses a combination of serializable and non-serializable objects, OWLCS cannot replicate the session state of the non-serializable objects.

4.6 Use setAttribute() to Modify Session Data in “No-Call” Scope

The SIP Servlet container automatically locks the associated call state when invoking the `doxxx` method of a SIP Servlet. However, applications may also attempt to modify session data in "no-call" scope. No-call scope refers to the context where call state data is modified outside the scope of a normal `doxxx` method. For example, data is modified in no-call scope when an HTTP Servlet attempts to modify SIP session data, or when a SIP Servlet attempts to modify a call state other than the one that the container locked before invoking the Servlet.

Applications must always use the SIP Session's `setAttribute` method to change attributes in no-call scope. Likewise, use `removeAttribute` to remove an attribute from a session object. Each time `setAttribute/removeAttribute` is used to update session data, the SIP Servlet container obtains and releases a lock on the associated call state. (The methods enqueue the object for updating, and return control immediately.) This ensures that only one application modifies the data at a time, and also ensures that your changes are replicated across SIP data tier nodes in a cluster.

If you use other set methods to change objects within a session, OWLCS cannot replicate those changes.

Note that the OWLCS container does not persist changes to a call state attribute that are made *after* calling `setAttribute`. For example, in the following code sample the `setAttribute` call immediately modifies the call state, but the subsequent call to `modifyState()` does not:

```
Foo foo = new Foo(..);
appSession.setAttribute("name", foo); // This persists the call state.
foo.modifyState(); // This change is not persisted.
```

Instead, ensure that your Servlet code modifies the call state attribute value *before* calling `setAttribute`, as in:

```
Foo foo = new Foo(..);
foo.modifyState();
appSession.setAttribute("name", foo);
```

Also, keep in mind that the SIP Servlet container obtains a lock to the call state for *each* individual `setAttribute` call. For example, when executing the following code in an HTTP Servlet, the SIP Servlet container obtains and releases a lock on the call state lock twice:

```
appSess.setAttribute("foo1", "bar2");
appSess.setAttribute("foo2", "bar2");
```

This locking behavior ensures that only one thread modifies a call state at any given time. However, another process could potentially modify the call state between sequential updates. The following code is not considered thread safe when done no-call state:

```
Integer oldValue = appSession.getAttribute("counter");
Integer newValue = incrementCounter(oldValue);
```

```
appSession.setAttribute("counter", newValue);
```

To make the above code thread safe, you must enclose it using the `wlssAppSession.doAction` method, which ensures that all modifications made to the call state are performed within a single transaction lock, as in:

```
wlssAppSession.doAction(new WlssAction() {
       public Object run() throws Exception {
         Integer oldValue = appSession.getAttribute("counter");
         Integer newValue = incrementCounter(oldValue);
         appSession.setAttribute("counter", newValue);
         return null;
       }
     });
```

Finally, be careful to avoid deadlock situations when locking call states in a "`do`*SipMethod*" call, such as `doInvite()`. Keep in mind that the OWLCS container has already locked the call state when the instructions of a `do`*SipMethod* are executed. If your application code attempts to access the current call state from within such a method (for example, by accessing a session that is stored within a data structure or attribute), the lock ordering results in a deadlock.

Example 4–1 shows an example that can result in a deadlock. If the code is executed by the container for a call associated with `callAppSession`, the locking order is reversed and the attempt to obtain the session with `getApplicationSession(callId)` causes a deadlock.

Example 4–1 Session Access Resulting in a Deadlock

```
WlssSipApplicationSession confAppSession = (WlssSipApplicationSession) appSession;
confAppSession.doAction(new WlssAction() {
  // confAppSession is locked
  public Object run() throws Exception {
    String callIds = confAppSession.getAttribute("callIds");
    for (each callId in callIds) {
      callAppSess = Session.getApplicationSession(callId);
      // callAppSession is locked
      attributeStr += callAppSess.getAttribute("someattrib");
    }
    confAppSession.setAttribute("attrib", attributeStr);
  }
}
```

See Section 6.3.1, "Modifying the SipApplicationSession" for more information about using the `com.bea.wcp.sip.WlssAction` interface.

4.7 send() Calls Are Buffered

If your SIP Servlet calls the `send()` method within a SIP request method such as `doInvite()`, `doAck()`, `doNotify()`, and so forth, keep in mind that the OWLCS container buffers all `send()` calls and transmits them in order *after* the SIP method returns. Applications cannot rely on `send()` calls to be transmitted immediately as they are called.

Caution: Applications must not wait or sleep after a call to `send()` because the request or response is not transmitted until control returns to the SIP Servlet container.

4.8 Mark SIP Servlets as Distributable

If you have designed and programmed your SIP Servlet to be deployed to a cluster environment, you must include the `distributable` marker element in the Servlet's deployment descriptor when deploying the application to a cluster of engine tier servers. If you omit the `distributable` element, OWLCS does not deploy the Servlet to a cluster of engine tier servers. If you mark `distributable` in sip.xml it must also be marked in the web.xml for a WAR file.

The `distributable` element is not required, and is ignored if you deploy to a single, combined-tier (non-replicated) OWLCS instance.

4.9 Use SipApplicationSessionActivationListener Sparingly

The SIP Servlet 1.1 specification introduces `SipApplicationSessionActivationListener`, which can provide callbacks to an application when SIP Sessions are passivated or activated. Keep in mind that callbacks occur only in a replicated OWLCS deployment. Single-server deployments use no SIP data tier, so SIP Sessions are never passivated.

Also, keep in mind that in a replicated deployment OWLCS activates and passivates a SIP Session many times, before and after SIP messages are processed for the session. (This occurs normally in any replicated deployment, even when RDBMS-based persistence is not configured.) Because this constant cycle of activation and passivation results in frequent callbacks, use `SipApplicationSessionActivationListener` sparingly in your applications.

4.10 Session Expiration Best Practices

For a JSR289 application, the container is more "intelligent" in removing sessions. For example, there is no need to explicity call invalidate() on a session or sipappsession.

However, if setExpirs() is used on a session and the application is of a JSR289 type then that call has no effect unless setInvalidateWhenRead(false) is called on the session.

4.11 Observe Best Practices for Java EE Applications

If you are deploying applications that use other Java EE APIs, observe the basic clustering guidelines associated with those APIs. For example, if you are deploying EJBs you must design all methods to be idempotent and make EJB homes clusterable in the deployment descriptor. See "Clustering Best Practices" for more information.

5 Composing SIP Applications

This chapter describes how to use OWLCS application composition features, in the following sections:

Note: The SIP Servlet v1.1 specification (`http://jcp.org/en/jsr/detail?id=289`) describes a formal application selection and composition process, which is fully implemented in OWLCS. Use the SIP Servlet v1.1 techniques, as described in this document, for all new development. Application composition techniques described in earlier versions of OWLCS are now deprecated.

OWLCS provides backwards compatibility for applications using version 1.0 composition techniques, provided that:

- you *do not* configure a custom Application Router, and
- you *do not* configure Default Application Router properties.

5.1 Application Composition Model

Application composition is the process of "chaining" multiple SIP applications into a logical path to apply services to a SIP request. The SIP Servlet v1.1 specification introduces an Application Router (AR) deployment, which performs a key role in composing SIP applications. The Application Router examines an initial SIP request and uses custom logic to determine which SIP application must process the request. In OWLCS, all initial requests are first delivered to the AR, which determines the application used to process the request.

OWLCS provides a default Application Router, which can be configured using a text file. However, most installations can develop and deploy a custom Application Router by implementing the `SipApplicationRouter` interface. A custom Application Router enables you to consult data stores when determining which SIP application must handle a request.

In contrast to the Application Router, which requires knowledge of which SIP applications are available for processing a message, individual SIP applications remain independent from one another. An individual application performs a very specific service for a SIP request, without requiring any knowledge of other applications deployed on the system. (The Application Router does require knowledge of deployed applications, and the `SipApplicationRouter` interface provides for automatic notification of application deployment and undeployment.)

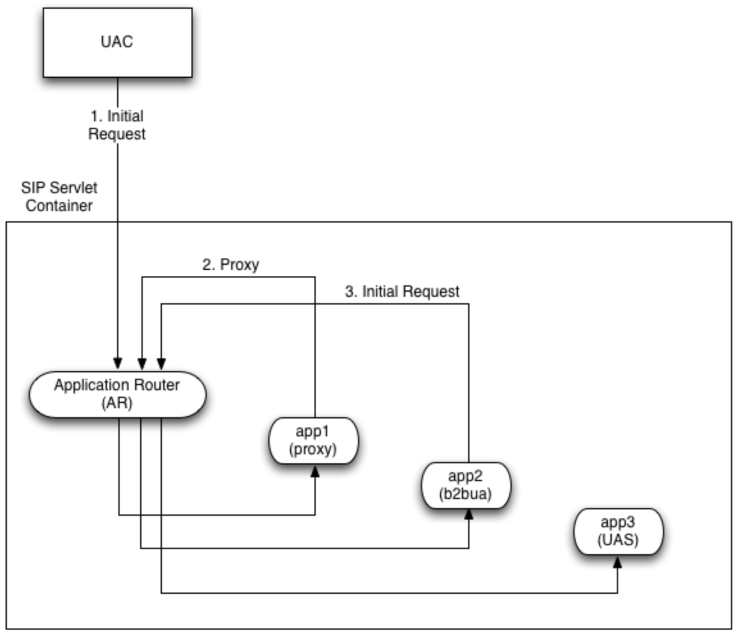
Individual SIP applications may complete their processing of an initial request by proxying or relaying the request, or by terminating the request as a User Agent Server (UAS). If an initial request is proxied or relayed, the SIP container again forwards the request to the Application Router, which selects the next SIP application to provide a service for the request. In this way, the AR can chain multiple SIP applications as needed to process a request. The chaining process is terminated when:

- a selected SIP application acts as a UAS to terminate the chain, or
- there are no more applications to select for that request. (In this case, the request is sent out.)

When the chain is terminated and the request sent, the SIP container maintains the established path of applications for processing subsequent requests, and the AR is no longer consulted.

Figure 5–1 shows the use of an Application Router for applying multiple service to a SIP request.

Figure 5–1 Composed Application Model



Note that the AR may select remote as well as local applications; the chain of services need not reside within the same OWLCS container.

5.2 Using the Default Application Router

OWLCS includes a Default Application Router (DAR) having the basic functionality described in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`), Appendix C: Default Application Router. In summary, the OWLCS DAR implements all methods of the

`SipApplicationRouter` interface, and is configured using the simple Java properties file described in the v1.1 specification.

Each line of the DAR properties file specifies one or more SIP methods, and is followed by SIP routing information in comma-delimited format. The DAR initially reads the properties file on startup, and then reads it each time a SIP application is deployed or undeployed from the container.

To specify the location of the configuration file used by the DAR, configure the properties using the Administration Console, as described in "Configuring a Custom Application Router", or include the following parameter when starting the OWLCS instance:

```
-Djavax.servlet.sip.ar.dar.configuration
```

(To specify a property file, rather than a URI, include the prefix `file:///`) This Java parameter is specified at the command line, therefore it can be included in your server startup script.

See Appendix C in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for detailed information about the format of routing information used by the Default Application Router.

Note that the OWLCS DAR accepts route region strings in addition to "originating," "terminating," and "neutral." Each new string value is treated as an extended route region. Also, the OWLCS DAR uses the order of properties in the configuration file to determine the route entry sequence; the `state_info` value has no effect when specified in the DAR configuration.

5.3 Configuring a Custom Application Router

By default OWLCS uses its DAR implementation.

If you develop a custom Application Router, you must store the implementation for the AR in the `/approuter` subdirectory of the domain home directory. Supporting libraries for the AR can be stored in a `/lib` subdirectory within `/approuter`. (If you have multiple implementations of `SipApplicationRouter`, use the `-Djavax.servlet.sip.ar.spi.SipApplicationRouterProvider` option at startup to specify which one to use.)

Note: In a clustered environment, the custom AR is deployed to all engine tier instances of the domain; you cannot deploy different AR implementations within the same domain.

See Section 15 in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for more information about the function of the AR. See also the SIP Servlet v1.1 API for information about how to implement a custom AR.

5.4 Session Key-Based Request Targeting

The SIP Servlet v1.1 specification also provides a mechanism for associating an initial request with an existing `SipApplicationSession` object. This mechanism is called session key-based targeting. Session key-based targeting is used to direct initial requests having a particular subscriber (request URI) or region, or other feature to an already-existing `SipApplicationSession`, rather than generating a new session. To use this targeting mechanism with an application, you create a method that generates a unique key and annotate that method with `@SipApplicationKey`. When the SIP container selects that application (for example, as a result of the AR choosing it for an initial request), it obtains a key using the annotated method, and uses the key and application name to determine if the `SipApplicationSession` exists. If one exists, the container associates the new request with the existing session, rather than generating a new session.

Note: If you develop a spiral proxy application using this targeting mechanism, and the application modifies the record-route more than once, it must generate different keys for the initial request, if necessary, when processing record-route hops. If it does not, then the application cannot discriminate record-route hops for subsequent requests.

See section 15 in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for more information about using session key-based targeting.

6 Securing SIP Servlet Resources

The chapter describes how to apply security constraints to SIP Servlet resources when deploying to OWLCS, in the following sections:

6.1 Overview of SIP Servlet Security

The SIP Servlet API specification defines a set of deployment descriptor elements that can be used for providing declarative and programmatic security for SIP Servlets. The primary method for declaring security constraints is to define one or more `security-constraint` elements in the `sip.xml` deployment descriptor. The `security-constraint` element defines the actual resources in the SIP Servlet, defined in `resource-collection elements`, that are to be protected. `security-constraint` also identifies the role names that are authorized to access the resources. All role names used in the `security-constraint` are defined elsewhere in `sip.xml` in a `security-role` element.

SIP Servlets can also programmatically refer to a role name within the Servlet code, and then map the hard-coded role name to an alternate role in the `sip.xml` `security-role-ref` element during deployment. Roles must be defined elsewhere in a `security-role` element before they can be mapped to a hard-coded name in the `security-role-ref` element.

The SIP Servlet specification also enables Servlets to propagate a security role to a called Enterprise JavaBean (EJB) using the `run-as` element. Once again, roles used in the `run-as` element must be defined in a separate `security-role` element in `sip.xml`.

Chapter 14 in the SIP Servlet API specification provides more details about the types of security available to SIP Servlets. SIP Servlet security features are similar to security features available with HTTP Servlets; you can find additional information about HTTP Servlet security by referring to these sections in the Oracle WebLogic Communication Services documentation:

- Securing Web Applications in *Programming WebLogic Security* provides an overview of declarative and programmatic security models for Servlets.
- EJB Security-Related Deployment Descriptors in *Securing Enterprise JavaBeans (EJBs)* describes all security-related deployment descriptor elements for EJBs, including the `run-as` element used for propagating roles to called EJBs.

See also the example `sip.xml` excerpt in Example 6–1, "Declarative Security Constraints in sip.xml".

6.2 Triggering SIP Response Codes

You can distinguish whether you are a proxy application, or a UAS application, by configuring the container to trigger the appropriate SIP response code, either a 401 SIP response code, or a 407 SIP response code. If your application needs to proxy an invitation, the 407 code is appropriate to use. If your application is a registrar application, you must use the 401 code.

To configure the container to respond with a 407 SIP response code instead of a 401 SIP response code, you must add the `<proxy-authentication>` element to the security constraint.

6.3 Specifying the Security Realm

You must specify the name of the current security realm in the sip.xml file as follows:

```
<login-config>
<auth-method>DIGEST</auth-method>
<realm-name>myrealm</realm-name>
</login-config>
```

6.4 Role Mapping Features

When you deploy a SIP Servlet, `security-role` definitions that were created for declarative and programmatic security must be assigned to actual principals and/or roles available in the Servlet container. OWLCS uses the `security-role-assignment` element in `weblogic.xml` to help you map `security-role` definitions to actual principals and roles. `security-role-assignment` provides two different ways to map security roles, depending on how much flexibility you require for changing role assignment at a later time:

- The `security-role-assignment` element can define the complete list of principal names and roles that map to roles defined in `sip.xml`. This method defines the role assignment at deployment time, but at the cost of flexibility; to add or remove principals from the role, you must edit `weblogic.xml` and redeploy the SIP Servlet.
- The `externally-defined` element in `security-role-assignment` enables you to assign principal names and roles to a `sip.xml` role at any time using the Administration Console. When using the `externally-defined` element, you can add or remove principals and roles to a `sip.xml` role without having to redeploy the SIP Servlet.

Two additional XML elements can be used for assigning roles to a `sip.xml` `run-as` element: `run-as-principal-name` and `run-as-role-assignment`. These role assignment elements take precedence over `security-role-assignment` elements if they are used, as described in "Assigning run-as Roles".

Optionally, you can choose to specify no role mapping elements in `weblogic.xml` to use implicit role mapping, as described in "Using Implicit Role Assignment".

The sections that follow describe OWLCS role assignment in more detail.

6.5 Using Implicit Role Assignment

With implicit role assignment, OWLCS assigns a `security-role` name in `sip.xml` to a role of the exact same name, which must be configured in the OWLCS security realm. To use implicit role mapping, you omit the `security-role-assignment` element in `weblogic.xml`, as well as any `run-as-principal-name`, and `run-as-role-assignment` elements use for mapping `run-as` roles.

When no role mapping elements are available in `weblogic.xml`, OWLCS implicitly maps `sip.xml` `security-role` elements to roles having the same name. Note that implicit role mapping takes place regardless of whether the role name defined in `sip.xml` is actually available in the security realm. OWLCS display a warning message anytime it uses implicit role assignment. For example, if you use the "everyone" role in `sip.xml` but you do not explicitly assign the role in `weblogic.xml`, the server displays the warning:

```
<Webapp: ServletContext(id=id,name=application,context-path=/context), the role:
everyone defined in web.xml has not been mapped to principals in
security-role-assignment in weblogic.xml. Will use the rolename itself as the
principal-name.>
```

You can ignore the warning message if the corresponding role has been defined in the OWLCS security realm. The message can be disabled by defining an explicit role mapping in `weblogic.xml`.

Use implicit role assignment if you want to hard-code your role mapping at deployment time to a known principal name.

6.6 Assigning Roles Using security-role-assignment

The `security-role-assignment` element in `weblogic.xml` enables you to assign roles either at deployment time or at any time using the Administration Console. The sections that follow describe each approach.

6.6.1 Important Requirements

If you specify a `security-role-assignment` element in `weblogic.xml`, OWLCS requires that you also define a duplicate `security-role` element in a `web.xml` deployment descriptor. This requirement applies even if you are deploying a pure SIP Servlet, which would not normally require a `web.xml` deployment descriptor (generally reserved for HTTP Web Applications).

Note: If you specify a security-role-assignment in weblogic.xml but there is no corresponding security-role element in web.xml, OWLCS generates the error message:

```
The security-role-assignment references an invalid security-role:
rolename
```

The server then implicitly maps the security-role defined in sip.xml to a role of the same name, as described in "Using Implicit Role Assignment".

For example, Example 6–1 shows a portion of a `sip.xml` deployment descriptor that defines a security constraint with the role, `roleadmin`. Example 6–2 shows that a `security-role-assignment` element has been defined in `weblogic.xml` to assign principals and roles to `roleadmin`. In OWLCS, this Servlet *must* be deployed with a `web.xml` deployment descriptor that also defines the `roleadmin` role, as shown in Example 6–3.

If the `web.xml` contents were not available, OWLCS would use implicit role assignment and assume that the `roleadmin` role was defined in the security realm; the principals and roles assigned in `weblogic.xml` would be ignored.

Example 6–1 Declarative Security Constraints in sip.xml

```
...
  <security-constraint>
      <resource-collection>
      <resource-name>RegisterRequests</resource-name>
      <servlet-name>registrar</servlet-name>
    </resource-collection>
    <auth-constraint>
      <javaee:role-name>roleadmin</javaee:role-name>
    </auth-constraint>
  </security-constraint>

  <security-role>
    <javaee:role-name>roleadmin</javaee:role-name>
  </security-role>
...
```

Example 6–2 Example security-role-assignment in weblogic.xml

```
<weblogic-web-app>
  <security-role-assignment>
       <role-name>roleadmin</role-name>
       <principal-name>Tanya</principal-name>
       <principal-name>Fred</principal-name>
       <principal-name>system</principal-name>
  </security-role-assignment>
</weblogic-web-app>
```

Example 6–3 Required security-role Element in web.xml

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <security-role>
    <role-name>roleadmin</role-name>
  </security-role>
</web-app>
```

6.6.2 Assigning Roles at Deployment Time

A basic `security-role-assignment` element definition in `weblogic.xml` declares a mapping between a `security-role` defined in `sip.xml` and one or more principals or roles available in the OWLCS security realm. If the `security-role` is used in combination with the `run-as` element in `sip.xml`, OWLCS assigns the first principal or role name specified in the `security-role-assignment` to the `run-as` role.

Example 6–2, "Example security-role-assignment in weblogic.xml" shows an example `security-role-assignment` element. This example assigns three users to the `roleadmin` role defined in Example 6–1, "Declarative Security Constraints in sip.xml". To change the role assignment, you must edit the `weblogic.xml` descriptor and redeploy the SIP Servlet.

6.6.3 Dynamically Assigning Roles Using the Administrative Console

The `externally-defined` element can be used in place of the `<principal-name>` element to indicate that you want the security roles defined in the `role-name` element of `sip.xml` to use mappings that you assign in the Administration Console. The `externally-defined` element gives you the flexibility of not having to specify a specific security role mapping for each security role at deployment time. Instead, you can use the Administration Console to specify and modify role assignments at anytime.

Additionally, because you may elect to use this element for some SIP Servlets and not others, it is not necessary to select the **ignore roles and polices from DD** option for the security realm. (You select this option in the **On Future Redeploys:** field on the **General** tab of the **Security->Realms->myrealm** control panel on the Administration Console.) Therefore, within the same security realm, deployment descriptors can be used to specify and modify security for some applications while the Administration Console can be used to specify and modify security for others.

Note: When specifying security role names, observe the following conventions and restrictions:

- The proper syntax for a security role name is as defined for an Nmtoken in the Extensible Markup Language (XML) recommendation available on the Web at: `http://www.w3.org/TR/REC-xml#NT-Nmtoken`.
- Do not use blank spaces, commas, hyphens, or any characters in this comma-separated list: \t, < >, #, |, &, ~, ?, ( ), { }.
- Security role names are case sensitive.
- The Oracle-suggested convention for security role names is that they be singular.

Example 6–4 shows an example of using the `externally-defined` element with the `roleadmin` role defined in Example 6–1, "Declarative Security Constraints in sip.xml". To assign existing principals and roles to the `roleadmin` role, the Administrator would use the OWLCS Administration Console.

See the "Users, Groups, and Security Roles" for information about adding and modifying security roles using the Administration Console.

Example 6–4 Example externally-defined Element in weblogic.xml

```
<weblogic-web-app>
     <security-role-assignment>
         <role-name>webuser</role-name>
         <externally-defined/>
     </security-role-assignment>
</weblogic-web-app>
```

6.7 Assigning run-as Roles

The `security-role-assignment` described in "Assigning Roles Using security-role-assignment" can be also be used to map `run-as` roles defined in `sip.xml`. Note, however, that two additional elements in `weblogic.xml` take precedence over the `security-role-assignment` if they are present: `run-as-principal-name` and `run-as-role-assignment`.

`run-as-principal-name` specifies an existing principle in the security realm that is used for all `run-as` role assignments. When it is defined within the `servlet-descriptor` element of `weblogic.xml`, `run-as-principal-name` takes precedence over any other role assignment elements for `run-as` roles.

`run-as-role-assignment` specifies an existing role or principal in the security realm that is used for all `run-as` role assignments, and is defined within the `weblogic-web-app` element.

See "weblogic.xml Deployment Descriptor Reference" for more information about individual `weblogic.xml` descriptor elements. See also "Role Assignment Precedence for SIP Servlet Roles" for a summary of the role mapping precedence for declarative and programmatic security as well as `run-as` role mapping.

6.8 Role Assignment Precedence for SIP Servlet Roles

OWLCS provides several ways to map `sip.xml` roles to actual roles in the SIP Container during deployment. For declarative and programmatic security defined in `sip.xml`, the order of precedence for role assignment is:

1. If `weblogic.xml` assigns a `sip.xml` role in a `security-role-assignment` element, the `security-role-assignment` is used.

   Note: OWLCS also requires a role definition in web.xml in order to use a security-role-assignment. See "Important Requirements".

2. If no `security-role-assignment` is available (or if the required `web.xml` role assignment is missing), implicit role assignment is used.

For `run-as` role assignment, the order of precedence for role assignment is:

1. If `weblogic.xml` assigns a `sip.xml` `run-as` role in a `run-as-principal-name` element defined within `servlet-descriptor`, the `run-as-principal-name` assignment is used.

Note: OWLCS also requires a role definition in web.xml in order to assign roles with run-as-principal-name. See "Important Requirements".

2. If `weblogic.xml` assigns a `sip.xml` `run-as` role in a `run-as-role-assignment` element, the `run-as-role-assignment` element is used.

   Note: OWLCS also requires a role definition in web.xml in order to assign roles with run-as-role-assignment. See "Important Requirements"

3. If `weblogic.xml` assigns a `sip.xml run-as` role in a `security-role-assignment` element, the `security-role-assignment` is used.

   Note: OWLCS also requires a role definition in web.xml in order to use a security-role-assignment. See "Important Requirements".

4. If no `security-role-assignment` is available (or if the required `web.xml` role assignment is missing), implicit role assignment is used.

6.9 Debugging Security Features

If you want to debug security features in SIP Servlets that you develop, specify the `-Dweblogic.Debug=wlss.Security startup` option when you start OWLCS. Using this debug option causes OWLCS to display additional security-related messages in the standard output.

6.10 weblogic.xml Deployment Descriptor Reference

The `weblogic.xml` DTD contains detailed information about each of the role mapping elements discussed in this section. See "weblogic.xml Deployment Descriptor Elements" for more information.

7 Enabling Message Logging

This chapter describes how to use message logging features on a development system, in the following sections:

- Section 7.1, "Overview"
- Section 7.2, "Enabling Message Logging"
- Section 7.3, "Specifying Content Types for Unencrypted Logging"
- Section 7.4, "Example Message Log Configuration and Output"
- Section 7.5, "Configuring Log File Rotation"

7.1 Overview

Message logging records SIP and Diameter messages (both requests and responses) received by OWLCS. This requires that the logging level be set to at least the INFO level. You can use the message log in a development environment to check how external SIP requests and SIP responses are received. By outputting the distinguishable information of SIP dialogs such as Call-IDs from the application log, and extracting relevant SIP messages from the message log, you can also check SIP invocations from HTTP servlets and so forth.

When you enable message logging, OWLCS records log records in the Managed Server log file associated with each engine tier server instance by default. You can optionally log the messages in a separate, dedicated log file, as described in "Configuring Log File Rotation".

7.2 Enabling Message Logging

You enable and configure message logging by adding a `message-debug` element to the `sipserver.xml` configuration file. OWLCS provides two different methods of configuring the information that is logged:

- Specify a predefined logging level (terse, basic, or full), or
- Identify the exact portions of the SIP message that you want to include in a log record, in a specified order

The sections that follow describe each method of configuring message logging functionality using elements in the `sipserver.xml` file. Note that you can also set these elements using the Administration Console, in the Configuration->Message Debug tab of the SipServer console extension node.

7.2.1 Specifying a Predefined Logging Level

The optional `level` element in `message-debug` specifies a predefined collection of information to log for each SIP request and response. The following levels are supported:

- `terse`—Logs only the `domain` setting, logging Servlet name, logging `level`, and whether or not the message is an incoming message.
- `basic`—Logs the `terse` items plus the SIP message status, reason phrase, the type of response or request, the SIP method, the **From** header, and the **To** header.
- `full`—Logs the `basic` items plus all SIP message headers plus the timestamp, protocol, request URI, request type, response type, content type, and raw content.

Example 7–1 shows a configuration entry that specifies the `full` logging level.

Example 7–1 Sample Message Logging Level Configuration in sipserver.xml

```
<message-debug>
   <level>full</level>
</message-debug>
```

7.2.2 Customizing Log Records

OWLCS also enables you to customize the exact content and order of each message log record. To configure a custom log record, you provide a `format` element that defines a log record `pattern` and one or more `tokens` to log in each record.

Note: If you specify a format element with a <level>full</level> level element undefined) in message-debug, OWLCS uses "full" message debugging and ignores the format entry. The format entry can be used in combination with either the "terse" or "basic" message-debug levels.

Table 7–1 describes the nested elements used in the `format` element.

Table 7–1 Nested format Elements

param-name	param-value Description
`pattern`	Specifies the pattern used to format a message log entry. The format is defined by specifying one or more integers, bracketed by "{" and "}". Each integer represents a `token` defined later in the `format` definition.
`token`	A string token that identifies a portion of the SIP message to include in a log record. Table 7–2 provides a list of available string tokens. You can define multiple `token` elements as needed to customize your log records.

Table 7–2 describes the string `token` values used to specify information in a message log record:

Table 7–2 Available Tokens for Message Log Records

Token	Description	Example or Type
%call_id	The Call-ID header. It is blank when forwarding.	43543543
%content	The raw content.	Byte array
%content_length	The content length.	String value
%content_type	The content type.	String value
%cseq	The CSeq header. It is blank when forwarding.	INVITE 1
%date	The date when the message was received. ("yyyy/MM/dd" format)	2004/05/16
%from	The From header (all). It is blank when forwarding.	sip:foo@oracle.com;tag=438943
%from_addr	The address portion of the From header.	foo@oracle.com
%from_tag	The tag parameter of the From header. It is blank when forwarding.	12345
%from_uri	The SIP URI part of the From header. It is blank when forwarding.	sip:foo@oracle.com
%headers	A List of message headers stored in a 2-element array. The first element is the name of the header, while the second is a list of all values for the header.	List of headers
%io	Whether the message is incoming or not.	TRUE
%method	The name of the SIP method. It records the method name to invoke when forwarding.	INVITE
%msg	Summary Call ID	String value
%mtype	The type of receiving.	SIPREQ
%protocol	The protocol used.	UDP
%reason	The response reason.	OK
%req_uri	The request URI. This token is only available for the SIP request.	sip:foo@oracle.com
%status	The response status.	200
%time	The time when the message was received. ("HH:mm:ss" format)	18:05:27
%timestampmillis	Time stamp in milliseconds.	9295968296
%to	The To header (all). It is blank when forwarding.	sip:foo@oracle.com;tag=438943
%to_addr	The address portion of the To header.	foo@oracle.com
%to_tag	The tag parameter of the To header. It is blank when forwarding.	12345
%to_uri	The SIP URI part of the To header. It is blank when forwarding.	sip:foo@oracle.com

See "Example Message Log Configuration and Output" for an example `sipserver.xml` file that defines a custom log record using two tokens.

7.3 Specifying Content Types for Unencrypted Logging

By default OWLCS uses String format (UTF-8 encoding) to log the content of SIP messages having a text or application/sdp Content-Type value. For all other Content-Type values, OWLCS attempts to log the message content using the character set specified in the `charset` parameter of the message, if one is specified. If no `charset` parameter is specified, or if the `charset` value is invalid or unsupported, OWLCS uses Base-64 encoding to encrypt the message content before logging the message.

If you want to avoid encrypting the content of messages under these circumstances, specify a list of String-representable Content-Type values using the `string-rep` element in `sipserver.xml`. The `string-rep` element can contain one or more `content-type` elements to match. If a logged message matches one of the configured `content-type` elements, OWLCS logs the content in String format using UTF-8 encoding, regardless of whether or not a `charset` parameter is included.

Note: You do not need to specify text/* or application/sdp content types as these are logged in String format by default.

Example 7–2 shows a sample `message-debug` configuration that logs String content for three additional Content-Type values, in addition to text/* and application/sdp content.

Example 7–2 Logging String Content for Additional Content Types

```
<message-debug>
  <level>full</level>
  <string-rep>
    <content-type>application/msml+xml</content-type>
    <content-type>application/media_control+xml</content-type>
    <content-type>application/media_control</content-type>
  </string-rep>
</message-debug>
```

7.4 Example Message Log Configuration and Output

Example 7–3 shows a sample message log configuration in `sipserver.xml`. Example 7–4, "Sample Message Log Output" shows sample output from the Managed Server log file.

Example 7–3 Sample Message Log Configuration in sipserver.xml

```
<message-debug>
   <format>
      <pattern>{0} {1}</pattern>
      <token>%headers</token>
      <token>%content</token>
   </format>
</message-debug>
```

Example 7–4 Sample Message Log Output

```
####<Aug 10, 2005 7:12:08 PM PDT> <Info> <WLSS.Trace> <jiri.bea.com> <myserver>
<ExecuteThread: '11' for queue: 'sip.transport.Default'> <<WLS Kernel>> <> <BEA-
331802> <SIP Tracer: logger Message: To: sut <sip:invite@10.32.5.230:5060>
<mailto:sip:invite@10.32.5.230:5060>
```

```
Content-Length: 136
Contact: user:user@10.32.5.230:5061
CSeq: 1 INVITE
Call-ID: 59.3170.10.32.5.230@user.call.id
From: user <sip:user@10.32.5.230:5061> <mailto:sip:user@10.32.5.230:5061> ;tag=59
Via: SIP/2.0/UDP 10.32.5.230:5061
Content-Type: application/sdp
Subject: Performance Test
Max-Forwards: 70
 v=0
o=user1 53655765 2353687637 IN IP4 127.0.0.1
s=-
c=IN IP4       127.0.0.1
t=0 0
m=audio 10000 RTP/AVP 0
a=rtpmap:0 PCMU/8000
>
####<Aug 10, 2005 7:12:08 PM PDT> <Info> <WLSS.Trace> <jiri.bea.com> <myserver>
<ExecuteThread: '11' for queue: 'sip.transport.Default'> <<WLS Kernel>> <> <BEA-
331802> <SIP Tracer: logger Message: To: sut <sip:invite@10.32.5.230:5060>
<mailto:sip:invite@10.32.5.230:5060>
Content-Length: 0
CSeq: 1 INVITE
Call-ID: 59.3170.10.32.5.230@user.call.id
Via: SIP/2.0/UDP 10.32.5.230:5061
From: user <sip:user@10.32.5.230:5061> <mailto:sip:user@10.32.5.230:5061> ;tag=59
Server: Oracle WebLogic Communications Server 10.3.1.0
 >
```

7.5 Configuring Log File Rotation

Message log entries for SIP and Diameter messages are stored in the main OWLCS log file by default. You can optionally store the messages in a dedicated log file. Using a separate file makes it easier to locate message logs, and also enables you to use OWLCS's log rotation features to better manage logged data.

Log rotation is configured using several elements nested within the main `message-debug` element in `sipserver.xml`. As with the other XML elements described in this section, you can also configure values using the Configuration->Message Debug tab of the SIP Server Administration Console extension.

Table 7–3 describes each element. Note that a server restart is necessary in order to initiate independent logging and log rotation.

Table 7–3 XML Elements for Configuring Log Rotation

Element	Description
`logging-enabled`	Determines whether a separate log file is used to store message debug log messages. By default, this element is set to false and messages are logged in the general log file.
`file-min-size`	Configures the minimum size, in kilobytes, after which the server automatically rotate log messages into another file. This value is used when the `rotation-type` element is set to `bySize`.
`log-filename`	Defines the name of the log file for storing messages. By default, the log files are stored under *`domain_home`*`/servers/`*`server_name`*`/logs`.

Table 7–3 (Cont.) XML Elements for Configuring Log Rotation

Element	Description
`rotation-type`	Configures the criterion for moving older log messages to a different file. This element may have one of the following values: ▪ `bySize`—This default setting rotates log messages based on the specified `file-min-size`. ▪ `byTime`—This setting rotates log messages based on the specified `rotation-time`. ▪ `none`—Disables log rotation.
`number-of-files-limited`	Specifies whether or not the server places a limit on the total number of log files stored after a log rotation. By default, this element is set to false.
`file-count`	Configures the maximum number of log files to keep when `number-of-files-limited` is set to true.
`rotate-log-on-startup`	Determines whether the server must rotate the log file at server startup time.
`log-file-rotation-dir`	Configures a directory in which to store rotated log files. By default, rotated log files are stored in the same directory as the active log file.
`rotation-time`	Configures a start time for log rotation when using the `byTime` log rotation criterion.
`file-time-span`	Specifies the interval, in hours, after which the log file is rotated. This value is used when the `rotation-type` element is set to `byTime`.
`date-format-pattern`	Specifies the pattern to use for rending dates in log file entries. The value of this element must conform to the `java.text.SimpleDateFormat` class.

Example 7–5 shows a sample `message-debug` configuration using log rotation.

Example 7–5 Sample Log Rotation Configuration

```
<?xml version='1.0' encoding='UTF-8'?>
<sip-server xmlns="http://www.bea.com/ns/wlcp/wlss/300"
xmlns:sec="http://www.bea.com/ns/weblogic/90/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wls="http://www.bea.com/ns/weblogic/90/security/wls">
  <message-debug>
    <logging-enabled>true</logging-enabled>
    <file-min-size>500</file-min-size>
    <log-filename>sip-messages.log</log-filename>
    <rotation-type>byTime</rotation-type>
    <number-of-files-limited>true</number-of-files-limited>
    <file-count>5</file-count>
    <rotate-log-on-startup>false</rotate-log-on-startup>
    <log-file-rotation-dir>old_logs</log-file-rotation-dir>
    <rotation-time>00:00</rotation-time>
    <file-time-span>20</file-time-span>
    <date-format-pattern>MMM d, yyyy h:mm a z</date-format-pattern>
  </message-debug>
</sip-server>
```

Part III

Parlay X Web Services and Multimedia Messaging

This part describes developing applications using the Parlay X Web Services and Parlay X Multimedia Messaging API.

Part IV contains the following chapters:

8 Parlay X Presence Web Services

This chapter describes support for the Parlay X 2.0 Presence Web services interfaces for developing applications. The Web service functions as a Presence Network Agent which can publish, subscribe, and listen for notifications on behalf of the users of the Web service. This chapter contains the following sections:

8.1 Introduction

OWLCS provides support for Part 14 of the Parlay X Presence Web Service as defined in the *Open Service Access, Parlay X Presence Web Services, Part 14, Presence ETSI ES 202 391-14* specification. The OWLCS Parlay X Web service maps the Parlay X Web service to a SIP/IMS network according to the Open Service Access, Mapping of Parlay X Presence Web Services to Parlay/OSA APIs, Part 14, Presence Mapping, Subpart 2, Mapping to SIP/IMS Networks, ETSI TR 102 397-14-2 specification.

Note: Due to the synchronous nature of the Web service, to receive a callback from the Web service the client must implement the Web service callback interface. For presence, the required interface is the `PresenceNotification` interface described in *Open Service Access, Parlay X Presence Web Services, Part 14, Presence ETSI ES 202 391-14.*

The HTTP server that hosts the presence Web service is a Presence Network Agent or a Parlay X to SIP gateway.

8.2 Installing the Web Services

The Web services are packaged as a standard .ear file and can be deployed the same as any other Web services through Admin Console. The .ear file contains two .war files that implement the two interfaces. The Web services use the Oracle SDP Platform, Client and Presence Commons shared libraries.

8.3 Configuring Web Services

The following four mbean attributes are configurable for the Presence Supplier Web service:

- SIPOutboundProxy: SipURI of the outbound proxy for SIP message. Empty string means no outbound proxy. For example, sip:127.0.0.1:5060; lr;transport=tcp.
- PublicXCAPRootUrl: URI where the Presence Server is deployed. This attribute is used to update the presence rules stored on the XDMS. Example: http://127.0.0.1:8001/services/
- Expires: Set the time in seconds after which the PUBLISH by a presentity expires. Default value is 3600 (that is, 1 hour).
- SessionTimeout: Set the time in seconds after which HTTP sessions times out. Data for all timed out sessions is discarded.

For Presence Consumer, there are three mbean attributes that can be configured.

- SIPOutboundProxy: SipURI of the outbound proxy for SIP message. Empty string means no outbound proxy. For example, sip:127.0.0.1:5060; lr;transport=tcp.
- Expires: Set the time in seconds after which the SUBSCRIBE by a watcher expires. Default value is 3600 (ie. 1 hour).
- SessionTimeout: Set the time in seconds after which HTTP sessions times out. Data for all timed out sessions is discarded.

8.4 Presence Web Services Interface Descriptions

The presence Web services consist of three interfaces:

- PresenceConsumer: The watchers use these methods to obtain presence data (Table 8–1).
- PresenceNotification: The presence consumer interface uses the client callback defined in this interface to send notifications (Table 8–2).
- PresenceSupplier: The presentity uses these methods to publish presence data and manage access to the data by its watchers (Table 8–3).

Table 8–1 PresenceConsumer Interface

Operation	Description
subscribePresence	The Web Service sends a SUBSCRIBE to the presence server.
getUserPresence	Returns the cached presence status because the status changes of the presentity are asynchronously sent to the Web services through a SIP NOTIFY. The Web services actually have the subscription, not the Web services client.
startPresenceNotification	Enables the Web service client from receiving asynchronous notifications whenever a presentity makes change to its presence status, or presence rules document.
endPresenceNotification	Disables the web service client to receive asynchronous notifications.

Table 8–2 PresenceNotification Interface

Operation	Description
statusChanged	The asynchronous operation is called by the Web Service when an attribute for which notifications were requested changes.
statusEnd	This method is called when the duration for the notifications (identified by the correlator) is over. In case of an error or explicit call to endPresenceNotification, this method is not called.
notifySubscription	This asynchronous method notifies the watcher that the presentity handled the pending subscription.
subscriptionEnded	This asynchronous operation is called by the Web Service to notify the watcher that the subscription has terminated.

Table 8–3 PresenceSupplier Interface

Operation	Description
publish	Maps directly to a SIP PUBLISH.
getOpenSubscriptions	Called by the presentity (supplier) to check if any watcher wants to subscribe to its presence data. No SIP message maps to this method. Returns pending subscriptions currently in the Web service server.
updateSubscriptionAuthorization	The supplier uses this method to answer any open pending subscriptions. An XCAP PUT message is sent to the XDMS server to update the presence-rule document.
getMyWatchers	Retrieves the local list of watchers from the Web service server.
getSubscribedAttributes	Retrieves the local list of subscribed attributes from the Web service server. Currently, only returns *Activity*.
blockSubscription	Causes the Web service server to end a watcher subscription by modifying the XCAP document on the XDMS server (that is, putting the watcher on the block list).

8.5 Using the Presence Web Services Interfaces

This section describes how to use each of the operations in the interfaces, and includes code examples.

8.5.1 Interface: PresenceConsumer, Operation: subscribePresence

This is the first operation the application must call before using another operation in this interface. It serves two purposes:

- It allows the Web services to associate the current HTTP session with a user.
- It provides a context for all the other operations in this interface by subscribing to at least one presentity (SUBSCRIBE presence event).

8.5.1.1 Code Example

```
// Setting the attribute to activity
PresenceAttributeType pa = PresenceAttributeType.ACTIVITY;
List<PresenceAttributeType> pat = new ArrayList<PresenceAttributeType>();
pat.add(pa);
```

```
SimpleReference sr = new SimpleReference();
sr.setCorrelator("");
sr.setInterfaceName("");
sr.setEndpoint("");
consumer.subscribePresence ("sip.presentity@test.example.com" , pat, "unused",
sr);
```

8.5.2 Interface: PresenceConsumer, Operation: getUserPresence

Call this operation to retrieve a subscribed presentity presence. If the person is offline, it returns `ActivityNone` and the hardstate note is written to `PresenceAttribute.note`. If it returns `Activity_Other`, the description of the activity is returned in the `OtherValue` field.

If the `Name` field is equal to "ServiceAndDeviceNote", `OtherValue` is a combination of the service note and the device note. Note that there can be more than one "ServiceAndDeviceNote" when the presentity is logged into multiple clients.

8.5.2.1 Code Example

```
PresenceAttributeType pat =
  PresenceAttributeType.ACTIVITY;
List<PresenceAttribute> result =
  consumer.getUserPresence(presentity, pat);
for (PresenceAttribute pa : result) {
  // Check to see if it is an activity type.
  if (pa.getTypeAndValue().getUnionElement() ==
            PresenceAttributeType.ACTIVITY){
     // Get the presence status.
     System.out.println("ACTIVITY: " +
         pa.getTypeAndValue().getActivity().toString());
     // Get the customized presence note.
     if (pa.getNote().length() > 0){
        System.out.println("Note: " + pa.getNote());
    }
  }
  // If this is of type OTHER, then we need to extract
  // different type of information.
  if (pa.getTypeAndValue().getUnionElement() ==
            PresenceAttributeType.OTHER){
    // This is "Activity_Other", a custom presence status.
    if (pa.getTypeAndValue().getOther()
           .getName().compareToIgnoreCase("ACTIVITY_OTHER") == 0){
   System.out.println("Other Activity->" +
          pa.getTypeAndValue().getOther().getValue() + "\n");
 } else {
    // Currently, the only other value beside ACTIVITY_OTHER is
    // "ServiceAndDeviceNote" which is the service note +
    // device note.
    System.out.println("Combined Note->" +
        pa.getTypeAndValue().getOther().getValue() + "\n");
    }
  }
}
```

8.5.3 Interface: PresenceNotification, Operation: statusChanged

This asynchronous operation is called by the Web Service when an attribute for which notifications were requested changes.

8.5.3.1 Code Example

```
public void
statusChanged(String context, String correlator, String uri,
List<PresenceAttribute> presenceAttributes) {
System.out.println("statusChanged Called:-");
System.out.println("Context = " + context);
System.out.println("Correlator = " + correlator);
System.out.println("Presentity = " + uri);
}
```

8.5.4 Interface: PresenceNotification, Operation: statusEnd

This method is called when the duration for the notifications (identified by the correlator) is over. In case of an error or explicit call to endPresenceNotification, this method is not called.

8.5.4.1 Code Example

```
public void statusEnd(String context, String correlator)
System.out.println("statusEnd Called:-");
System.out.println("Context = " + context);
System.out.println("Correlator = " + correlator);
}
```

8.5.5 Interface: PresenceNotification, Operation: notifySubscription

This asynchronous method notifies the watcher that the presentity handled the pending subscription.

8.5.5.1 Code Example

```
public void notifySubscription(String context, String uri,
List<PresencePermission> presencePermissions) {
System.out.println("notifySubscription Called:-");
System.out.println("Context = " + context);
System.out.println("Uri = " + uri);
if (presencePermissions.size() > 0){
for (PresencePermission p:presencePermissions){
System.out.println("Permission " +
p.getPresenceAttribute().value()
+ "->" + p.isDecision());
}
}
}
```

8.5.6 Interface: PresenceNotification, Operation: subscriptionEnded

This asynchronous operation is called by the Web Service to notify the watcher that the subscription has terminated.

8.5.6.1 Code Example

```
public void subscriptionEnded(String context, String uri, String reason) {
System.out.println("subscriptionEnded Called:-");
System.out.println("Context = " + context);
System.out.println("Uri = " + uri);
System.out.println("Reason = " + reason);
}
```

8.5.7 Interface PresenceSupplier, Operation: publish and Oracle Specific "Unpublish"

This is the first operation the application must call before using another operation in this interface. It serves three purposes:

- It allows the Web services to associate the current HTTP session with a user.
- It publishes the user's presence status.
- It subscribes to watcher-info so that the Web services can keep track of any watcher requests.

There are three attributes that are of interest when performing a PUBLISH. These attributes can be set in a PresenceAttribute structure and passed into the PUBLISH method.

- Presense status with a customized note: this is the customized note configured in the My Presence text box in Oracle Communicator. The <note> element is contained in the <person> element of the Presence Information Data Format (PIDF) XML file.
- Device note: implicitly inserted by Oracle Communicator, or inserted from a Web service. The <note> element is contained in the <device> element of the Presence Information Data Format (PIDF) XML file.
- Service note: configured in the Presence tab in the Oracle Communicator preferences. The <note> element is contained in the <tuple> element of the Presence Information Data Format (PIDF) XML file.

8.5.7.1 Code Example

```
// A simple way to publish the Presence Status
PresenceAttribute pa = new PresenceAttribute();
OtherValue other = new OtherValue();
//Set the name to "DeviceNote" to indicate the value must be used as device note.
other.setName("DeviceNote");
other.setValue("Device Name");
//More other values can be defined for ServiceNote etc
CommunicationValue comm = new CommunicationValue();
AttributeTypeAndValue typeValue = new AttributeTypeAndValue();
typeValue.setUnionElement(PresenceAttributeType.ACTIVITY);
typeValue.setActivity(activity);
typeValue.setPlace(PlaceValue.PLACE_NONE);
typeValue.setPrivacy(PrivacyValue.PRIVACY_NONE);
typeValue.setSphere(SphereValue.SPHERE_NONE);
typeValue.setCommunication(comm);
typeValue.setOther(other);
pa.setTypeAndValue(typeValue);
String note = "My Note";
pa.setNote(note);
XMLGregorianCalendar dateTime = null;
dateTime = DatatypeFactory.newInstance().newXMLGregorianCalendar(new
GregorianCalendar());
pa.setLastChange(dateTime);
List<PresenceAttribute> pat = new ArrayList<PresenceAttribute>();
pat.add(pa);
supplier.publish(pat);

//To UNPUBLISH,set the OtherValue to (Expires, 0)
OtherValue other = new OtherValue();
other.setName("Expires");
other.setValue(0);
```

8.5.8 Interface: PresenceSupplier, Operation: getOpenSubscriptions

This operation retrieves a list of new requests to be on your watcher list.

8.5.8.1 Code Example

```
List<SubscriptionRequest> srList = getOpenSubscriptions();
for (SubscriptionRequest sr :srList) {
System.out.println(sr.getWatcher() .toString());
}
```

8.5.9 Interface: PresenceSupplier, Operation: updateSubscriptionAuthorization

This operation allows you to place a watcher on either the block or allow list.

8.5.9.1 Code Example

```
PresencePermission p = new PresencePermission();
p.setDecision(true);
List<PresencePermission> pp = new ArrayList<PresencePermission>();
p.setPresenceAttribute(PresenceAttributeType.ACTIVITY);
pp.add(p);
updateSubscriptionAuthorization("sip:allow@test.example.com",pp);
```

8.5.10 Interface: PresenceSupplier, Operation: getMyWatchers

This operation retrieves the list of watchers in your allow list.

8.5.10.1 Code Example

```
List<String> watchers = getMyWatchers();

for(String watcher: watchers){
System.out.println(watcher);
}
```

8.5.11 Interface: PresenceSupplier, Operation: getSubscribedAttributes

This operation returns only a single item of PresenceTypeAttribute.Activity. An exception is thrown if there is no existing subscription.

8.5.11.1 Code Example

```
List<PresenceAttributeType> pat =
getSubscribedAttributes("sip:watcher@test.example.com");
```

8.5.12 Interface: PresenceSupplier, Operation: blockSubscription

This operation places a watcher into the block list.

8.5.12.1 Code Example

```
blockSubscription("sip:block.this.watcher@test.example.com");
```

8.6 OWLCS Parlay X Presence Custom Error Codes

Table 8–4 and Table 8–5 describe the error codes and their associated error message.

Table 8–4 OWLCS Parlay X Presence Custom Error Codes: PolicyException

Error Code	Error Message
POL0001	General Policy Exception. It can be of following types: SDP20201 Watcher is on the block, polite-block or pending list. SDP20202 Subscription is pending.
POL0002	Privacy verification failed for address <address>, request is refused.
POL0003	Too many addresses specified in message part.

Table 8–5 OWLCS Parlay X Presence Custom Error Codes: ServiceException

Error Code	Error Message
SVC0001	General Service Exception. It can be of the following types: SDP20101 Invalid result from XDMS server. SDP20102 Invalid HTTP session data. SDP20103 Invalid uri. SDP20104 Peer unavaliable. SDP20105 Unknown host. SDP20106 Service not avaliable. SDP20107 Internal error. SDP20108 User unauthenticated.
SVC0002	Invalid input value for message part.
SVC0003	Invalid input value for message part, valid values are <values>.
SVC0004	No valid addresses provided in message part.
SVC0005	Correlator <correlator> specified in message part is a duplicate
SVC0220	No subscription request from watcher <watcher> for attribute <attribute>.
SVC0221	<watcher> is not a watcher.

8.7 Buddy List Manager API

The Contact Management API (CMAPI) is an API for manipulating resource-lists (also known as *Buddy Lists*) and presence-rules documents. Through this high-level API it is possible to act on behalf of a user to add or remove buddies to the buddy list as well as allowing or blocking other users (watchers) from seeing the user's presence information. The CMAPI is capable of querying and manipulating those resources stored on the XDMS (XML Document Management Server). The CMAPI consists of a web service: XML Document Management Client (XDMC) Service and a Java client stub that is part of the `oracle.sdp.client` shared library.

8.7.1 Consuming the API

The CMAPI is part of the oracle.sdp.client shared library. Once this library is available, developers can import the package and use the API:

```
import oracle.sdp.presence.integration.Buddy;
import oracle.sdp.presence.integration.BuddyListManager;
import oracle.sdp.presence.integration.BuddyListManagerFactory;
import oracle.sdp.presence.integrationimpl.BuddyListManagerImpl;
```

8.7.1.1 Obtaining the BuddyListManagerFactory

The BuddyListManagerFactory itself follows the singleton pattern, and there is only one instance of a factory per XDMS/XDMC combination. That is, when creating a BuddyListManagerFactory, you must supply the XCAP root URL to the XDMS from where documents are downloaded, as well as supplying the URL to the XDM Client Service that is running on the client side; the XDMC Service URL is passed in through the BindingProvider.ENDPOINT_ADDRESS_PROPERTY property. For each such combination of XCAP root URL and XDM Client Service endpoint, there can only exist exactly one BuddyListManagerFactory instance. Therefore it is possible to create different factories pointing to the different XDMS/XDMC Service combinations.

Example 8–1 Obtaining an instance of the BuddyListManagerFactory

```
// Create the URI pointing to the XDMS.
URI xcapRoot = new URI("http://localhost:8001/services");
// Location of where the XDM Client webservice is.
String wsUrl = "http://localhost:8001/XdmClientService/services/XdmClient";
String sWsSecurityPolicy = new String[]{"oracle/wss11_saml_token_with_message_
protection_client_policy"};
Map<String, Object> params = new HashMap<String,Object>();
params.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, wsUrl);
params.put(BindingProvider.USERNAME_PROPERTY, "alice");
params.put(ParlayXConstants.POLICIES, sWsSecurityPolicy);
// Obtain the instance to the factory
BuddyListManagerFactory factory = BuddyListManagerFactory.getInstance(xcapRoot,
params);
```

Example 8–1 shows how to obtain a reference to a factory pointing to the XCAP root of `localhost:8001/` services. Every operation performed on this factory is in the context of this particular XCAP root. Hence, when creating a BuddyListManager for a particular user, that BuddyListManager's XCAP root is the one of the factory through which it was created.

8.7.1.2 Creating a BuddyListManager

It is important to realize that a BuddyListManager (BLM) is acting on behalf of a particular user. Therefore, if a BLM is created for user *Alice*, all operations performed on that particular BLM are on behalf of Alice and manipulate her documents. Example 8–2 shows how to create a BLM for Alice through the factory created in the previous section.

Example 8–2 Obtaining a BuddyListManager for the user Alice

```
URI user = new URI("sip:alice@example.com");
Map<String, Object> params = new HashMap<String,Object>();
params.put(XDMClientFactory.PROP_ASSERTED_IDENTITY, assertedId);
BuddyListManager manager = factory.createBuddyListManager(user, params);
```

Example 8–2 shows how to create a BLM for the user Alice with SIP address of *sip:alice@example.com*. If manipulation of the buddy list and presence rules document of another user is required, then a separate BLM must be created with the appropriate SIP address.

8.7.1.3 Adding a Buddy to a Buddy List and Retrieving the List

Adding a buddy to a buddy list is done by first creating a buddy, setting the information needed on that buddy and then using the BLM to add it to the buddy list. Example 8–3 shows how to use the BLM representing *Alice* to add *Bob* as a new buddy of Alice and then getting the updated list back.

Example 8–3 Adding a New Buddy to the Buddy List of Alice

```
URI uri = new URI("sip:bob@example.com");
Buddy bob = manager.createBuddy(uri);
// Optionally, setting additional information.
manager.setDisplayname("Bobby");
VCard vcard = bob.getVCard();
vcard.setCity("San Francisco");
vcard.setCountry("USA");
// very important to set the VCard back on the buddy again
bob.setVCard(vcard);
// Update the buddy info using the BLM
manager.updateBuddy(bob);
// Getting the updated buddy list
List<Buddy> buddies = manager.getBuddies();
```

Example 8–3 shows how to create a new Buddy, *Bob*, and how that buddy is added to Alice's buddy list by using the BLM representing Alice. To add more information about the user Bob, such as the address and other information, access Bob's Vcard information and then set the appropriate properties.

Note: Since the method `getVCard()` is actually returning a clone of the VCard, the method `setVCard()` must be called on the buddy again in order for the information to be updated.

8.7.1.4 Removing a Buddy from a Buddy List

Removing a buddy is very similar to adding a buddy. Use the method `removeBuddy` and pass in the buddy that is to be removed. If there are many buddies to remove, use the `removeBuddies` method and pass in the list of buddies to remove. Example 8–4 shows how Bob is removed from Alice's buddy list.

Example 8–4 Removing a Buddy

```
URI uri = new URI("sip:bob@example.com");
Buddy bob = manager.createBuddy(uri);
manager.removeBuddy(bob);
```

8.7.1.5 Manipulating your presence rules document

To allow a watcher to view the presence status, use the method `allowWatcher`*`(String watcher)`* to add the watcher to the allow list. Use `blockWatcher`*`(String watcher)`* to block someone from viewing your presence status.

Example 8–5 Allowing or blocking watchers

```
manager.allowWatcher("sip:bob@example.com");
manager.blockWatcher("sip:carol@example.com");
```

8.7.2 Exceptions

`BuddyListException` is the base exception, and if the program is not set to register the specific exception, then it can simply catch it.

`XDMException` is the base exception for all exceptions concerning communication with the remote XDMS. `XDMException` signals that an error occurred when communicating with the XDMS (for example: a connection problem, wrong path to the XCAP root, or something else).

`DocumentConflictException` is a subclass to the `XDMException`; it signals that a mid-air conflict was detected that could not be resolved. This can occur when multiple clients access the same document on the XDMS. `BuddyListManager` attempts to resolve such a clash, but if it cannot, it throws an exception.

9 Parlay X Web Services Multimedia Messaging API

This chapter describes support for the Parlay X 2.1 Multimedia Messaging Web Services interfaces for developing applications. The Web service functions as a Messaging Agent which can send, receive, and listen to notifies on behalf of the users of the Web service. This chapter contains the following sections:

9.1 Introduction

The following sections describe the semantics of each of the supported operations along with implementation-specific details for the Parlay X Gateway.

The product support the interfaces defined in the Parlay X 2.1 Multimedia Messaging Web Services specification.

9.2 Installing the Web Services

The Web services are packaged as a standard .ear file and can be deployed the same as any other Web services through Admin Console. The .ear file contains three .war files that implement the three interfaces. The Web services use the Oracle SDP Platform, Client and Presence Commons shared libraries.

9.3 Configuring Web Services

There are four mbean attributes that are configurable for the Messaging Web service:

1. SIPOutboundProxy - SipURI of the outbound proxy for SIP message. Empty string means no outbound proxy. Currently, only support IP address. For example, sip:127.0.0.1:5060; lr;transport=tcp.
2. SessionTimeout - Set the time in seconds after which HTTP sessions time out. Data for all timed out sessions is discarded.
3. MessageLifetime - Set the time in seconds after which messages expire from the message store. Setting this to 0 causes messages to be kept in the store indefinitely

(never expire). Messages stay in the message store for at most MessageLifetime + MessageScanPeriod seconds. Setting this attribute has immediate effect (for instance, reducing the value could cause some messages to be immediately expired if they are older than the lifetime).

4. MessageScanPeriod - Set the period in seconds for scanning for and deleting expired messages. Setting this to 0 disables scanning. Setting this attribute has immediate effect.

9.4 Messaging Web Services Interface Descriptions

The messaging Web services consist of four interfaces:

- SendMessage: Use these methods to send messages (Table 9–1).
- ReceiveMessage: Use these methods to receive message content (Table 9–2).
- MessageNotificationManager: Use these methods to manage which users are notified when messages are received through the Web service (Table 9–3).
- MessageNotification: The client callback defined in this interface is used to send notifications (Table 9–4).

Table 9–1 SendMessage Interface

Operation	Description
sendMessage	Sends a SIP MESSAGE to designated user(s). Returns an outgoing message ID.
getMessageDeliveryStatus	Returns a set of delivery statuses for each recipient of an outgoing message sent through sendMessage.

Table 9–2 ReceiveMessage Interface

Operation	Description
getMessage	Receives an incoming message.
getMessageURIs	Not implemented.
getReceivedMessages	Returns a set of incoming messages for a given user.

Table 9–3 MessageNotificationManager Interface

Operation	Description
startMessageNotification	Starts message notification at a given endpoint for a user. Notifies endpoint when messages are received for user.
stopMessageNotification	Stops message notification at an endpoint for a user.

Table 9–4 MessageNotification Interface

Operation	Description
notifyMessageDeliveryReceipt	Client callback invoked to notify the user of a message's final delivery status.
notifyMessageReception	Client callback invoked to notify the client that the user received a message.

9.5 Using the Messaging Web Services Interfaces

This section describes how to use each of the operations in the interfaces, and includes code examples. The following requirements apply:

- An argument of type "StringSipURI" means that the argument is a String but must be a valid URI with "sip" or "sips" scheme, otherwise a ServiceException gets thrown. Refer to the Oracle Fusion Middleware WebLogic Communication Services API Reference for additional documentation on the content indirection API.

9.5.1 Interface SendMessage, Operation: sendMessage

This operation sends a SIP MESSAGE to designated user(s). Returns an outgoing message ID.

Table 9–5 Interface: SendMessage, Operation: sendMessage

Argument	Type	Required	Description
addresses	List<StringSipURI>	yes	Destination address(es) for this message.
senderAddress	StringSipURI	yes	Message sender address.
subject	String	no	Message subject. If there is no plain text attachment with the request, the subject is treated as the message content.
priority	MessagePriority	no	This value is ignored.
charging	ChargingInformation	no	This value is ignored.
receiptRequest	SimpleReference	no	Defines the application endpoint, interfaceName, and correlator that is used to notify the application of the final delivery status of the message.
Return Value	**Type**		**Description**
messageIdentifier	String		This identifier is used in a getMessageDeliveryStatus operation invocation to get the delivery status of sent messages.

9.5.1.1 Code Example

```
Map<String, Object> params = new HashMap<String, Object>();
params.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    "http://webservicehost:7001/sendMessageEndpoint");
SendMessageClient sendMsgClient = new SendMessageClient(params);
List<String> recipients = new ArrayList<String>();
recipients.add("sip:receiver@example.com");
String correlator = UUID.randomUUID().toString();
SimpleReference ref = new SimpleReference();
ref.setCorrelator(correlator);
ref.setEndpoint("http://clienthost:8080/notificationEndpoint");
ref.setInterface("MessageNotification");
String msgID = sendMsgClient.sendMessage(recipients,
    "sip:sender@example.com", "message content",
    MessagePriority.DEFAULT, null, ref);
```

9.5.2 Interface SendMessage, Operation: getMessageDeliveryStatus

This operation returns a set of delivery statuses for each recipient of an outgoing message sent via sendMessage. Call this operation with the ID returned by sendMessage.

Table 9–6 Interface SendMessage, Operation: getMessageDeliveryStatus

Argument	Type	Required	Description
messageIdentifier	String	yes	Identifier related to the delivery status request.
Return Value	**Type**		**Description**
status	List<DeliveryInformation>		A list of status of the messages that were previously sent. Each item represents a sent message, its destination address, and its delivery status.

9.5.2.1 Code Example

```
String msgID = sendMsgClient.sendMessage(...);
List<DeliveryInformation> infoList =
    sendMsgClient.getMessageDeliveryStatus(msgID);
for (DeliveryInformation info : infoList) {
    System.out.println("recipient: " + info.getAddress());
    System.out.println("status: " + info.getDeliveryStatus());
}
```

9.5.3 Interface MessageNotificationManager, Operation: startMessageNotification

This operation starts message notification at a given endpoint for a user. This means that when messages are received for this user, the client callback notifyMessageReception is invoked at the given MessageNotification endpoint. This also means that the web service stores received messages for this user, and the received messages can be obtained through the ReceiveMessage interface.

Table 9–7 Interface MessageNotificationManager, Operation: startMessageNotification

Argument	Type	Required	Description
reference	SimpleReference	yes	Defines the application endpoint, interfaceName, and correlator that is used to notify the application when a message is received.
messageServiceActivationNumber	StringSipURI	yes	The application is notified when messages are received for this SIP address.
criteria	String	no	This value is ignored.

9.5.3.1 Code Example

```
Map<String, Object> params = new HashMap<String, Object>();
params.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    "http://webservicehost:7001/msgNotiMgrEndpoint");
MessageNotificationManagerClient msgNotiMgrClient =
    new MessageNotificationManagerClient(params);
SimpleReference ref = new SimpleReference();
String correlator = UUID.randomUUID().toString()
ref.setCorrelator(correlator);
ref.setEndpoint("http://clienthost:8080/notificationEndpoint");
ref.setInterface("MessageNotification");
msgNotiMgrClient.startMessageNotification(ref,
    "sip:receiver@example.com","dummy_criteria_ignored");
```

9.5.4 Interface MessageNotificationManager, Operation: stopMessageNotification

This operation stops message notification at an endpoint for a user. If a user no longer has notification endpoints, all received messages for that user are no longer stored.

Table 9–8 Interface MessageNotificationManager, Operation: stopMessageNotification

Argument	Type	Required	Description
correlator	String	yes	The correlator associated with an invocation of the startMessageNotification operation.

9.5.4.1 Code Example

```
msgNotiMgrClient.stopMessageNotification(correlator);
```

9.5.5 Interface ReceiveMessage, Operation: getReceivedMessages

This operation returns a set of incoming messages for a given user. Messages may only be received after notification has been enabled by invoking the startMessageNotifcation operation in the MessageNotificationManager interface.

Table 9–9 Interface ReceiveMessage, Operation: getReceivedMessages

Argument	Type	Required	Description
registrationIdentifier	StringSipURI	yes	The recipient SIP address for incoming messages.
priority	MessagePriority	no	This value is ignored.
Return Value	**Type**		**Description**
references	List<MessageReference>		A list of messages received for this user. Each item may either have a message identifier or message content, but is *not* guaranteed to have both an identifier and content.

9.5.5.1 Code Example

```
Map<String, Object> params = new HashMap<String, Object>();
params.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    "http://webservicehost:7001/receiveMessageEndpoint");
ReceiveMessageClient recvMsgClient = new ReceiveMessageClient(params);
List<MessageReference> msgs =
    recvMsgClient.getReceivedMessages("sip:receiver@example.com",
    MessagePriority.DEFAULT);
for (MessageReference ref : msgs) {
    System.out.println("to: "+ref.getMessageServiceActivationNumber();
    System.out.println("from: "+ref.getSenderAddress());
    System.out.println("subject: "+ref.getSubject());
    String id = ref.getMessageIdentifier();
    if (id == null || id.isEmpty()) {
        System.out.println("message: "+ref.getMessage());
    } else {
        System.out.println("ID: "+id);
    }
}
```

9.5.6 Interface: ReceiveMessage, Operation: getMessage

This operation receives an incoming message as an attachment. Messages may only be received after notification has been enabled by invoking the startMessageNotifcation operation in the MessageNotificationManager interface.

Table 9–10 Oracle WebLogic Communication ServicesInterface: ReceiveMessage, Operation: getMessage

Argument	Type	Required	Description
messageIdentifier	String	yes	A string identifying the incoming message. This string is obtained either from the notifyMessageReception callback, or the getReceivedMessages operation invocation.
Return Value	**Type**		**Description**
n/a	n/a		After invoking the getMessage operation, the message content is stored in an attachment of type DataHandler.

9.5.6.1 Code Example

```
List<MessageReference> msgs =
    recvMsgClient.getReceivedMessages("sip:receiver@example.com",
    MessagePriority.DEFAULT);
for (MessageReference ref : msgs) {
    String id = ref.getMessageIdentifier();
    String msgContent;
    if (id == null || id.isEmpty()) {
        msgContent = ref.getMessage();
    } else {
        System.out.println("ID: " + id);
        recvMsgClient.getMessage(id);
        DataHandler dh = recvMsgClient.getAttachment();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        BufferedOutputStream out = new BufferedOutputStream(baos);
        dh.writeTo(out);
        out.flush();
        msgContent = baos.toString();
    }
    System.out.println("message: " + msgContent);
}
```

Part IV

Call Control

This part describes using call control functionality.

Part V contains the following chapter:

10 Third Party Call Service

This chapter describes how to perform third party call handling using a multimedia messaging API, and provides samples applications. It contains the following section:

10.1 Overview of Parlay X 2.1 Third Party Call Communication Services

The Third Party Call Parlay X 2.1 communication services implement the Parlay X 2.1 Third Party Call interface, (Standards reference: ETSI ES 202 391-2 V1.2.1 (2006-12), Open Service Access (OSA); Parlay X Web Services; Part 2: Third Party Call (Parlay X 2)).

Using a Third Party Call Parlay X 2.1 communication service, an application can:

- Set up a call between two parties. For example, an application could set up a call between an investor and a broker if a particular stock reaches a predetermined price. Or a computer user could set up a call between himself and someone in the address book with a mouse click.
- Query for the status of a previously set up call
- Cancel a call it is creating as it is about to be set up
- Terminate an ongoing call it created

10.1.1 How It Works

In the Parlay X 2.1 Third Party Call communication services model, a call has two distinct stages:

10.1.1.1 Call Setup

There are two parties involved in Third Party Call calls: the A-party (the caller) and the B-party (the callee). When a call is set up using a Third Party Call communication service, OWLCS attempts to set up a call leg to the A-party. When the caller goes off-hook (“answers”), OWLCS attempts to set up a call leg to the B-party. When the callee goes off-hook, the two call legs are connected using the underlying network. This ends the call setup-phase.

The application can cancel the call during this phase.

10.1.1.2 Call Duration

While the call is underway, the audio channel that connects the caller and the callee is completely managed by the underlying network. During this phase of the call, the application can only query as to the status of the call. A call can be terminated in two ways, either using the application-facing interface, or having the caller or callee hang up.

Requests using a Parlay X 2.1 Third Party Call communication service flow only in one direction, from the application to the network. Therefore this communication service supports only application-initiated (or mobile-terminated) functionality.

Note: Third Party Call communication services manage only the signalling, or controlling, aspect of a call. The media, or audio, channel is managed by the underlying network. Only parties residing on the same network can be controlled, unless:

- ·The network plug-in connects to a media gateway controller

  ·One of the participants is connected to a signalling gateway so that, from a signalling point of view, all parties reside on the same network

10.1.2 Supported Networks

Off the shelf, Parlay X 2.1 Third Party Call communication services can be configured to support the SIP network protocol.

Note: OWLCS acts as a Back-to-Back User Agent. During the call duration phase, the actual call is peer-to-peer.

10.2 Configuring Parlay X 2.1 Third Party Call

This section contains a description of the configuration attributes and operations available for the Parlay X 2.1 Third Party Call.

10.2.1 Configuration Workflow for Parlay X 2.1 Third Party Call/SIP

Follow these configuration steps:

1. Using the Management Console or an MBean browser, select the MBean detailed in Properties for Parlay X 2.1 Third Party Call/SIP.
2. Configure behavior of the network protocol plug-in instance:
   - Attribute: ThirdPartyCallControllerURI
   - Attribute: ISCRouteURI
   - Attribute: MaximumCallLength
   - Attribute: StatusRetentionTime
   - Attribute: PAssertedIdentityURI

Note: There are not any management actions.

Table 10–1 lists the properties for Parlay X 2.1 Third Party Call

Table 10–1 Properties for Parlay X 2.1 Third Party Call

Property	Description

MBean	Domain=oracle.sdp
	Name=thirdpartycall
	InstanceName=ThirdPartyCall
	Type=oracle.sdp.thirdpartycall.management.TPCMBean

Supported Address Scheme	sip
Service type	ThirdPartyCall
Exposes to the service communication layer a JAVA representation of:	Parlay X2.1 Part 2: Third Party Call
Interfaces with the network nodes using:	SIP: Session Initiation Protocol, RFC 3261
Deployment artifacts:	thirdpartycallwswar-11.1.1.war, thirdpartycallmanagerwar-11.1.1.war, thirdpartycallutil-11.1.1.jar, thirdpartycallmanager-11.1.1.jar, parlayx-11.1.1.jar, packaged in thirdpartycallear-11.1.1.ear

10.2.2 Attributes and Operations for Parlay X 2.1 Third Party Call

Table 10–2 contains a list of attributes for configuration and maintenance

Table 10–2 Configuration and Maintenance Attributes

Attribute	Scope	Unit	Format	Description
ThirdPartyCall ControllerURI	Cluster	NA	String in URI format	Specifies the Controller SIP URI that is used to establish the third party call. If this value is set, a call appears to the callee to come from this URI. By default, the value is "None", where no controller URI is used to establish the call. In this case, the call appears to the callee to come from the caller
ISCRouteURI	Cluster	NA	String in URI format	Specifies the URI of the IMS service control route.
MaximumCall Duration	Cluster	minutes	int	Specifies for how long time a call is allowed to be ongoing. If this time expires, the call is terminated.
StatusRetentionTime	Cluster	minutes	int	Specifies for how long time to retain status information about the call after it has been terminated.
PAssertedIdentity URI	Cluster	NA	String in URI format	Specifies the SIP URI used in the P-Asserted-Identity header added by the Third Party Call service. If left blank no P-Asserted-Identity header is added.

10.3 Statement of Compliance

This section describes the standards compliance for the communication services for Parlay X 2.1 Third Party call.

The Parlay X 2.1 interface complies to ETSI ES 202 391-2 V1.2.1 Open Service Access (OSA); Parlay X Web Services; Part 2: Third Party Call (Parlay X 2). For more information, see the relevant specification at http://parlay.org/en/specifications/pxws.asp

Table 10–3 Statement of Compliance, Parlay X 2.1 Third Party Call

Method	Compliant Yes/No
Interface: ThirdPartyCall	
MakeCall	Y
GetCallInformation	Y
EndCall	Y
CancelCall	Y

The SIP plug-in for Parlay X 2.1 Third Party Call is an integration plug-in that utilizes the Oracle WebLogic SIP Server to connect to a SIP/IMS network. The plug-in connects to a SIP servlet executing in WebLogic SIP Server. The SIP Servlet uses the SIP API provided by the WebLogic SIP server, which in its turn converts the API calls to SIP messages.

The SIP servlet acts as a Back-to-Back User Agent for all calls.

The SIP servlet uses the WebLogic SIP server, which conforms to RFC 3261 (http://www.ietf.org/rfc/rfc3261.txt).

The implementation of the SIP based third party call is in compliance with RFC 3725 - Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP) Flow I (http://www.ietf.org/rfc/rfc3725.txt).

Table 10–4 Statement of Compliance, SIP for Parlay X 2.1 Third Party Call

Message/Response	Compliant Yes/No	Comment
REGISTER	-	Not used in the context
INVITE	Y	
ACK	Y	
CANCEL	Y	
BYE	Y	
OPTIONS	-	Not used in the context.
100 Trying	Y	
180 Ringing	Y	
181 Call Is Being Forwarded	Y	
182 Queued	Y	
183 Session Progress	Y	
200 OK	Y	

Table 10–4 (Cont.) Statement of Compliance, SIP for Parlay X 2.1 Third Party Call

Message/Response	Compliant Yes/No	Comment
300 Multiple Choices	Y	UA treated as unreachable.
301 Moved Permanently	Y	UA treated as unreachable.
302 Moved Temporarily	Y	UA treated as unreachable.
305 Use Proxy	Y	UA treated as unreachable.
380 Alternate Service	Y	UA treated as unreachable.
400 Bad Request	Y	UA treated as unreachable.
401 Unauthorized	Y	UA treated as unreachable.
402 Payment Required	Y	UA treated as unreachable.
403 Forbidden	Y	UA treated as unreachable.
404 Not Found	Y	UA treated as unreachable.
405 Method Not Allowed	Y	UA treated as unreachable.
406 Not Acceptable	Y	UA treated as unreachable.
407 Proxy Authentication Required	Y	UA treated as unreachable.
408 Request Timeout	Y	Treated as no answer from UA.
410 Gone	Y	UA treated as unreachable.
413 Request Entity Too Large	Y	UA treated as unreachable.
414 Request URI Too Long	Y	UA treated as unreachable.
415 Unsupported Media Type	Y	UA treated as unreachable.
416 Unsupported URI Scheme	Y	UA treated as unreachable.
420 Bad Extension	Y	UA treated as unreachable.
421 Extension Required	Y	UA treated as unreachable.
423 Interval Too Brief	Y	UA treated as unreachable.
480 Temporarily Unavailable	Y	UA treated as unreachable.
481 Call/Transaction Does Not Exist	Y	UA treated as unreachable.
482 Loop Detected	Y	UA treated as unreachable.
483 Too Many Hops	Y	UA treated as unreachable.
484 Address Incomplete	Y	UA treated as unreachable.
485 Ambiguous	Y	UA treated as unreachable.
486 Busy Here	Y	UA treated as busy.
487 Request Terminated	Y	UA treated as unreachable.
488 Not Acceptable Here	Y/N	UA treated as unreachable.
491 Request Pending	Y	UA treated as unreachable.
493 Undecipherable	Y	UA treated as unreachable.
500 Server Internal Error	Y	UA treated as unreachable.
501 Not Implemented	Y	UA treated as unreachable.
502 Bad Gateway	Y	UA treated as unreachable.

Table 10–4 (Cont.) Statement of Compliance, SIP for Parlay X 2.1 Third Party Call

Message/Response	Compliant Yes/No	Comment
503 Service Unavailable	Y	UA treated as unreachable.
Server Time-out	Y	UA treated as unreachable.
505 Version Not Supported	Y	UA treated as unreachable.
513 Message Too Long	Y	UA treated as unreachable.
600 Busy Everywhere	Y	UA treated as unreachable.
603 Decline	Y	UA treated as unreachable.
604 Does Not Exist Anywhere	Y	UA treated as unreachable.
606 Not Acceptable	Y	UA treated as unreachable.

Part V
Using Diameter

This part describes developing applications using Diameter. Diameter is a peer-to-peer protocol that involves delivering attribute-value pairs (AVPs). A Diameter message includes a header and one or more AVPs. The collection of AVPs in each message is determined by the type of Diameter application, and the Diameter protocol also allows for extension by adding new commands and AVPs. Diameter enables multiple peers to negotiate their capabilities with one another, and defines rules for session handling and accounting functions.

OWLCS includes an implementation of the base Diameter protocol that supports the core functionality and accounting features described in RFC 3588 (`http://www.ietf.org/rfc/rfc3588.txt`). OWLCS uses the base Diameter functionality to implement multiple Diameter applications, including the Sh, Rf, and Ro applications described later in this document.

You can also use the base Diameter protocol to implement additional client and server-side Diameter applications. The base Diameter API provides a simple, Servlet-like programming model that enables you to combine Diameter functionality with SIP or HTTP functionality in a converged application.

Part VI contains the following chapters:

11 Using the Diameter Base Protocol API

The following chapter provides an overview of using the OWLCS Diameter Base protocol implementation to create your own Diameter applications, in the following sections:

- Section 11.1, "Diameter Protocol Packages"
- Section 11.2, "Overview of the Diameter API"
- Section 11.3, "Working with Diameter Nodes"
- Section 11.4, "Implementing a Diameter Application"
- Section 11.5, "Working with Diameter Sessions"
- Section 11.6, "Working with Diameter Messages"
- Section 11.7, "Working with AVPs"
- Section 11.8, "Creating Converged Diameter and SIP Applications"

11.1 Diameter Protocol Packages

The sections that follow provide an overview of the base Diameter protocol packages, classes, and programming model used for developing client and server-side Diameter applications. See also the following sections for information about using the provided Diameter protocol applications in your SIP Servlets:

- Chapter 12, "Using the Profile Service API" describes how to access and manage subscriber profile data using the Diameter Sh application.
- Chapter 14, "Using the Diameter Rf Interface API for Offline Charging" describes how to issue offline charging requests using the Diameter Rf application.
- Chapter 15, "Using the Diameter Ro Interface API for Online Charging" describes how to perform online charging using the Diameter Ro application.

11.2 Overview of the Diameter API

All classes in the Diameter base protocol API reside in the root `com.bea.wcp.diameter` package. Table 11–1 describes the key classes, interfaces, and exceptions in this package.

Table 11–1 Key Elements of the Diameter Base Protocol API

Category	Element	Description
Diameter Node	Node	A class that represents a Diameter node implementation. A diameter node can represent a client- or server-based Diameter application, as well as a Diameter relay agent.
Diameter Applications	Application, ClientApplication	A class that represents a basic Diameter application. ClientApplication extends Application for client-specific features such as specifying destination hosts and realms. All Diameter applications must extend one of these classes to return an application identifier. The classes can also be used directly to create new Diameter sessions.
	ApplicationId	A class that represents the Diameter application ID. This ID is used by the Diameter protocol for routing messages to the appropriate application. The `ApplicationId` corresponds to one of the Auth-Application-Id, Acct-Application-Id, or Vendor-Specific-Application-Id AVPs contained in a Diameter message.
	Session	A class that represents a Diameter session. Applications that perform session-based handling must extend this class to provide application-specific behavior for managing requests and answering messages.
Message Processing	Message, Request, Answer	The Message class is a base class used to represent request and answer message types. Request and Answer extend the base class.
	Command	A class that represents a Diameter command code.
	RAR, RAA	These classes extend the `Request` and `Answer` classes to represent re-authorization messages.
	ResultCode	A class that represents a Diameter result code, and provides constant values for the base Diameter protocol result codes.
AVP Handling	Attribute	A class that provides Diameter attribute information.
	Avp, AvpList	Classes that represent one or more attribute-value pairs in a message. AvpList is also used to represent AVPs contained in a grouped AVP.
	Type	A class that defines the supported AVP datatypes.
Error Handling	DiameterException	The base exception class for Diameter exceptions.
	MessageException	An exception that is raised when an invalid Diameter message is discovered.
	AvpException	An exception that is raised when an invalid AVP is discovered.

Table 11–1 (Cont.) Key Elements of the Diameter Base Protocol API

Supporting Interfaces	Enumerated	An enum value that implements this interface can be used as the value of an AVP of type INTEGER32, INTEGER64, or ENUMERATED.
	SessionListener	An interface that applications can implement to subscribe to messages delivered to a Diameter session.
	MessageFactory	An interface that allows applications to override the default message decoder for received messages, and create new types of `Request` and `Answer` objects. The default decoding process begins by decoding the message header from the message bytes using an instance of `MessageFactory`. This is done so that an early error message can be generated if the message header is invalid. The actual message AVPs are decoded in a separate step by calling `decodeAvps`. AVP values are fully decoded and validated by calling `validate`, which in turn calls `validateAvp` for each partially-decoded AVP in the message.

In addition to these base Diameter classes, accounting-related classes are stored in the `com.bea.wcp.diameter.accounting` package, and credit-control-related classes are stored in `com.bea.wcp.diameter.cc`. See Chapter 14, "Using the Diameter Rf Interface API for Offline Charging", and Chapter 15, "Using the Diameter Ro Interface API for Online Charging" for more information about classes in these packages.

11.2.1 File Required for Compiling Application Using the Diameter API

The following jar files are part of the Diameter API that we expose. To compile against this API, access this file from the following locations:

The `wlssdiameter.jar` file is located at the following location: `MIDDLEWARE_HOME/server/lib/wlss/`.

11.3 Working with Diameter Nodes

A diameter node is represented by the `com.bea.wcp.diameter.Node` class. A Diameter node may host one or more Diameter applications, as configured in the `diameter.xml` file. In order to access a Diameter application, a deployed application (such as a SIP Servlet) must obtain the diameter Node instance and request the application. Example 11–1 shows the sample code used to access the Rf application.

Example 11–1 Accessing a Diameter Node and Application

```
ServletContext sc = getServletConfig().getServletContext();
Node node = sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication) node.getApplication(Charging.RF_APPLICATION_
ID);
```

Diameter Nodes are generally configured and started as part of a OWLCS instance. However, for development and testing purposes, you can also run a Diameter node as a standalone process. To do so:

1. Set the environment for your domain:

   ```
   cd ~/bea/user_projects/domains/diameter/bin
   . ./setDomainEnv.sh
   ```

2. Locate the `diameter.xml` configuration file for the Node you want to start:

```
cd ../config/custom
```

3. Start the Diameter node, specifying the `diameter.xml` configuration file to use:

```
java com.bea.wcp.diameter.Node diameter.xml
```

11.4 Implementing a Diameter Application

All Diameter applications must extend either the base `Application` class or, for client applications, the `ClientApplication` class. The model for creating a Diameter application is similar to that for implementing Servlets in the following ways:

- Diameter applications override the `init()` method for initialization tasks.
- Initialization parameters configured for the application in `diameter.xml` are made available to the application.
- A session factory is used to generate new application sessions.

Diameter applications must also implement the `getId()` method to return the proper application ID. This ID is used to deliver Diameter messages to the correct application.

Applications can optionally implement `rcvRequest()` or `rcvAnswer()` as needed. By default, `rcvRequest()` answers with UNABLE_TO_COMPLY, and `rcvRequest()` drops the Diameter message.

Example 11–2 shows a simple Diameter client application that does not use sessions.

Example 11–2 Simple Diameter Application

```
public class TestApplication extends ClientApplication {
  protected void init() {
    log("Test application initialized.");
  }
  public ApplicationId getId() {
    return ApplicationId.BASE_ACCOUNTING;
  }
  public void rcvRequest(Request req) throws IOException {
    log("Got request: " + req.getHopByHopId());
    req.createAnswer(ResultCode.SUCCESS).send();
  }
}
```

11.5 Working with Diameter Sessions

Applications that perform session-based handling must extend the base Session class to provide application-specific behavior for managing requests and answering messages. If you extend the base Session class, you must implement either `rcvRequest()` or `rcvAnswer()`, and may implement both methods.

The base Application class is used to generate new Session objects. After a session is created, all session-related messages are delivered directly to the session object. The OWLCS container automatically generates the session ID and encodes the ID in each message. Session attributes are supported much in the same fashion as attributes in `SipApplicationSession`.

Example 11–3 shows a simple Diameter session implementation.

Example 11–3 Simple Diameter Session

```
public class TestSession extends Session {
  public TestSession(TestApplication app) {
    super(app);
  }
  public void rcvRequest(Request req) throws IOException {
    getApplication().log("rcvReuest: " + req.getHopByHopId());
    req.createAnswer(ResultCode.SUCCESS).send();
  }
}
```

To use the sample session class, the `TestApplication` in Example 11–2 would need to add a factory method:

```
public class TestApplication extends Application {
  ...
  public TestSession createSession() {
    return new TestSession(this);
  }
}
```

`TestSession` could then be used to create new requests as follows:

```
TestSession session = testApp.createSession();
Request req = session.creatRequest();
req.sent();
```

The answer is delivered directly to the Session object.

11.6 Working with Diameter Messages

The base `Message` class is used for both Request and Answer message types. A Message always includes an application ID, and optionally includes a session ID. By default, messages are handled in the following manner:

1. The message bytes are parsed.
2. The application and session ID values are determined.
3. The message is delivered to a matching session or application using the following rules:
   a. If the Session-Id AVP is present, the associated Session is located and the session's `rcvMessage()` method is called.
   b. If there is no Session-Id AVP present, or if the session cannot be located, the Diameter application's `rcvMessage()` method is called
   c. If the application cannot be located, an UNABLE_TO_DELIVER response is generated.

The message type is determined from the Diameter command code. Certain special message types, such as RAR, RAA, ACR, ACA, CCR, and CCA, have getter and setter methods in the `Message` object for convenience.

11.6.1 Sending Request Messages

Either a `Session` or `Application` can originate and receive request messages. Requests are generated using the `createRequest()` method. You must supply a command code for the new request message. For routing purposes, the destination host or destination realm AVPs are also generally set by the originating session or application.

Received answers can be obtained using `Request.getAnswer()`. After receiving an answer, you can use `getSession()` to obtain the relevant session ID and `getResultCode()` to determine the result. You can also use `Answer.getRequest()` to obtain the original request message.

Requests can be sent asynchronously using the `send()` method, or synchronously using the blocking `sendAndWait()` method. Answers for requests that were sent asynchronously are delivered to the originating session or application. You can specify a request timeout value when sending the message, or can use the global `request-timeout` configuration element in `diameter.xml`. An UNABLE_TO_DELIVER result code is generated if the timeout value is reached before an answer is delivered. `getResultCode()` on the resulting Answer returns the result code.

11.6.2 Sending Answer Messages

New answer messages are generated from the `Request` object, using `createAnswer()`. All generated answers should specify a ResultCode and an optional Error-Message AVP value. The `ResultCode` class contains pre-defined result codes that can be used.

Answers are delivered using the `send()` method, which is always asynchronous (non-blocking).

11.6.3 Creating New Command Codes

A Diameter command code determines the message type. For instance, when sending a request message, you must supply a command code.

The `Command` class represents pre-defined commands codes for the Diameter base protocol, and can be used to create new command codes. Command codes share a common name space based on the code itself.

The `define()` method enables you to define codes, as in:

```
static final Command TCA = Command.define(1234, "Test-Request", true, true);
```

The `define()` method registers a new Command, or returns a previous command definition if one was already defined. Commands can be compared using the reference equality operator (==).

11.7 Working with AVPs

Attribute Value Pair (AVP) is a method of encapsulating information relevant to the Diameter message. AVPs are used by the Diameter base protocol, the Diameter application, or a higher-level application that employs Diameter.

The `Avp` class represents a Diameter attribute-value pair. You can create new AVPs with an attribute value in the following way:

```
Avp avp = new Avp(Attribute.ERROR_MESSAGE, "Bad request");
```

You can also specify the attribute name directly, as in:

```
Avp avp = new Avp("Error-Message", "Bad request");
```

The value that you specify must be valid for the specified attribute type.

To create a grouped AVP, use the `AvpList` class, as in:

```
AvpList avps = new AvpList();
avps.add(new Avp("Event-Timestamp", 1234));
avps.add(new Avp("Vendor-Id", 1111));
```

11.7.1 Creating New Attributes

You can create new attributes to extend your Diameter application. The Attribute class represents an AVP attribute, and includes the AVP code, name, flags, optional vendor ID, and type of attribute. The class also maintains a registry of defined attributes. All attributes share a common namespace based on the attribute code and vendor ID.

The `define()` method enables you to define new attributes, as in:

```
static final Attribute TEST = Attribute.define(1234, "Test-Attribute", 0,
Attribute.FLAG_MANDATORY, Type.INTEGER32);
```

Table 11–1 lists the available attribute types and describes how they are mapped to Java types.

The `define()` method registers a new attribute, or returns a previous definition if one was already defined. Attributes can be compared using the reference equality operator (==).

Table 11–2 Attribute Types

Diameter Type	Type Constant	Java Type
Integer32	Type.INTEGER32	Integer
Integer64	Type.INTEGER64	Long
Float32	Type.FLOAT32	Float
OctetString	Type.BYTES	ByteBuffer (read-only)
UTF8String	Type.STRING	String
Address	Type.ADDRESS	InetAddress
Grouped	Type.GROUPED	AvpList

11.8 Creating Converged Diameter and SIP Applications

The Diameter API enables you to create converged applications that utilize both SIP and Diameter functionality. A SIP Servlet can access an available Diameter application through the Diameter Node, as shown in Example 11–4.

Example 11–4 Accessing the Rf Application from a SIP Servlet

```
ServletContext sc = getServletConfig().getServletContext();
Node node = (Node) sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication) node.getApplication(Charging.RF_APPLICATION_
ID);
```

SIP uses Call-id (the SIP-Call-ID header) to identify a particular call session between two users. OWLCS automatically links a Diameter session to the currently-active call state by encoding the SIP Call-id into the Diameter session ID. When a Diameter message is received, the container automatically retrieves the associated call state and locates the Diameter session. A Diameter session is serializable, so you can store the session as an attribute in a the `SipApplicationSession` object, or vice versa.

Converged applications can use the Diameter `SessionListener` interface to receive notification when a Diameter message is received by the session. The `SessionListener` interface defines a single method, `rcvMessage()`. Example 11–5 shows an example of how to implement the method.

Example 11–5 Implementing SessionListener

```
Session session = app.createSession();
session.setListener(new SessionListener() {
  public void rcvMessage(Message msg) {
    if (msg.isRequest()) System.out.println("Got request!");
  }
});
```

Note: The `SessionListener` implementation must be serializable for distributed applications.

12

Using the Profile Service API

The following chapter describes how to use the Diameter Sh profile service and the Profile Service API, based on the OWLCS Diameter protocol implementation, in your own applications, and contains the following sections:

12.1 Overview of Profile Service API and Sh Interface Support

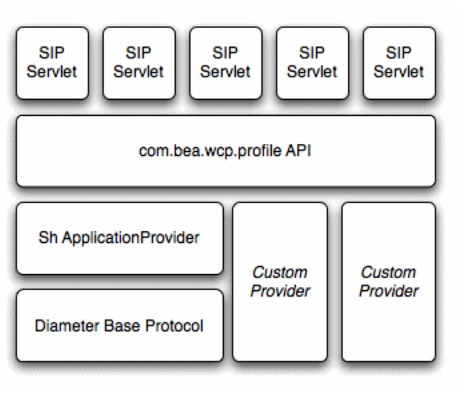
The IMS specification defines the Sh profile service as the method of communication between the Application Server (AS) function and the Home Subscriber Server (HSS), or between multiple IMS Application Servers. The AS uses the Sh profile service in two basic ways:

- To query or update a user's data stored on the HSS
- To subscribe to and receive notifications when a user's data changes on the HSS

The user data available to an AS may be defined by a service running on the AS (*repository data*), or it may be a subset of the user's IMS profile data hosted on the HSS. The Sh interface specification, 3GPP TS 29.328, defines the IMS profile data that can be queried and updated through Sh. All user data accessible through the Sh profile service is presented as an XML document with the schema defined in 3GPP TS 29.328.

The IMS Sh profile service is implemented as a provider to the base Diameter protocol support in OWLCS. The provider transparently generates and responds to the Diameter command codes defined in the Sh application specification. A higher-level Profile Service API enables SIP Servlets to manage user profile data as an XML document using XML Document Object Model (DOM). Subscriptions and notifications for changed profile data are managed by implementing a profile listener interface in a SIP Servlet.

Figure 12–1 Profile Service API and Sh Provider Implementation



OWLCS includes a provider for the Diameter Sh profile service. Providers to support additional interfaces defined in the IMS specification may be provided in future releases. Applications using the profile service API are able to use additional providers as they are made available.

12.2 Enabling the Sh Interface Provider

See "Configuring Diameter Sh Client Nodes and Relay Agents" in *Configuring Network Resources* for full instructions on setting up Diameter support.

12.3 Overview of the Profile Service API

OWLCS provides a simple profile service API that SIP Servlets can use to query or modify subscriber profile data, or to manage subscriptions for receiving notifications about changed profile data. Using the API, a SIP Servlet explicitly requests user profile documents through the Sh provider application. The provider returns an XML document, and the Servlet can then use standard DOM techniques to read or modify profile data in the local document. Updates to the local document are applied to the HSS after a "put" operation.

12.4 Creating a Document Selector Key for Application-Managed Profile Data

The document selector key identifies the XML document to be retrieved by a Diameter interface, and uses the format `protocol://uri/reference_type[/access_key]`. Servlets that manage profile data can explicitly obtain an Sh XML document from a Profile Service using a document selector key, and then work with the document using DOM.

The `protocol` portion of the selector identifies the Diameter interface provider to use for retrieving the document. Sh XML documents require the `sh://` protocol designation.

With Sh document selectors, the next element, `uri`, generally corresponds to the User-Identity or Public-Identity of the user whose profile data is being retrieved. If you are requesting an Sh data reference of type LocationInformation or UserState, the URI value can be the User-Identity or MSISDN for the user.

Table 12–1 summarizes the possible URI values that can be supplied depending on the Sh data reference you are requesting. 3GPP TS 29.328 describes the possible data references and associated reference types in more detail.

Table 12–1 Possible URI Values for Sh Data References

Sh Data Reference Number	Data Reference Type	Possible URI Value in Document Selector
0	RepositoryData	User-Identity or Public-Identity
10	IMSPublicIdentity	
11	IMSUserState	
12	S-CSCFName	
13	InitialFilterCriteria	
14	LocationInformation	User-Identity or MSISDN
15	UserState	
17	Charging information	User-Identity or Public-Identity
17	MSISDN	

The final element of the document selector key, `reference_type`, specifies the data reference type being requested. For some data reference requests, only the `uri` and `reference_type` are required. Other Sh requests use an access key, which requires a third element in the document selector key corresponding to the value of the Attribute-Value Pair (AVP) defined in the document selector key.

Table 12–1 summarizes the required document selector key elements for each type of Sh data reference request.

Table 12–2 Summary of Document Selector Elements for Sh Data Reference Requests

Data Reference Type	Required Document Selector Elements	Example Document Selector
RepositoryData	sh://uri/reference_type/Service-Indication	sh://sip:user@oracle.com/RepositoryData/Call Screening/
IMSPublicIdentity	sh://uri/reference_type/[*Identity-Set*] where *Identity-Set* is one of: ▪ All-Identities ▪ Registered-Identities ▪ Implicit-Identities	sh://sip:user@oracle.com/IMSPublicIdentity/Registered-Identities
IMSUserState	sh://uri/reference_type	sh://sip:user@oracle.com/IMSUserState/
S-CSCFName	sh://uri/reference_type	sh://sip:user@oracle.com/S-CSCFName/
InitialFilterCriteria	sh://uri/reference_type/Server-Name	sh://sip:user@oracle.com/InitialFilterCriteria/www.oracle.com/
LocationInformation	sh://uri/reference_type/(CS-Domain | PS-Domain)	sh://sip:user@oracle.com/LocationInformation/CS-Domain/

Table 12–2 (Cont.) Summary of Document Selector Elements for Sh Data Reference Requests

Data Reference Type	Required Document Selector Elements	Example Document Selector
UserState	sh://uri/reference_type/(CS-Domain | PS-Domain)	sh://sip:user@oracle.com/UserState/PS-Domain/
Charging information	sh://uri/reference_type	sh://sip:user@oracle.com/Charging information/
MSISDN	sh://uri/reference_type	sh://sip:user@oracle.com/MSISDN/

12.5 Using a Constructed Document Key to Manage Profile Data

OWLCS provides a helper class, `com.bea.wcp.profile.ProfileService`, to help you easily retrieve a profile data document. The `getDocument()` method takes a constructed document key, and returns a read-only `org.w3c.dom.Document` object. To modify the document, you make and edit a copy, then send the modified document and key as arguments to the `putDocument()` method.

Note: If Diameter Sh client node services are not available on the OWLCS instance when getDocument() the profile service throws a "No registered provider for protocol" exception.

OWLCS caches the documents returned from the profile service for the duration of the service method invocation (for example, when a `doRequest()` method is invoked). If the service method requests the same profile document multiple times, the subsequent requests are served from the cache rather than by re-querying the HSS.

Example 12–1 shows a sample SIP Servlet that obtains and modifies profile data.

Example 12–1 Sample Servlet Using ProfileService to Retrieve and Write User Profile Data

```
package demo;
import com.bea.wcp.profile.*;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServlet;
import org.w3c.dom.Document;
import java.io.IOException;
public class MyServlet extends SipServlet {
      private ProfileService psvc;
      public void init() {
        psvc = (ProfileService)
getServletContext().getAttribute(ProfileService.PROFILE_SERVICE);
      }
      protected void doInvite(SipServletRequest req) throws IOException {
        String docSel = "sh://" + req.getTo() + "/IMSUserState/";
        // Obtain and change a profile document.
        Document doc = psvc.getDocument(docSel); // Document is read only.
        Document docCopy = (Document) doc.cloneNode(true);
        // Modify the copy using DOM.
        psvc.putDocument(docSel, docCopy); // Apply the changes.
      }
}
```

12.6 Monitoring Profile Data with ProfileListener

The IMS Sh interface enables applications to receive automatic notifications when a subscriber's profile data changes. OWLCS provides an easy-to-use API for managing profile data subscriptions. A SIP Servlet registers to receive notifications by implementing the `com.bea.wcp.profile.ProfileListener` interface, which consists of a single `update` method that is automatically invoked when a change occurs to profile to which the Servlet is subscribed. Notifications are not sent if that same Servlet modifies the profile information (for example, if a user modifies their own profile data).

Note: In a replicated environment, Diameter relay nodes always attempt to push notifications directly to the engine tier server that subscribed for profile updates. If that engine tier server is unavailable, another server in the engine tier cluster is chosen to receive the notification. This model succeeds because session information is stored in the SIP data tier, rather than the engine tier.

12.6.1 Prerequisites for Listener Implementations

In order to receive a call back for subscribed profile data, a SIP Servlet must do the following:

- Implement `com.bea.wcp.profile.ProfileListener`.
- Create one or more subscriptions using the `subscribe` method in the `com.bea.wcp.profile.ProfileService` helper class.
- Register itself as a listener using the `listener` element in `sip.xml`.

"Implementing ProfileListener" describes how to implement `ProfileListener` and use the `susbscribe` method. In addition to having a valid listener implementation, the Servlet must declare itself as a listener in the `sip.xml` deployment descriptor file. For example, it must add a `listener` element declaration similar to:

```
<listener>
   <lisener-class>com.mycompany.MyLisenerServlet</listener-class>
</listener>
```

12.6.2 Implementing ProfileListener

Actual subscriptions are managed using the `subscribe` method of the `com.bea.wcp.profile.ProfileService` helper class. The subscribe method requires that you supply the current `SipApplicationSession` and the key for the profile data document you want to monitor. See "Creating a Document Selector Key for Application-Managed Profile Data".

Applications can cancel subscriptions by calling `ProfileSubscription.cancel()`. Also, pending subscriptions for an application are automatically cancelled if the application session is terminated.

Example 12–2 shows sample code for a Servlet that implements the `ProfileListener` interface.

Example 12–2 Sample Servlet Implementing ProfileListener Interface

```
package demo;
    import com.bea.wcp.profile.*;
    import javax.servlet.sip.SipServletRequest;
    import javax.servlet.sip.SipServlet;
    import org.w3c.dom.Document;
    import java.io.IOException;
    public class MyServlet extends SipServlet implements ProfileListener {
      private ProfileService psvc;
      public void init() {
        psvc = (ProfileService)
getServletContext().getAttribute(ProfileService.PROFILE_SERVICE);
      }
      protected void doInvite(SipServletRequest req) throws IOException {
        String docSel = "sh://" + req.getTo() + "/IMSUserState/";
        // Subscribe to profile data.
        psvc.subscribe(req.getApplicationSession(), docSel, null);
}
      public void update(ProfileSubscription ps, Document document) {
        System.out.println("IMSUserState updated: " + ps.getDocumentSelector());
      }
    }
```

13

Developing Custom Profile Service Providers

This chapter describes how to use the Profile Service API to develop custom profile rovider, in the following sections:

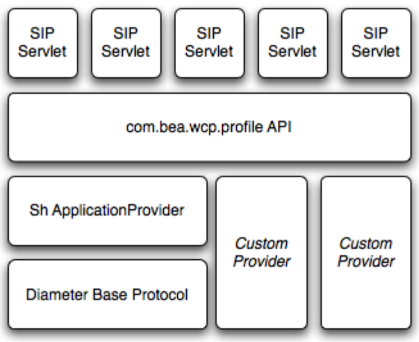
- Section 13.1, "Overview of the Profile Service API"
- Section 13.2, "Implementing Profile Service API Methods"
- Section 13.3, "Configuring and Packaging Profile Providers"
- Section 13.4, "Configuring Profile Providers Using the Administration Console"

13.1 Overview of the Profile Service API

OWLCS includes a profile service API, `com.bea.wcp.profile.API`, that may have multiple profile service provider implementations. A profile provider performs the work of accessing XML documents from a data repository using a defined protocol. Deployed SIP Servlets and other applications need not understand the underlying protocol or the data repository in which the document is stored; they simply reference profile data using a custom URL, and OWLCS delegates the request processing to the correct profile provider.

The provider performs the necessary protocol operations for manipulating the document. All providers work with documents in XML DOM format, so client code can work with many different types of profile data in a common way.

Figure 13–1 Profile Service API and Provider Implementation



Each profile provider implemented using the API may enable the following operations against profile data:

- Creating new documents.
- Querying and updating existing documents.
- Deleting documents.
- Managing subscriptions for receiving notifications of profile document changes.

Clients that want to use a profile provider obtain a profile service instance through a Servlet context attribute. They then construct an appropriate URL and use that URL with one of the available Profile Service API methods to work with profile data. The contents of the URL, combined with the configuration of profile providers, determines the provider implementation that OWLCS to process the client's requests.

The sections that follow describe how to implement the profile service API interfaces in a custom profile provider.

13.2 Implementing Profile Service API Methods

A custom profile providers is implemented as a shared Java EE library (typically a simple JAR file) deployed to the engine tier cluster. The provider JAR file must include, at minimum, a class that implements `com.bea.wcp.profile.ProfileServiceSpi`. This interface inherits methods from `com.bea.wcp.profile.ProfileService` and defines new methods that are called during provider registration and unregistration.

In addition to the provider implementation, you must implement the `com.bea.wcp.profile.ProfileSubscription` interface if your provider supports subscription-based notification of profile data updates. A `ProfileSubscription` is returned to the client subscriber when the profile document is modified.

The Oracle Fusion Middleware WebLogic Communication Services API Reference describes each method of the profile service API in detail. Also keep in mind the following notes and best practices when implementing the profile service interfaces:

- The `putDocument`, `getDocument`, and `deleteDocument` methods each have two distinct method signatures. The basic version of a method passes only the document selector on which to operate. The alternate method signature also passes the address of the sender of the request for protocols that require explicit information about the requestor.
- The `subscribe` method has multiple method signatures to allow passing the sender's address, as well as for supporting time-based subscriptions.
- If you do not want to implement a method in `com.bea.wcp.profile.ProfileServiceSpi`, include a "no-op" method implementation that throws the OperationNotSupportedException.

`com.bea.wcp.profile.ProfileServiceSpi` defines provider methods that are called during registration and unregistration. Providers can create connections to data stores or perform any required initializing in the `register` method. The `register` method also supplies a `ProviderBean` instance, which includes any context parameters configured in the provider's configuration elements in `profile.xml`.

Providers must release any backing store connections, and clean up any state that they maintain, in the `unregister` method.

13.3 Configuring and Packaging Profile Providers

Providers must be deployed as a shared Java EE library, because all other deployed applications must be able to access the implementation.

See "Creating Shared Java EE Libraries and Optional Packages". For most profile providers, you can simply package the implementation classes in a JAR file. Then register the library with OWLCS using the instructions in See "Deploying Shared Java EE Libraries and Dependent Applications".

After installing the provider as a library, you must also identify the provider class as a provider in a `profile.xml` file. The `name` element uniquely identifies a provider configuration, and the `class` element identifies the Java class that implements the profile service API interfaces. One or more context parameters can also be defined for the provider, which are delivered to the implementation class in the `register` method. For example, context parameters might be used to identify backing stores to use for retrieving profile data.

Example 13–1 shows a sample configuration for a provider that accesses data using XCAP.

Example 13–1 Provider Mapping in profile.xml

```
<profile-service xmlns="http://www.bea.com/ns/wlcp/wlss/profile/300"
                 xmlns:sec="http://www.bea.com/ns/weblogic/90/security"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema=instance"
                 xmlns:wls="http;//www.bea.com/ns/weblogic/90/security/wls">
 <mapping>
   <map-by>provider-name</map-by>
 </mapping>
 <provider>
    <name>xcap</name>
    <provider-class>com.mycompany.profile.XcapProfileProvider</provider-class>
    <param>
       <name>server</name>
       <value>example.com</name>
    </param>
    ...
 </provider>
</profile-service>
```

13.3.1 Mapping Profile Requests to Profile Providers

When an application makes a request using the Profile Service API, OWLCS must find a corresponding provider to process the request. By default, OWLCS maps the prefix of the requested URL to a provider `name` element defined in `profile.xml`. For example, with the basic configuration shown in Example 13–1, OWLCS would map Profile Service API requests beginning with `xcap://` to the provider class com.mycompany.profile.XcapProfileProvider.

Alternately, you can define a `mapping` entry in `profile.xml` that lists the prefixes corresponding to each named provider. Example 13–2 shows a mapping with two alternate prefixes.

Example 13–2 Mapping a Provider to Multiple Prefixes

```
...
<mapping>
   <map-by>prefix</map-by>
      <provider>
```

```
            <provider-name>xcap</provider-name>
            <doc-prefix>sip</doc-prefix>
            <doc-prefix>subscribe</doc-prefix>
        </provider>
   <by-prefix>
<mapping>
...
```

If the explicit mapping capabilities of `profile.xml` are insufficient, you can create a custom mapping class that implements the com.bea.wcp.profile.ProfileRouter interface, and then identify that class in the `map-by-router` element. Example 13–3 shows an example configuration.

Example 13–3 Using a Custom Mapping Class

```
...
<mapping>
   <map-by-router>
      <class>com.bea.wcp.profile.ExampleRouter</class>
   </map-by-router>
</mapping>
...
```

13.4 Configuring Profile Providers Using the Administration Console

You can optionally use the Administration Console to create or modify a `profile.xml` file. To do so, you must enable the profile provider console extension in the `config.xml` file for your domain.

Example 13–4 Enabling the Profile Service Resource in config.xml

```
...
<custom-resource>
    <name>ProfileService</name>
    <target>AdminServer</target>
    <descriptor-file-name>custom/profile.xml</descriptor-file-name>

<resource-class>com.bea.wcp.profile.descriptor.resource.ProfileServiceResource</re
source-class>

<descriptor-bean-class>com.bea.wcp.profile.descriptor.beans.ProfileServiceBean</de
scriptor-bean-class>
  </custom-resource>
</domain>
```

The profile provider extension appears under the SipServer node in the left pane of the console, and enables you to configure new provider classes and mapping behavior.

14 Using the Diameter Rf Interface API for Offline Charging

The following chapter describes how to use the Diameter Rf interface API, based on the OWLCS Diameter protocol implementation, in your own applications, and contains the following sections:

14.1 Overview of Rf Interface Support

Offline charging is used for network services that are paid for periodically. For example, a user may have a subscription for voice calls that is paid monthly. The Rf protocol allows an IMS Charging Trigger Function (CTF) to issue offline charging events to a Charging Data Function (CDF). The charging events can either be one-time events or may be session-based.

OWLCS provides a Diameter Offline Charging Application that can be used by deployed applications to generate charging events based on the Rf protocol. The offline charging application uses the base Diameter protocol implementation, and allows any application deployed on OWLCS to act as CTF to a configured CDF.

For basic information about offline charging, see RFC 3588: Diameter Base Protocol (`http://www.ietf.org/rfc/rfc3588.txt`). For more information about the Rf protocol, see 3GPP TS 32.299 (`http://www.3gpp.org/ftp/Specs/html-info/32299.htm`).

14.2 Understanding Offline Charging Events

For both event and session based charging, the CTF implements the accounting state machine described in RFC 3588. The server (CDF) implements the accounting state machine "SERVER, STATELESS ACCOUNTING" as specified in RFC 3588.

The reporting of offline charging events to the CDF is managed through the Diameter Accounting Request (ACR) message. Rf supports the ACR event types described in Table 14–1.

Table 14–1 Rf ACR Event Types

Request	Description
START	Starts an accounting session.
INTERIM	Updates an accounting session.
STOP	Stops an accounting session
EVENT	Indicates a one-time accounting event.

The START, INTERIM, and STOP event types are used for session-based accounting. The EVENT type is used for event based accounting, or to indicate a failed attempt to establish a session.

14.2.1 Event-Based Charging

Event-based charging events are reported through the ACR EVENT message. Example 14–1 shows the basic message flow.

Example 14–1 Message Flow for Event-Based Charging

```
CTF (WLSS)           CDF (Server)
    |                     |
    | --- ACR (EVENT) --> |
    |                     |
    |                (Process accounting request)
    |                     |
    | <-- ACA (EVENT) --- |
    |                     |
```

14.2.2 Session-Based Charging

Session-based charging uses the ACR START, INTERIM, and STOP requests to report usage to the CDF. During a session, the CTF may report multiple ACR INTERIM requests depending on the session lifecycle. Example 14–2 shows the basic message flow

Example 14–2 Message Flow for Session-Based Charging

```
CTF (WLSS)             CDF (Server)
    |                       |
    | --- ACR (START) ----> |
    |                       |
    |                  (Open CDR)
    |                       |
    | <-- ACA (START) ----- |
    |                       |
   ...                     ...
    | --- ACR (INTERIM) --> |
    |                       |
    |                  (Update CDR)
    |                       |
    | <-- ACA (INTERIM) --- |
   ...                     ...
    | --- ACR (STOP) -----> |
    |                       |
    |                  (Close CDR)
    |                       |
    | <-- ACA (STOP) ------ |
    |                       |
```

Here, ACA START is sent a receipt of a service request by OWLCS. ACA INTERIM is typically sent upon expiration of the AII timer. ACA STOP is issued upon request for service termination by OWLCS.

14.3 Configuring the Rf Application

The `Rf` API is packaged as a Diameter application similar to the Sh application used for managing profile data. The Rf Diameter API can be configured and enabled by editing the Diameter configuration file located in `DOMAIN_ROOT/config/custom/diameter.xml`, or by using the Diameter console extension. Additionally, configuration of both the CDF realm and host can be specified using the cdf.realm and cdf.host initialization parameters to the Diameter Rf application.

Example 14–3 shows a sample excerpt from `diameter.xml` that enables Rf with a CDF realm of "oracle.com" and host "cdf.oracle.com:"

Example 14–3 Sample Rf Application Configuration (diameter.xml)

```
<application>
  <application-id>3</application-id>
  <accounting>true</accounting>
  <class-name>com.bea.wcp.diameter.charging.RfApplication</class-name>
  <param>
    <name>cdf.realm</name>
    <value>oracle.com</value>
  </param>
  <param>
    <name>cdf.host</name>
    <value>cdf.oracle.com</value>
  </param>
</application>
```

Because the `RfApplication` uses the Diameter base accounting messages, its Diameter application id is 3 and there is no vendor ID.

14.4 Using the Offline Charging API

OWLCS provides an offline charging API to enable any deployed application to act as a CTF and issue offline charging events. This API supports both event-based and session-based charging events.

The classes in package `com.bea.wcp.diameter.accounting` provide general support for Diameter accounting messages and sessions. Table 14–2 summarizes the classes.

Table 14–2 Diameter Accounting Classes

Class	Description
ACR	An Accounting-Request message.
ACA	An Accounting-Answer message.
ClientSession	A Client-based accounting session.
RecordType	Accounting record type constants.

In addition, classes in package `com.bea.wcp.diameter.charging` support the Rf application specifically. Table 14–3 summarizes the classes.

Table 14–3 Diameter Rf Application Support Classes

Charging	Common definitions for 3GPP charging functions
RfApplication	Offline charging application
RfSession	Offline charging session

The `RfApplication` class can be used to directly send ACR requests for event-based charging. The application also has the option of directly modifying the ACR request before it is sent out. This is necessary in order for an application to add any custom AVPs to the request.

In particular, an application must set the Service-Information AVP it carries the service-specific parameters for the CDF. The Service-Information AVP of the ACR request is used to send the application-specific charging service information from the CTF (WLSS) to the CDF (Charging Server). This is a grouped AVP whose value depends on the application and its charging function. The Offline Charging API allows the application to set this information on the request before it is sent out.

For session-based accounting, the `RfApplication` class can also be used to create new accounting sessions for generating session-based charging events. Each accounting session is represented by an instance of `RfSession`, which encapsulates the accounting state machine for the session.

14.4.1 Accessing the Rf Application

If the Rf application is deployed, then applications deployed on OWLCS can obtain an instance of the application from the Diameter node (`com.bea.wcp.diameter.Node` class). Example 14–4 shows the sample Servlet code used to obtain the Diameter `Node` and access the Rf application.

Example 14–4 Accessing the Rf Application

```
ServletContext sc = getServletConfig().getServletContext();
Node node = sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication) node.getApplication(Charging.RF_APPLICATION_
ID);
```

Applications can safely use a single instance of `RfApplication` to issue offline charging requests concurrently, in multiple threads. Each instance of `RfSession` actually holds the per-session state unique to each call.

14.4.2 Implementing Session-Based Charging

For session-based charging requests, an application first uses the `RfApplication` to create an instance of `RfSession`. The application can then use the session object to create one or more charging requests.

The first charging request must be an ACR START request, followed by zero or more ACR INTERIM requests. The session ends with an ACR STOP request. Upon receipt of the corresponding ACA STOP message, the `RfApplication` automatically terminates the `RfSession`.

Example 14–5 shows the sample code used to start a new session-based accounting session.

Example 14–5 Starting a Session-Based Account Session

```
  RfSession session = rfApp.createSession();
```

```
sipRequest.getApplicationSession().setAttribute("RfSession", session);
ACR acr = session.createACR(RecordType.START);
acr.addAvp(Charging.SERVICE_INFORMATION, ...);
ACA aca = acr.sendAndWait(1000);
if (!aca.getResultCode().isSuccess()) {
  ... error ...
}
```

In Example 14–5, the `RfSession` is stored as a SIP application session attribute so that it can be used to send additional accounting requests as the call progresses. Example 14–6 shows how to send an INTERIM request.

Example 14–6 Sending an INTERIM request

```
  RfSession session = (RfSession)
req.getApplicationSession().getAttribute("RfSession");
  ACR acr = session.createACR(RecordType.INTERIM);
  ACA aca = acr.sendAndWait(1000);
  if (!aca.getResultCode().isSuccess()) {
    ... error ...
  }
```

An application may want to send one or more ACR INTERIM requests while a call is in progress. The frequency of ACR INTERIM requests is usually based on the Acct-Interim-Interval AVP value in the ACA START message sent by the CDF. For this reason, an application timer must be used to send ACR INTERIM requests at the requested interval. See 3GPP TS 32.299 for more details about interim requests.

14.4.2.1 Specifying the Session Expiration

The Acct-Interim-Interval (AII) timer value is used to indicate the expiration time of an Rf accounting session. It is specified when ACR START is sent to the CDF to initiate the accounting session. The CDF responds with its own AII value, which must be used by the CTF to start a timer upon whose expiration an ACR INTERIM message must be sent. This INTERIM message informs the CDF that the session is still in use. Otherwise, the CDF terminates the session automatically.

It is the application's responsibility to send ACR INTERIM messages, because these are used to send updated Service-Information data to the CDF. Oracle recommends creating a ServletTimer that is set to expire according to the AII value. When the timer expires, the application must send an ACR INTERIM message with the updated service information data.

14.4.2.2 Sending Asynchronous Events

Applications generally use the synchronous `sendAndWait()` method. However, if latency is critical, an asynchronous API is provided wherein the application Servlet is asynchronously notified when an answer message is received from the CDF. To use the asynchronous API, an application first registers an instance of `SessionListener` in order to asynchronously receive messages delivered to the session, as shown in Example 14–7.

Example 14–7 Registering a SessionListener

```
RfSession session = rfApp.createSession();
session.setAttribute("SAS", sipReq.getApplicationSession());
session.setListener(this);
```

Attributes can be stored in an `RfSession` instance similar to the way SIP application session attributes are stored. In the above example, the associated SIP application was stored as an `RfSession` so that it is available to the listener callback.

When a Diameter request or answer message is received from the CDF, the application Servlet is notified by calling the `rcvMessage(Message msg)` method. The associated SIP application session can then be retrieved from the `RfSession` if it was stored as a session attribute, as shown in Example 14–8.

Example 14–8 Retrieving the RfSession after a Notification

```
  public void rcvMessage(Message msg) {
    if (msg.getCommand() != Command.ACA) {
      if (msg.isRequest()) {
        ((Request) msg).createAnswer(ResultCode.UNABLE_TO_COMPLY, "Unexpected
request").send();
      }
      return;
    }
    ACA aca = (ACA) msg;
    RfSession session = (RfSession) aca.getSession();
    SipApplicationSession appSession = (SipApplicationSession)
session.getAttribute("SAS");
    ...
  }
```

14.4.3 Implementing Event-Based Charging

For an event-based charging request, the charging request is a one-time event and the session is automatically terminated upon receipt of the corresponding EVENT ACA message. The `sendAndWait(long timeout)` method can be used to synchronously send the EVENT request and block the thread until a response has been received from the CDF. Example 14–9 shows an example that uses an `RfSession` for sending an event-based charging request.

Example 14–9 Event-Based Charging Using RfSession

```
  RfSession session = rfApp.createSession();
  ACR acr = session.createACR(RecordType.EVENT);
  acr.addAvp(Charging.SERVICE_INFORMATION, ...);
  ACA aca = acr.sendAndWait(1000);
  if (!aca.getResultCode().isSuccess()) {
    ... send error response ...
  }
```

For convenience, it is also possible send event-based charging requests using the `RfApplication` directly, as shown in Example 14–10.

Example 14–10 Event-Based Charging Using RfApplication

```
  ACR acr = rfApp.createEventACR();
  acr.addAvp(Charging.SERVICE_INFORMATION, ...);
  ACA aca = acr.sendAndWait(1000);
```

Internally, the `RfApplication` creates an instance of `RfSession` associated with the ACR request, so this method is equivalent to creating the session explicitly.

For both session and event based accounting, the `RfSession` class automatically handles creating session IDs, as well as updating the Accounting-Record-Number AVP used to sequence messages within the same accounting session.

In the above cases the applications waits for up to 1000 ms to receive an answer from the CDF. If no answer is received within that time, the Diameter core delivers an UNABLE_TO_COMPLY error response to the application, and cancels the request. If no timeout is specified with `sendAndWait()`, then the default request timeout of 30 seconds is used. This default value can be configured using the Diameter console extension.

14.4.4 Using the Accounting Session State

The accounting session state for offline charging is serializable, so it can be stored as a SIP application session attribute. Because the client APIs are synchronous, it is not necessary to maintain any state for the accounting session once the Servlet has finished handling the call.

For event-based charging events it is not necessary for the application to maintain any accounting session state because it is only used internally, and is disposed once the ACA response has been received.

15 Using the Diameter Ro Interface API for Online Charging

The following chapter describes how to use the Diameter Ro interface API, based on the OWLCS Diameter protocol implementation, in your own applications, and contains the following sections:

15.1 Overview of Ro Interface Support

Online charging, also known as credit-based charging, is used to charge prepaid services. A typical example of a prepaid service is a calling card purchased for voice or video. The Ro protocol allows a Charging Trigger Function (CTF) to issue charging events to an Online Charging Function (OCF). The charging events can be immediate, event-based, or session-based.

OWLCS provides a Diameter Online Charging Application that deployed applications can use to generate charging events based on the Ro protocol. This enables deployed applications to act as CTF to a configured OCF. The Diameter Online Charging Application uses the base Diameter protocol that underpins both the Rf and Sh applications.

The Diameter Online Charging Application is based on IETF RFC 4006: Diameter Credit Control Application (`http://www.ietf.org/rfc/rfc4006.txt`). However, the application supports only a subset of the RFC 4006 required for compliance with 3GPP TS 32.299: Telecommunication management; Charging management; Diameter charging applications (`http://www.3gpp.org/ftp/Specs/html-info/32299.htm`). Specifically, the OWLCS Diameter Online Charging Application provides no direct support for service-specific Attribute-Value Pairs (AVPs), but the API that is provided is flexible enough to allow applications to include custom service-specific AVPs in any credit control request.

15.2 Understanding Credit Authorization Models

RFC 4006 defines two basic types of credit authorization models:

- Credit authorization with unit reservation, and
- Credit authorization with direct debiting.

Credit authorization with unit reservation can be performed with either event-based or session-based charging events. Credit authorization with direct debiting uses immediate charging events. In both models, the CTF requests credit authorization from the OCF prior to delivering services to the end user. In both models

The sections that follow describe each model in more detail.

15.2.1 Credit Authorization with Unit Determination

RFC 4006 defines both Event Charging with Unit Reservation (ECUR) and Session Charging with Unit Reservation (SCUR). Both charging events are session-based, and require multiple transactions between the CTF and OCF. ECUR begins with an interrogation to reserve units before delivering services, followed by an additional interrogation to report the actual used units to the OCF upon service termination. With SCUR, it is also possible to include one or more intermediate interrogations for the CTF in order to report currently-used units, and to reserve additional units if required. In both cases, the session state is maintained in both the CTF and OCF.

For both ECUR and SCUR, the online charging client implements the "CLIENT, SESSION BASED" state machine described in RFC 4006.

15.2.2 Credit Authorization with Direct Debiting

For direct debiting, Immediate Event Charging (IEC) is used. With IEC, a single transaction is created where the OCF deducts a specific amount from the user's account immediately after completing the credit authorization. After receiving the authorization, the CTF delivers services. This form of credit authorization is a one-time event in which no session state is maintained.

With IEC, the online charging client implements the "CLIENT, EVENT BASED" state machine described in IETF RFC 4006.

15.2.3 Determining Units and Rating

Unit determination refers to calculating the number of non-monetary units (service units, time, events) that can be assigned prior to delivering services. Unit rating refers to determining a price based on the non-monetary units calculated by the unit determination function.

It is possible for either the OCF or the CTF to handle unit determination and unit rating. The decision lies with the client application, which controls the selection of AVPs in the credit control request sent to the OCF.

15.3 Configuring the Ro Application

The `RoApplication` is packaged as a Diameter application similar to the Sh application used for managing profile data. The Ro Diameter application can be configured and enabled by editing the Diameter configuration file located in `DOMAIN_ROOT/config/custom/diameter.xml`, or by using the Diameter console extension.

The application init parameter ocs.host specifies the host identity of the OCF. The OCF host must also be configured in the peer table as part of the global Diameter configuration. Alternately, the init parameter ocs.realm can be used to specify more than one OCF host using realm-based routing. The corresponding realm definition must also exist in the global Diameter configuration.

Example 15–1 shows a sample excerpt from diameter.xml that enables Ro with an OCF host name of "myocs.oracle.com."

Example 15–1 Sample Ro Application Configuration (diameter.xml)

```
<application>
  <application-id>4</application-id>
  <class-name>com.bea.wcp.diameter.charging.RoApplication</class-name>
  <param>
    <name>ocs.host</name>
    <value>myocs.oracle.com</value>
  </param>
</application>
```

Because the RoApplication is based on the Diameter Credit Control Application, its Diameter application id is 4.

15.4 Overview of the Online Charging API

OWLCS provides an online charging API to enable any deployed application to act as a CTF and issue online charging events to an OCS through the Ro protocol. All online charging requests use the Diameter Credit-Control-Request (CCR) message. The CC-Request-Type AVP is used to indicate the type of charging used. In the charging API, the CC-Request-Type is represented by the RequestType class in package com.bea.wcp.diameter.cc. Table 15–1 shows the request types associated with different credit authorization models.

Table 15–1 Credit Control Request Types

Type	Description	RequestType Field in com.bea.wcp.diameter.cc.RequestType
IEC	Immediate Event Charging	EVENT_REQUEST
ECUR	Event Charging with Unit Reservation	INITIAL or TERMINATION_REQUEST
SCUR	Session Charging with Unit Reservation	INITIAL, UPDATE, or TERMINATION_ REQUEST

For ECUR and SCUR, units are reserved prior to service delivery and committed upon service completion. Units are reserved with INITIAL_REQUEST and committed with a TERMINATION_REQUEST. For SCUR, units can also be updated with UPDATE_ REQUEST.

The base diameter package, com.bea.wcp.diameter, contains classes to support the re-authorization requests used in Ro. The com.bea.wcp.diameter.cc package contains classes to support credit-control applications, including Ro applications. com.bea.wcp.diameter.charging directly supports the Ro credit-control application. Table 15–2 summarizes the classes of interest to Ro credit-control.

Table 15–2 Summary of Ro Classes

Class	Description	Package
`Charging`	Constant definitions	`com.bea.wcp.diameter.charging`
`RoApplication`	Online charging application	`com.bea.wcp.diameter.charging`
`RoSession`	Online charging session	`com.bea.wcp.diameter.charging`
`CCR`	Credit Control Request	`com.bea.wcp.diameter.cc`
`CCA`	Credit Control Answer	`com.bea.wcp.diameter.cc`
ClientSession	Credit control client session	`com.bea.wcp.diameter.cc`
RequestType	Credit-control request type	`com.bea.wcp.diameter.cc`
RAR	Re-Auth-Request message	`com.bea.wcp.diameter`
RAA	Re-Auth-Answer message	`com.bea.wcp.diameter`

15.5 Accessing the Ro Application

If the Ro application is deployed, then applications deployed on OWLCS can obtain an instance of the application from the Diameter node (`com.bea.wcp.diameter.Node` class). Example 15–2 shows the sample Servlet code used to obtain the Diameter `Node` and access the Ro application.

Example 15–2 Accessing the Ro Application

```
private RoApplication roApp;
void init(ServletConfig conf) {
    ServletContext ctx = conf.getServletContext();
    Node node = (Node) ctx.getParameter("com.bea.wcp.diameter.Node");
    roApp = node.getApplication(Charging.RO_APPLICATION_ID);
  }
```

This code example would make `RoApplication` available to the Servlet as an instance variable. The instance of `RoApplication` is safe for use by multiple concurrent threads.

15.6 Implementing Session-Based Charging

The `RoApplication` can be used to create new sessions for session-based credit authorization. The `RoSession` class implements the appropriate state machine depending on the credit control type, either ECUR (Event-Based Charging with Unit Reservation) or SCUR (Session-based Charging with Unit Reservation). The `RoSession` class is also serializable, so it can be stored as a SIP session attribute. This allows the session to be restored when necessary to terminate the session or update credit authorization.

The example in Example 15–3 creates a new `RoSession` for event-based charging, and sends a CCR request to start the first interrogation. The `RoSession` instance is saved so that it can be terminated later, after the service has finished.

Note that the `RoSession` class automatically handles creating session IDs; the application is not required to set the session ID.

Example 15–3 Creating and Using a RoSession

```
RoSession session = roApp.createSession();
CCR ccr = session.createCCR(RequestType.INITIAL);
CCA cca = ccr.sendAndWait();
sipAppSession.setAttribute("RoSession", session);
...
```

15.6.1 Handling Re-Auth-Request Messages

The OCS may initiate credit re-authorization by issuing a Re-Auth-Request (RAR) to the CTF. The application can register a session listener for handling this type of request. Upon receiving a RAR, the Diameter subsystem invoke the session listener on the applications corresponding `RoSession` object. The application must then respond to the OCS with an appropriate RAA message and initiate credit re-authorization to the CTF by sending a CCR with the CC-Request-Type AVP set to the value UPDATE_REQUEST, as described in section 5.5 of RFC 4006 (`http://www.ietf.org/rfc/rfc4006.txt`).

A session listener must implement the `SessionListener` interface and be serializable, or it must be an instance of `SipServlet`. A Servlet can register a listener as follows:

```
RoSession session = roApp.createSession();
session.addListener(new SessionListener() {
  public void rcvMessage(Message msg) {
    System.out.println("Got message: id = " msg.getSession().getId());
  }
});
```

Example 15–4 shows sample `rcvMessage()` code for processing a Re-Auth-Request.

Example 15–4 Managing a Re-Auth-Request

```
RoSession session = roApp.createSession();
session.addListener(new SessionListener() {
public void rcvMessage(Message msg) {
  Request req = (Request)msg;
  if (req.getCommand() != Command.RE_AUTH_REQUEST) return;
  RoSession session = (RoSession) req.getSession();
  Answer ans = req.createAnswer();
  ans.setResultCode(ResultCode.LIMITED_SUCCESS); // Per RFC 4006 5.5
  ans.send();
  CCR ccr = session.createCCR(Ro.UPDATE_REQUEST);
  ... // Set CCR AVPs according to requested credit re-authorization
  ccr.send();
  CCA cca = (CCA) ccr.waitForAnswer();
}
```

In Example 15–4, upon receiving the Re-Auth-Request the application sends an RAA with the result code DIAMETER_LIMITED_SUCCESS to indicate to the OCS that an additional CCR request is required in order to complete the procedure. The CCR is then sent to initiate credit re-authorization.

Note: Because the Diameter subsystem locks the call state before delivering the request to the corresponding RoSession, the call state remains locked while the handler processes the request.

15.7 Sending Credit-Control-Request Messages

The CCR class represents a Diameter Credit-Control-Request message, and can be used to send credit control requests to the OCF. For both ECUR (Event-Based Charging with Unit Reservation) and SCUR (Session-Based Charging with Unit Reservation), an instance of `RoSession` is used to create new CCR requests. You can also use `RoApplication` directly to create CCR messages for IEC (Immediate Event Charging). Example 15–5 shows an example of how to create and send a CCR.

Example 15–5 Creating and Sending a CCR

```
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setServiceContextId("sample_id");
CCA cca = ccr.sendAndWait();
```

Once a CCR request is created, you can set whatever application- or service-specific AVPs that are required before sending the request using the `addAvp()` method. Because some of the same AVPs need to be included in each new request for the session, it is also possible to set these AVPs on the session itself. Example 15–6 shows a sample that sets:

- Subscription-Id to identify the user for the session
- Service-Identifier to indicate the service requested, and
- Requested-Service-Unit to specify the units requested.

A custom AVP is also added directly to the CCR request.

Example 15–6 Setting AVPs in the CCR

```
session.setSubscriptionId(...);
session.setServiceIdentifier(...);
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setRequestedServiceUnit(...);
ccr.addAvp(CUSTOM_MESSAGE, "This is a test");
ccr.send();
```

In this case, the same Subscription-Id and Service-Identifier are added to every new request for the session. The custom AVP "Custom-Message" is added to the message before it is sent out.

15.8 Handling Failures

Applications can examine the Result-Code AVP in CCA error responses from the OCF to detect the cause of a failure and take an appropriate action. Locally-generated errors, such as an unavailable peer or invalid route specification, cause the request send method to throw an `IOException` to with a detailed message indicating the nature of the failure.

Applications can also use the Diameter Timer Tx value for determining when the OCF fails to respond to a credit authorization request. Timer Tx has a default value of 10 seconds, but can be overridden using the `tx.timer` init parameter in the `RoApplication` configuration. Timer Tx starts when a CCR is sent to the OCF. The timer resets after the corresponding CCA is received.

If Tx expires before a corresponding CCA arrives, any call to `waitForAnswer` immediately returns null to indicate that the request has timed out. An application can then take action according to the value of the Credit-Control-Failure-Handling (CCFH) AVP in the request. See section 5.7, "Failure Procedures" in RFC 4006 (`http://www.ietf.org/rfc/rfc4006.txt`) for more details.

Example 15–7 terminates the credit control session if timer Tx expires before receiving the CCA. If the CCA is received later by the Diameter subsystem, the message is ignored because the session longer exists.

Example 15–7 Checking for Timer Tx Expiry

```
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setCreditControlFailureHandling(RequestType.TERMINATION);
ccr.send();
CCA cca = ccr.waitForAnswer();
if (cca == null) {
  session.terminate();
}
```

Part VI

Using Oracle User Messaging Service

This part describes how to use Oracle User Messaging Service.

This part contains the following chapters:

16 Oracle User Messaging Service

This chapter describes Oracle User Messaging Service (UMS).

This chapter includes the following topic:

- Section 16.1, "User Messaging Service Overview"

16.1 User Messaging Service Overview

Oracle User Messaging Service enables two-way communication between users and deployed applications. Key features include:

- Support for a variety of messaging channels—Messages can be sent and received through Email, IM (XMPP), SMS (SMPP), and Voice. Messages can also be delivered to a user's SOA/WebCenter Worklist.
- Two-way Messaging—In addition to sending messages from applications to users (referred to as *outbound* messaging), users can initiate messaging interactions (inbound messaging). For example, a user can send an email or text message to a specified address; the message is routed to the appropriate application which can then respond to the user or invoke another process according to its business logic.
- User Messaging Preferences—End users can use a web interface to define preferences for how and when they receive messaging notifications. Applications immediately become more flexible; rather than deciding whether to send to a user's email address or instant messaging client, the application can simply send the message to the user, and let UMS route the message according to the user's preferences.
- Robust Message Delivery—UMS keeps track of delivery status information provided by messaging gateways, and makes this information available to applications so that they can respond to a failed delivery. Or, applications can specify one or more *failover* addresses for a message in case delivery to the initial address fails. Using the failover capability of UMS frees application developers from having to implement complicated retry logic.
- Pervasive integration within Fusion Middleware: UMS is integrated with other Fusion Middleware components providing a single consolidated bi-directional user messaging service.
  - Integration with Oracle BPEL—Oracle JDeveloper includes pre-built BPEL activities that enable messaging operations. Developers can add messaging capability to a SOA composite application by dragging and dropping the desired activity into any workflow.

- Integration with Oracle Human Workflow—UMS enables the Human Workflow engine to send actionable messages to and receive replies from users over email.
- Integration with Oracle BAM—Oracle BAM uses UMS to send email alerts in response to monitoring events.
- Integration with Oracle WebCenter—UMS APIs are available to developers building applications for Oracle WebCenter Spaces. The API is a realization of Parlay X Web Services for Multimedia Messaging, version 2.1, a standard web service interface for rich messaging.

16.1.1 Components

There are three types of components that make up Oracle User Messaging Service. These components are standard Java EE applications, making it easy to deploy and manage them using the standard tools provided with Oracle WebLogic Server.

- UMS Server: The UMS Server orchestrates message flows between applications and users. The server routes outbound messages from a client application to the appropriate driver, and routes inbound messages to the correct client application. The server also maintains a repository of previously sent messages in a persistent store, and correlates delivery status information with previously sent messages.
- UMS Drivers: UMS Drivers connect UMS to the messaging gateways, adapting content to the various protocols supported by UMS. Drivers can be deployed or undeployed independently of one another depending on what messaging channels are available in a given installation.
- UMS Client applications: UMS client applications implement the business logic of sending and receiving messages. A UMS client application might be a SOA application that sends messages as one step of a BPEL workflow, or a WebCenter Spaces application that can send messages from a web interface.

In addition to the components that make up UMS itself, the other key entities in a messaging environment are the external gateways required for each messaging channel. These gateways are not a part of UMS or Oracle WebLogic Server. Since UMS Drivers support widely-adopted messaging protocols, UMS can be integrated with existing infrastructures such as a corporate email servers or XMPP (Jabber) servers. Alternatively, UMS can connect to outside providers of SMS or text-to-speech services that support SMPP or VoiceXML, respectively.

16.1.2 Architecture

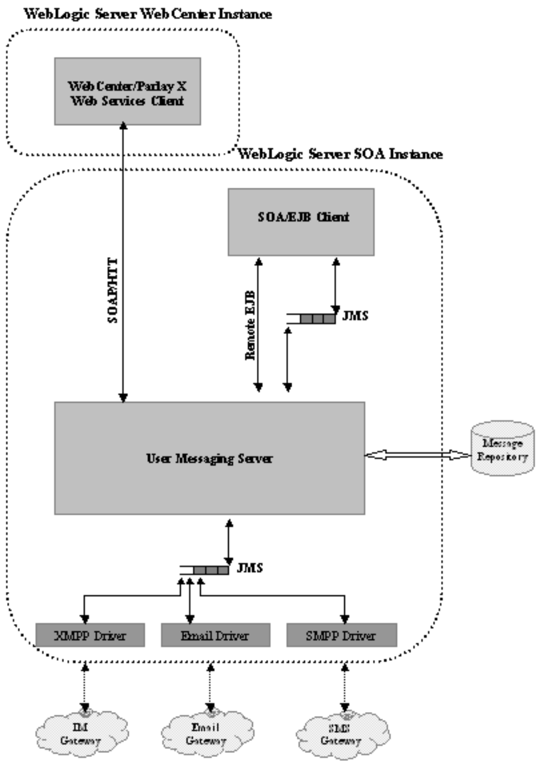
The system architecture of Oracle User Messaging Service is shown in Figure 16–1.

For maximum flexibility, the components of UMS are separate Java EE applications. This allows them to be deployed and managed independently of one another. For example, a particular driver can be stopped and reconfigured without affecting message delivery on all other channels.

Exchanges between UMS client applications and the UMS Server occur as SOAP/HTTP web service requests for web service clients, or through Remote EJB and JMS calls for BPEL messaging activities. Exchanges between the UMS Server and UMS Drivers occur through JMS queues.

Oracle UMS server and drivers are installed alongside SOA or BAM in their respective WebLogic Server instances. A WebCenter installation includes the necessary libraries to act as a UMS client application, invoking a server deployed in a SOA instance.

Figure 16–1 UMS architecture



17 Sending and Receiving Messages using the User Messaging Service Java API

This chapter describes how to use the User Messaging Service (UMS) API to develop applications, and describes how to build two sample applications, usermessagingsample.ear and usermessagingsample-echo.ear. It contains the following topics:

17.1 Overview of UMS Java API

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

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

17.1.1 Creating a J2EE Application Module

There are two choices for a J2EE application module that uses the UMS EJB Client API:

- EJB Application Module - Stateless Session Bean - This is a backend, core message-receiving or message-sending application.
- Web Application Module - This is for applications that have an HTML or Web frontend.

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 Oracle Fusion Middleware User Messaging Service API Reference.

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

17.2 Creating a UMS Client Instance

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

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

17.2.1 Creating a MessagingEJBClient Instance Using a Programmatic or Declarative Approach

Example 17–1 shows code for creating a MessagingEJBClient instance using the programmatic approach:

Example 17–1 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 allows you to make changes at deployment time.

You must specify all the required Application Info properties as environment entries in your J2EE module's descriptor (ejb-jar.xml, or web.xml).

Example 17–2 shows code for creating a MessagingEJBClient instance using the declarative approach:

Example 17–2 Declarative Approach to Creating a MessagingEJBClient Instance

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

17.2.2 API Reference for Class MessagingClientFactory

The API reference for class MessagingClientFactory can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3 Sending a Message

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

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.

17.3.1 Creating a Message

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

17.3.1.1 Creating a Plaintext Message

Example 17–3 shows how to create a plaintext message using the UMS Java API.

Example 17–3 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");
```

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

Example 17–4 shows how to create a multipart or alternative message using the UMS Java API.

Example 17–4 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");
```

17.3.1.3 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 17–5 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 must contain one or more values of this header. The value of this header must be the name of a valid delivery type. Refer to the available values for *DeliveryType* in the enum DeliveryType.

Example 17–5 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);
```

17.3.2 API Reference for Class MessageFactory

The API reference for class MessageFactory can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3.3 API Reference for Interface Message

The API reference for interface Message can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3.4 API Reference for Enum DeliveryType

The API reference for enum DeliveryType can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3.5 Addressing a Message

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

17.3.5.1 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.

17.3.5.2 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.

17.3.5.2.1 Creating a Single Address Object Example 17–6 shows code for creating a single Address object:

Example 17–6 Creating a Single Address Object

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

17.3.5.2.2 Creating Multiple Address Objects in a Batch Example 17–7 shows code for creating multiple Address objects in a batch:

Example 17–7 Creating Multiple Address Objects in a Batch

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

17.3.5.2.3 Adding Sender or Recipient Addresses to a Message Example 17–8 shows code for adding sender or recipient addresses to a message:

Example 17–8 Adding Sender or Recipient Addresses to a Message

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

17.3.5.3 Creating a Recipient with a Failover Address

Example 17–9 shows code for creating a recipient with a Failover address:

Example 17–9 Creating a Single Address Object with Failover

```
String recipientWithFailoverStr = "Email:john.doe@oracle.com,
IM:jabber|john.doe@oracle.com";
Address recipient =
AddressFactory.getInstance().createAddress(recipientWithFailoverStr);
```

17.3.5.4 API Reference for Class AddressFactory

The API reference for class AddressFactory can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3.5.5 API Reference for Interface Address

The API reference for interface Address can be accessed from the Oracle Fusion Middleware User Messaging Service API Reference.

17.3.6 Retrieving Message Status

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

17.3.6.1 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.
```

17.3.6.2 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.

17.4 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.

17.4.1 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 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);
```

17.4.2 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 non-blocking 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.

17.4.3 Asynchronous Receiving

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

17.4.4 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 get 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 desired 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 e-mail address "spammer@foo.com":

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

17.5 Using the UMS EJB 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 EJB 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 e-mail client.

Of the two application modules choices described in Section 17.1.1, "Creating a J2EE 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 App module from the provided usermessagingsample-src.zip source. The servlets uses the UMS EJB Client API to create an UMS EJB Client instance (which in turn registers the application's info) 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.
  - *usermessagingclient-ejb.jar* -- Contains the Message EJB 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 pre-built sample application, and the source code (usermessagingsample-src.zip) are available on OTN.

17.5.1 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 17.5.2, "Configuring the E-Mail Driver"
2. Section 17.5.3, "Using JDeveloper 11g to Build the Application"
3. Section 17.5.4, "Deploying the Application"
4. Section 17.5.5, "Testing the Application"

17.5.2 Configuring the E-Mail Driver

To enable the Oracle User Messaging Service's E-Mail Driver to perform outbound messaging and status retrieval, configure the E-Mail 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.

17.5.3 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:

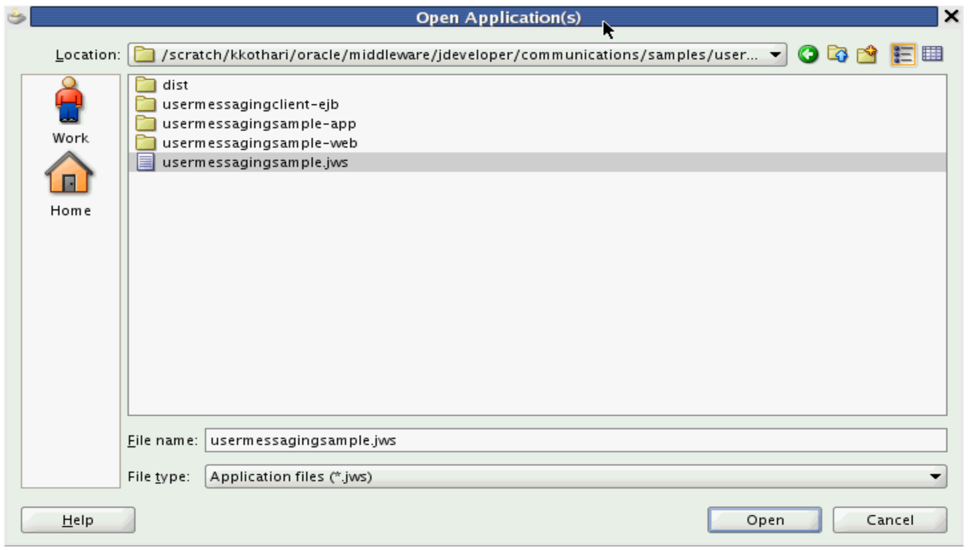
17.5.3.1 Opening the Project

1. Unzip *usermessagingsample-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_11.1.1/sdpmessaging.jar.

2. Open *usermessagingsample.jws* (contained in the .zip file) in Oracle JDeveloper.

Figure 17–1 Oracle JDeveloper Main Window



Location: /scratch/kkothari/oracle/middleware/jdeveloper/communications/samples/user...

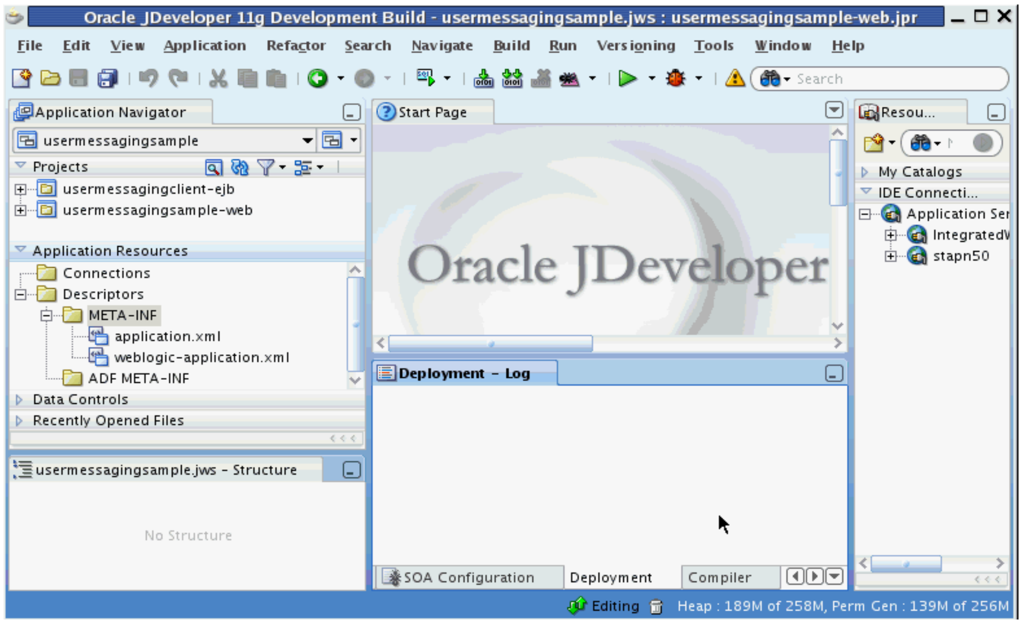
dist
usermessagingclient-ejb
usermessagingsample-app
usermessagingsample-web
usermessagingsample.jws

File name: usermessagingsample.jws

File type: Application files (*.jws)

In the Oracle JDeveloper main window the project appears.

Figure 17–2 Oracle JDeveloper Main Window



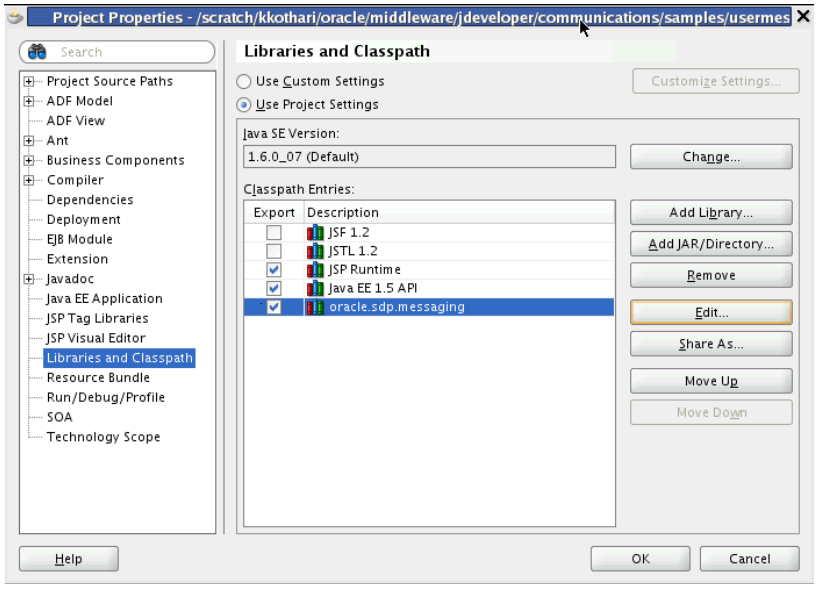
3. Verify that the build dependencies for the sample application have been satisfied by checking that the following library has been added to the Web module.

   - 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.

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

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

Figure 17–3 Verifying Libraries



Libraries and Classpath

( ) Use Custom Settings
(●) Use Project Settings

Java SE Version: 1.6.0_07 (Default)

Classpath Entries:

Export	Description
☐	JSF 1.2
☐	JSTL 1.2
✓	JSP Runtime
✓	Java EE 1.5 API
✓	oracle.sdp.messaging

   3. Click **OK**.

4. Verify that the *usermessagingclient-ejb* project exists in the application. This is an EJB 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 *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 17–10).

Example 17–10 Application Information

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

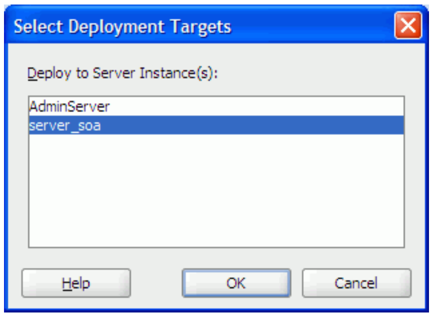
17.5.4 Deploying the Application

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 17.7, "Creating a New Application Server Connection".

2. Deploy the application by selecting the **usermessagingsample application**, **Deploy**, **usermessagingsample**, **to**, and **SOA_server** (Figure 17–4).

Figure 17–4 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 Messaging Preferences.

> **Note:** Refer to "Configuring Notifications" in the *Oracle Fusion Middleware SOA Developer's Guide* for more information.

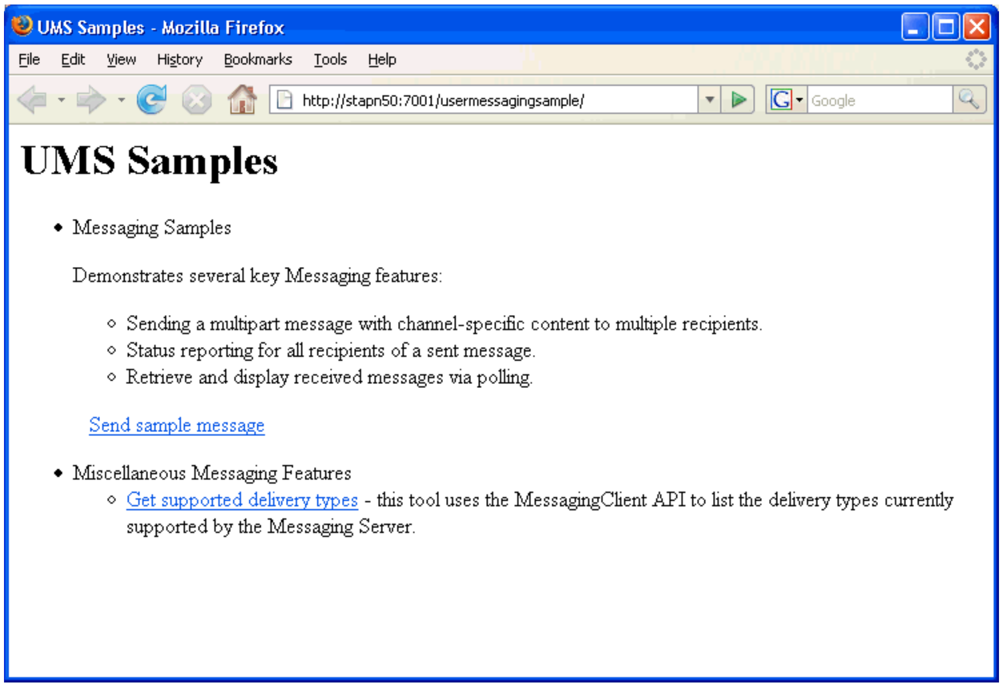
17.5.5 Testing the Application

Once *usermessagingsample* has been deployed to a running instance of WLS, 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 17–5).

Figure 17–5 Testing the Sample Application


UMS Samples - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50:7001/usermessagingsample/
Google

UMS Samples

- Messaging Samples

  Demonstrates several key Messaging features:

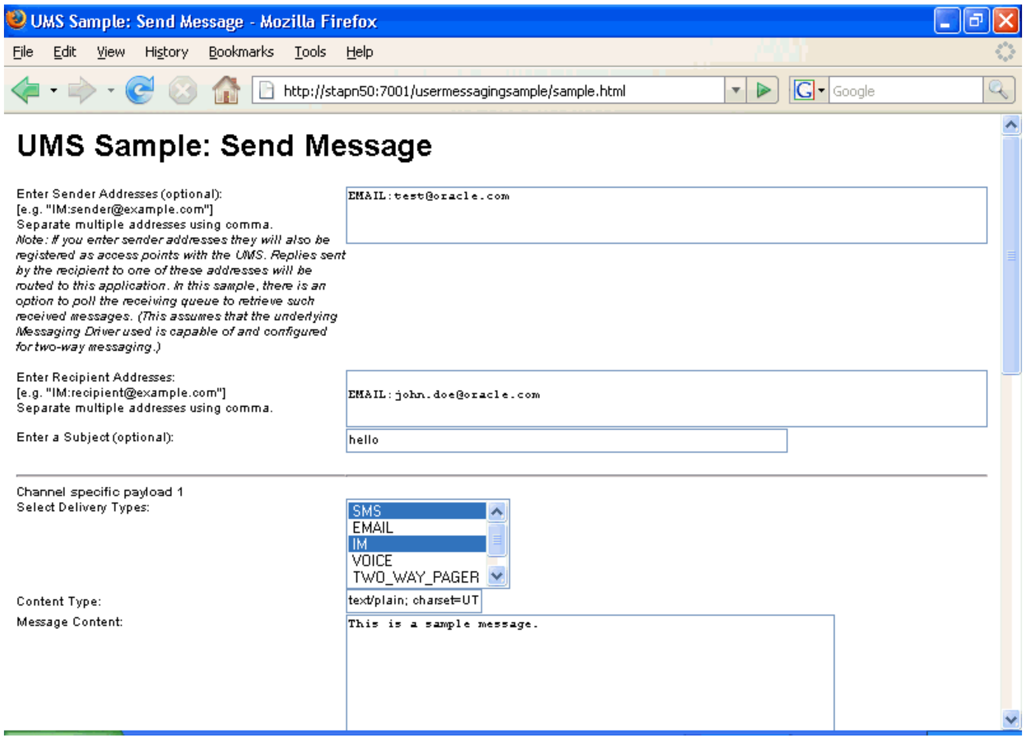
  - Sending a multipart message with channel-specific content to multiple recipients.
  - Status reporting for all recipients of a sent message.
  - Retrieve and display received messages via polling.

  Send sample message

- Miscellaneous Messaging Features
  - Get supported delivery types - this tool uses the MessagingClient API to list the delivery types currently supported by the Messaging Server.

2. Click **Send sample message**. The *Send Message* page appears (Figure 17–6).

Figure 17–6 Addressing the Test Message


UMS Sample: Send Message - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50:7001/usermessagingsample/sample.html
Google

UMS Sample: Send Message

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.)*	EMAIL:test@oracle.com
Enter Recipient Addresses: [e.g. "IM:recipient@example.com"] Separate multiple addresses using comma.	EMAIL:john.doe@oracle.com
Enter a Subject (optional):	hello
Channel specific payload 1 Select Delivery Types:	SMS, EMAIL, IM, VOICE, TWO_WAY_PAGER
Content Type:	text/plain; charset=UT
Message Content:	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:

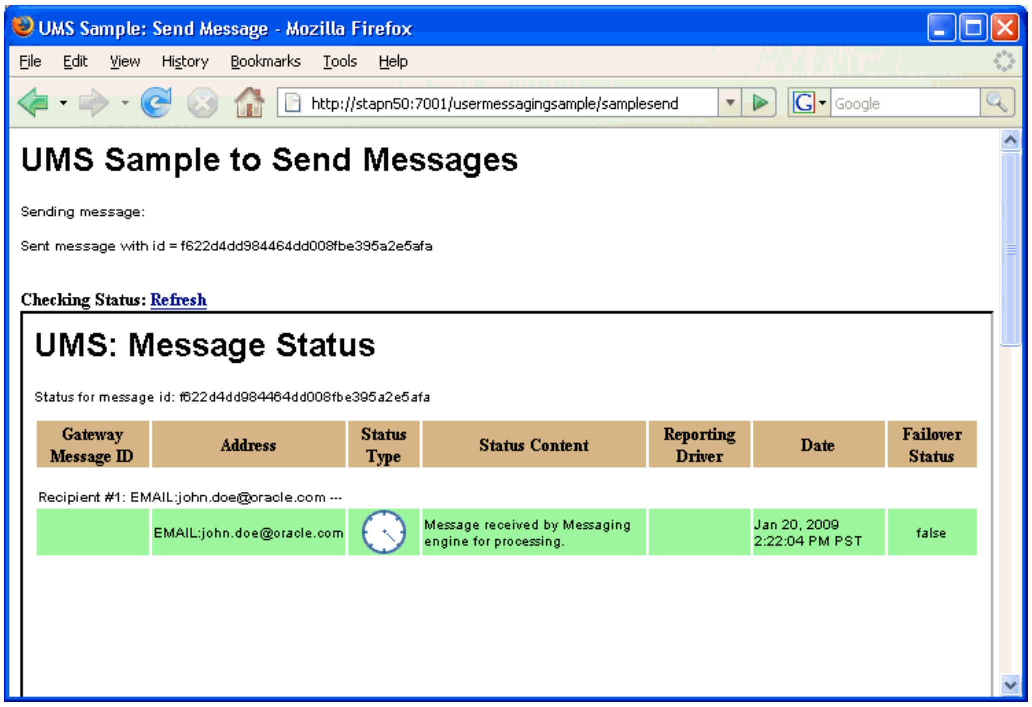
   E*mail:<recipient_address1>, Email:<recipient_address2>*.

   If you have configured user messaging preferences, you can address the message simply to "User:<username>". For example, User:weblogic.

5. As an optional step, enter a subject line or content for the e-mail.

6. Click **Send**. The *Message Status* page appears, showing the progress of transaction ("Message received by Messaging engine for processing," in Figure 17–7).

Figure 17–7 Message Status


UMS Sample: Send Message - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50:7001/usermessagingsample/samplesend
Google

UMS Sample to Send Messages

Sending message:

Sent message with id = f622d4dd984464dd008fbe395a2e5afa

Checking Status: Refresh

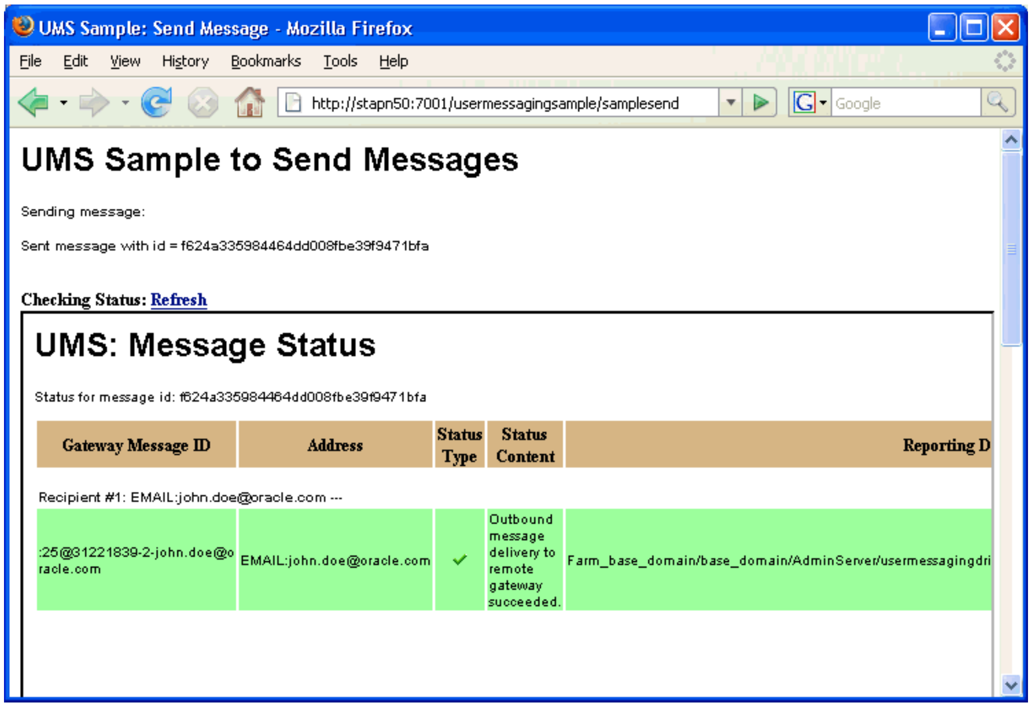
UMS: Message Status

Status for message id: f622d4dd984464dd008fbe395a2e5afa

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

7. Click **Refresh** to update the status. When the e-mail message has been delivered to the e-mail server, the *Status Content* field displays *Outbound message delivery to remote gateway succeeded.*, as illustrated in Figure 17–8.

Figure 17–8 Checking the Message Status


UMS Sample: Send Message - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50:7001/usermessagingsample/samplesend
Google

UMS Sample to Send Messages

Sending message:

Sent message with id = f624a335984464dd008fbe39f9471bfa

Checking Status: Refresh

UMS: Message Status

Status for message id: f624a335984464dd008fbe39f9471bfa

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/usermessagingdri

17.6 Using the UMS EJB 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 EJB Client API to asynchronously receive messages from an e-mail address and echo a reply back to the sender.

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.
  - *usermessagingclient-ejb.jar* -- Contains the Message EJB Client deployment descriptors.
    - * META-INF
    - *ejb-jar.xml*
    - *weblogic-ejb-jar.xml*
  - *usermessagingsample-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*
  - *usermessagingsample-echo-web.war* -- Contains the Web-based front-end and servlets.
    - * WEB-INF
    - *web.xml*
    - *weblogic.xml*

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

17.6.1 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 17.6.2, "Configuring the E-Mail Driver"
2. Section 17.6.3, "Using JDeveloper 11g to Build the Application"
3. Section 17.6.4, "Deploying the Application"
4. Section 17.6.5, "Testing the Application"

17.6.2 Configuring the E-Mail Driver

To enable the Oracle User Messaging Service's E-Mail Driver to perform inbound and outbound messaging and status retrieval, configure the E-Mail 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.

17.6.3 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:

17.6.3.1 Opening the Project

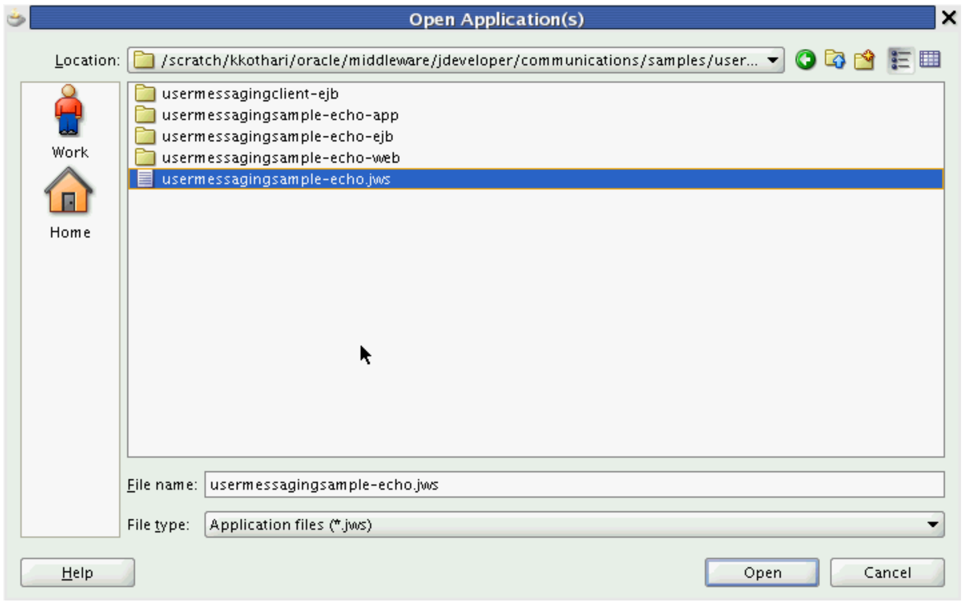
1. Unzip *usermessagingsample.echo-src.zip*, to the JDEV_HOME/communications/samples/ directory. This directory must be used for the shared library references to be valid in the project.

Note: If you choose to use a different directory, you must update the oracle.sdp.messaging library source path to JDEV_HOME/communications/modules/oracle.sdp.messaging_11.1.1/sdpmessaging.jar.

---

2. Open *usermessagingsample-echo.jws* (contained in the .zip file) in Oracle JDeveloper (Figure 17–9).

Figure 17–9 Opening the Project



Location: /scratch/kkothari/oracle/middleware/jdeveloper/communications/samples/user...

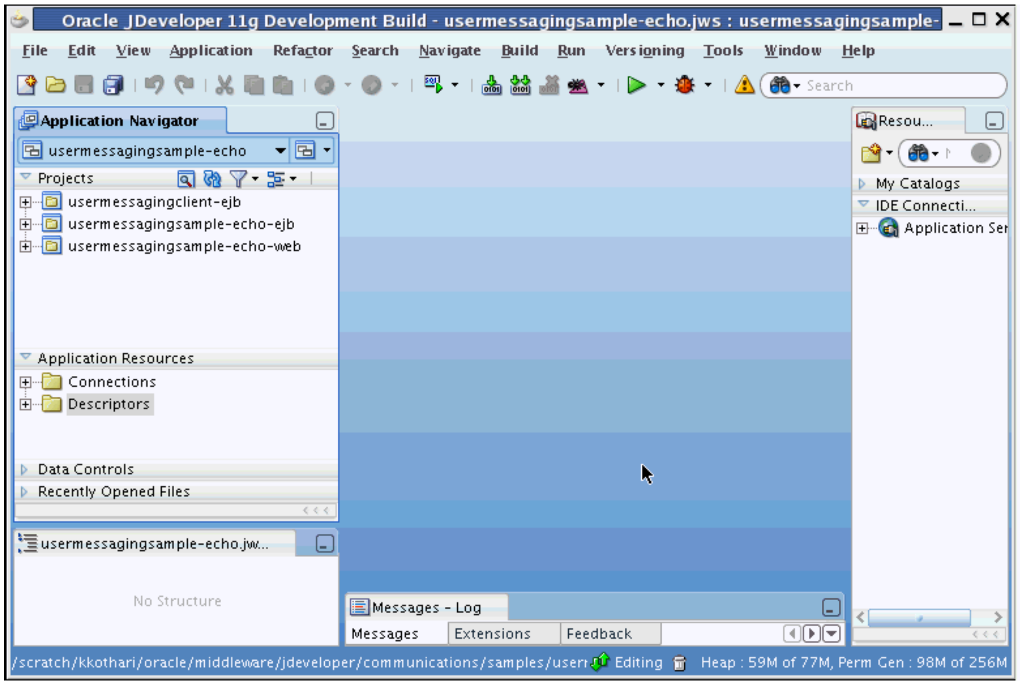
usermessagingclient-ejb
usermessagingsample-echo-app
usermessagingsample-echo-ejb
usermessagingsample-echo-web
usermessagingsample-echo.jws

File name: usermessagingsample-echo.jws

File type: Application files (*.jws)

In the Oracle JDeveloper main window the project appears (Figure 17–10).

Figure 17–10 Oracle JDeveloper Main Window


Oracle JDeveloper 11g Development Build - usermessagingsample-echo.jws : usermessagingsample-
File Edit View Application Refactor Search Navigate Build Run Versioning Tools Window Help
Search
Application Navigator
usermessagingsample-echo
Projects
usermessagingclient-ejb
usermessagingsample-echo-ejb
usermessagingsample-echo-web
Application Resources
Connections
Descriptors
Data Controls
Recently Opened Files
usermessagingsample-echo.jw...
No Structure
Messages - Log
Messages
Extensions
Feedback
Resou...
My Catalogs
IDE Connecti...
Application Ser
/scratch/kkothari/oracle/middleware/jdeveloper/communications/samples/user
Editing
Heap : 59M of 77M, Perm Gen : 98M of 256M

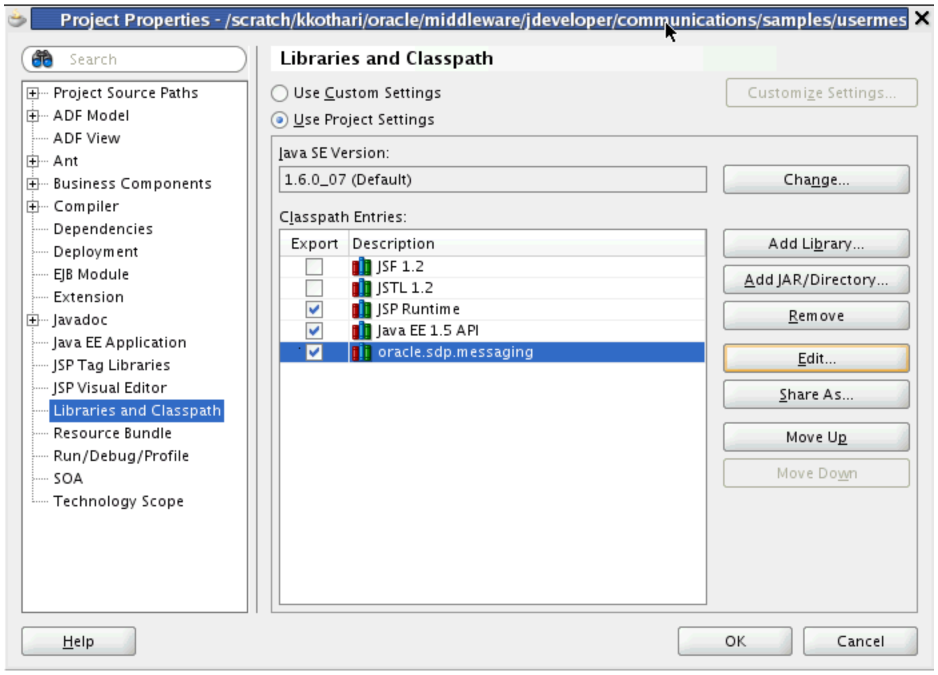
3. Verify that the build dependencies for the sample application have been satisfied by checking that the following library has been added to the *usermessagingsample-echo-web* and *usermessagingsample-echo-ejb* modules.

   - Library: *oracle.sdp.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 on the module and select **Project Properties**.
   2. In the left pane, select **Libraries and Classpath** (Figure 17–11).

Figure 17–11 Verifying Libraries



Project Properties - /scratch/kkothari/oracle/middleware/jdeveloper/communications/samples/usermes

Libraries and Classpath

Use Custom Settings [✓ Use Project Settings]

Java SE Version: 1.6.0_07 (Default)

Classpath Entries:

Export	Description
☐	JSF 1.2
☐	JSTL 1.2
✓	JSP Runtime
✓	Java EE 1.5 API
✓	oracle.sdp.messaging

3. Click **OK**.

4. Verify that the *usermessagingclient-ejb* project exists in the application. This is an EJB 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 *usermessagingsample-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 *usermessagingsample-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. (Example 17–11).

Example 17–11 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.messaging.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.messaging.sample.ejbApp.ClientReceiverHomeLocal</env-e
ntry-value>
            </env-entry>
```

8. Note that the ApplicationName ("UMSEchoApp") and ApplicationInstanceName ("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 17–12).

Example 17–12 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>
```

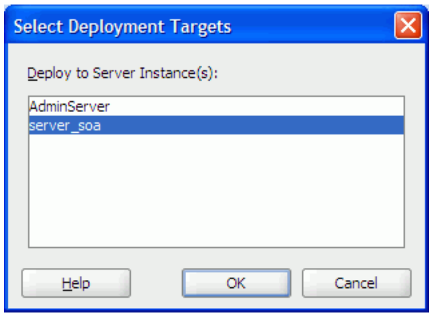
Note: If you chose a different ApplicationName and ApplicationInstanceName for your own application, remember to update this message selector. Asynchronous receiving does not work otherwise.

17.6.4 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 17.7, "Creating a New Application Server Connection."
2. Deploy the application by selecting the **usermessagingsample-echo application**, **Deploy**, **usermessagingsample-echo**, **to**, and **SOA_server** (Figure 17–12).

Figure 17–12 Deploying the Project



Deploy to Server Instance(s):

AdminServer
server_soa

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 Messaging Preferences.

Note: Refer to "Configuring Notifications" in the *Oracle Fusion Middleware SOA Developer's Guide* for more information.

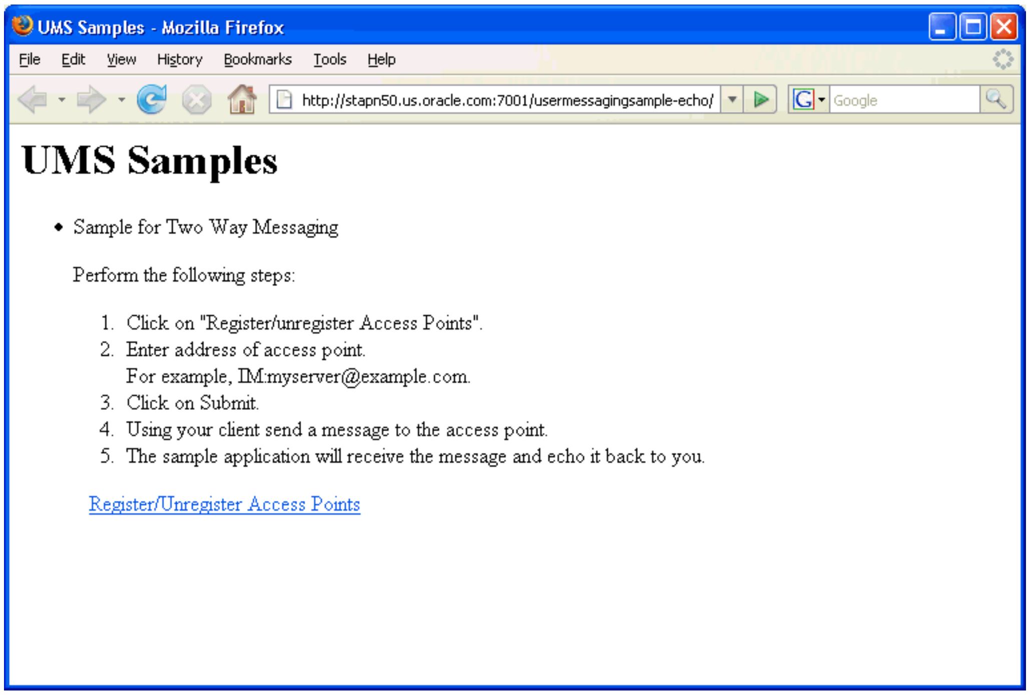
17.6.5 Testing the Application

Once *usermessagingsample-echo* has been deployed to a running instance of WLS, 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 17–13).

Figure 17–13 Testing the Sample Application


UMS Samples - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50.us.oracle.com:7001/usermessagingsample-echo/
Google

UMS Samples

- Sample for Two Way Messaging

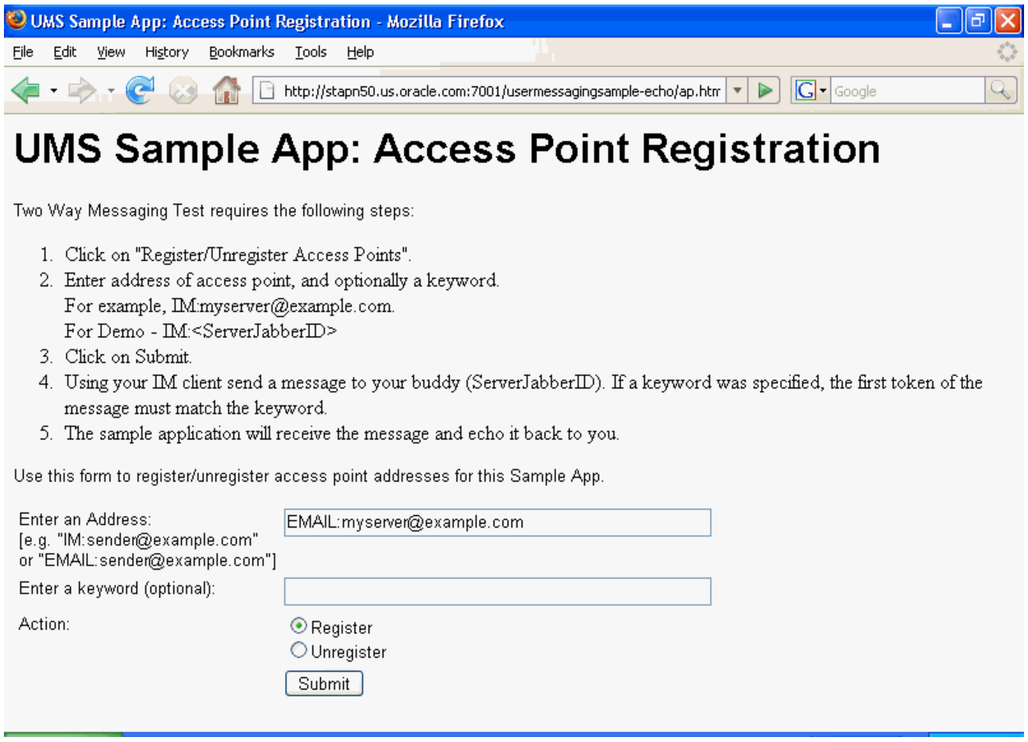
Perform the following steps:

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 17–14).

Figure 17–14 Registering an Access Point


UMS Sample App: Access Point Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50.us.oracle.com:7001/usermessagingsample-echo/ap.htm
Google

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"]	EMAIL:myserver@example.com
Enter a keyword (optional):	
Action:	[✓ Register] Unregister
	Submit

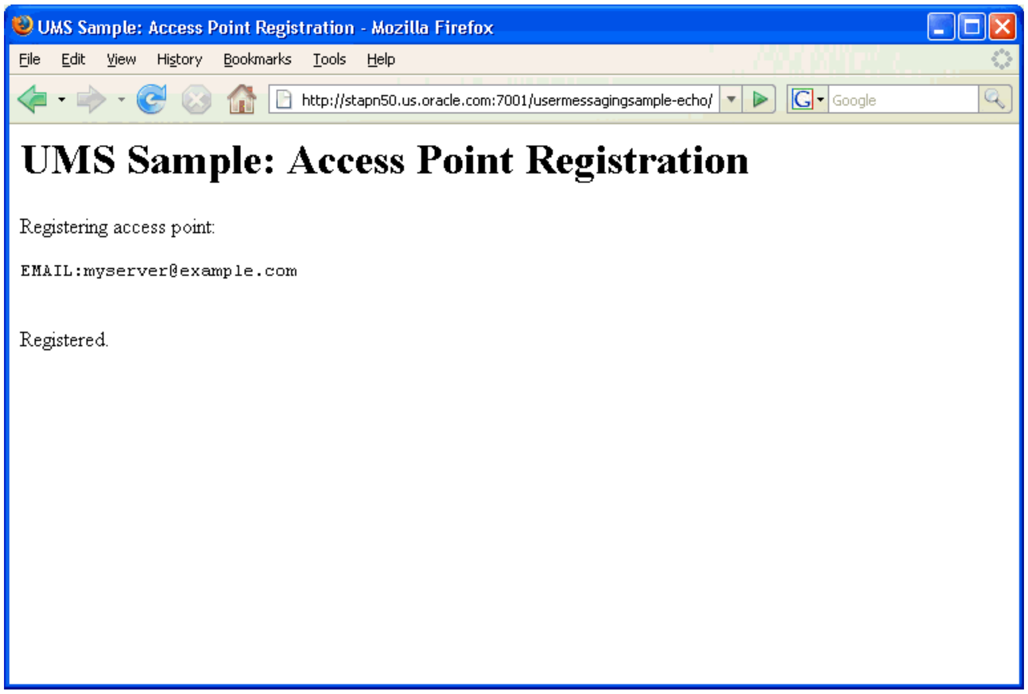
3. Enter the access point address in the following format:

   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 17–15).

Figure 17–15 Access Point Registration Status


UMS Sample: Access Point Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stapn50.us.oracle.com:7001/usermessagingsample-echo/
Google

UMS Sample: Access Point Registration

Registering access point:

```
EMAIL:myserver@example.com
```

Registered.

5. Send a message from your messaging client (for e-mail, your e-mail client) to the address you just registered as an access point in the previous step.

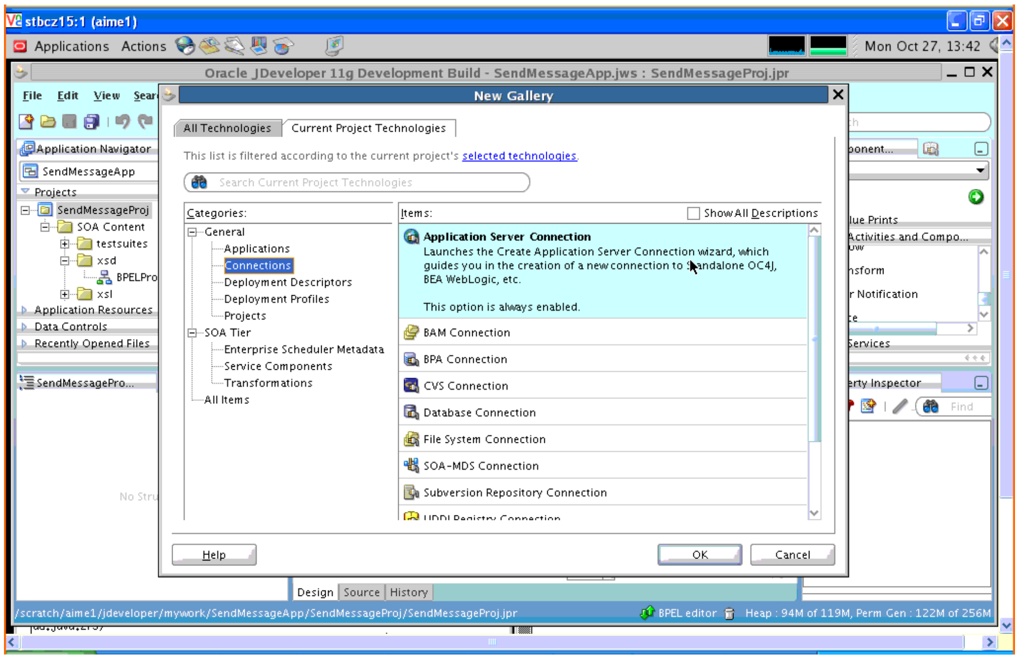
   If the UMS messaging driver for that channel is configured correctly, you can expect to receive an echo message back from the *usermessagingsample-echo* application.

17.7 Creating a New Application Server Connection

Perform the following steps to create a new Application Server Connection.

1. Create a new Application Server Connection by right-clicking the project and selecting **New**, **Connections**, and **Application Server Connection** (Figure 17–16).

Figure 17–16 New Application Server Connection



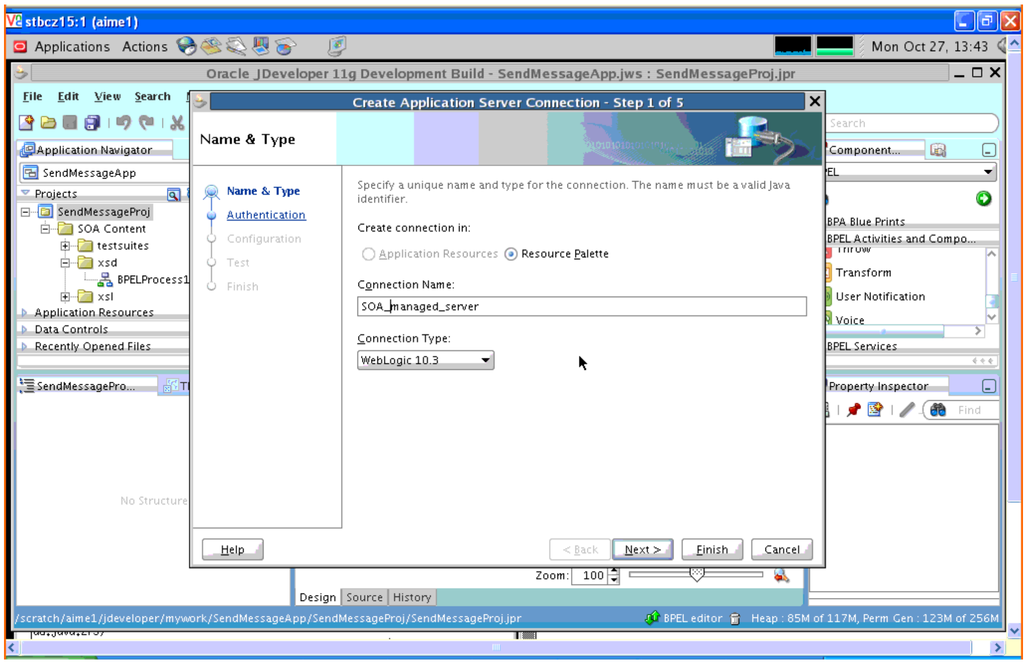
This list is filtered according to the current project's selected technologies.

Application Server Connection
Launches the Create Application Server Connection wizard, which guides you in the creation of a new connection to standalone OC4J, BEA WebLogic, etc.

This option is always enabled.

2. Name the connection “SOA_server” and click **Next** (Figure 17–17).
3. Select "WebLogic 10.3" as the *Connection Type*.

Figure 17–17 New Application Server Connection



Specify a unique name and type for the connection. The name must be a valid Java identifier.

Create connection in: ○ Application Resources [✓ Resource Palette]

Connection Name: SOA_managed_server

Connection Type: WebLogic 10.3

4. Enter the authentication information. The typical values are:

   Username: `weblogic`
   Password: `weblogic`

5. On the *Connection* screen, enter the hostname, port and SSL port for the SOA admin server, and enter the name of the domain for WLS Domain.

6. Click **Next**.
7. On the *Test* screen click **Test Connection**.
8. Verify that the message "Success!" appears.

   The Application Server Connection has been created.

18 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: To learn about the architecture and components of Oracle User Messaging Service, see *Oracle Fusion Middleware Getting Started with Oracle SOA Suite*.

This chapter contains the following sections:

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

18.1 Overview of 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.

18.2 Send Message Interface

The SendMessage interface allows 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 Oracle Fusion Middleware User Messaging Service API Reference 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 does not produce delivery receipts, even if a receiptRequest is specified. In other words, the receiptRequest parameter is ignored.

18.2.1 sendMessage Operation

Table 18–1 describes message descriptions for the `sendMessageRequest` input in the `sendMessage` operation.

Table 18–1 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 needs to handle this according to a SLA for the specific application and its use can therefore result in a PolicyException.
subject	xsd:string	Yes	Message subject. If mapped to SMS this parameter is used as the senderAddress, even if a separate senderAddress is provided.
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.

Table 18–1 (Cont.) sendMessage Input Message Descriptions

Part Name	Part Type	Optional	Description
charging	common: ChargingInformation	Yes	Charging to apply to this message.
receiptRequest	common:SimpleReference	Yes	Defines the application endpoint, interfaceName and correlator that is used to notify the application when the message has been delivered to a terminal or if delivery is impossible.

Table 18–2 describes `sendMessageResponse` output messages for the `sendMessage` operation.

Table 18–2 sendMessageResponse Output Message Descriptions

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

18.2.2 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 18–3 describes the `getMessageDeliveryStatusRequest` input messages for the `getMessageDeliveryStatus` operation.

Table 18–3 getMessageDeliveryStatusRequest Input Message Descriptions

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

Table 18–4 describes the getMessageDeliveryStatusResponse output messages for the `getMessageDeliveryStatus` operation.

Table 18–4 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.

18.3 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.

18.3.1 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 must 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 18–5 describes the `getReceivedMessagesRequest` input messages for the `getReceivedMessages` operation.

Table 18–5 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 get retrieved. If not specified, all messages shall be returned, that is, the same as specifying "Low."

Table 18–6 describes the `getReceivedMessagesResponse` output messages for the `getReceivedMessages` operation.

Table 18–6 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 get retrieved. If not specified, all messages shall be returned. This is the same as specifying Low.

18.3.2 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 18–7 describes the `getMessageRequest` input messages for the `getMessage` operation.

Table 18–7 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.

18.3.3 getMessageURIs Operation

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

- These URIs are HTTP URLs which can be dereferenced to retrieve the content.
- If the inbound message has a Content-Type of "multipart", then multiple URIs are returned, one per sub-part. If the Content-Type is not "multipart", then a single URI is 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, we 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 is no URI 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 is no URI 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 18–8 describes the `getMessageURIsRequest` input messages for the `getMessageURIs` operation.

Table 18–8 getMessageURIsRequest Input Message Descriptions

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

Table 18–9 describes the `getMessageURIsResponse` output messages for the `getMessageURIs` operation.

Table 18–9 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.

18.4 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.

18.4.1 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).

18.4.1.1 startReceiveMessage Operation

Invoking this operation allows a client to bind itself to a given endpoint for the purpose of 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 Oracle Fusion Middleware User Messaging Service API Reference 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 18–10 describes the `startReceiveMessageRequest` input messages for the `startReceiveMessage` operation.

Table 18–10 startReceiveMessageRequest Input Message Descriptions

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

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

18.4.1.2 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 in order 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 18–11 describes the `stopReceiveMessageRequest` input messages for the `stopReceiveMessage` operation.

Table 18–11 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.

18.5 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, we also provide pre-built Web Service stubs and interfaces 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, we recommend 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 Oracle Fusion Middleware User Messaging Service API Reference. 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 Oracle Fusion Middleware User Messaging Service API Reference 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(), etc). For examples of how to use this API, see the Messaging samples on Oracle Technology Network (OTN), and specifically usermessagingsample-parlayx-src.zip.

18.6 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.

Note: To learn about the architecture and components of Oracle User Messaging Service, see *Oracle Fusion Middleware Getting Started with Oracle SOA Suite*.

This chapter contains the following sections:

- Section 18.6.1, "Overview"
- Section 18.6.2, "Running the Pre-Built Sample"
- Section 18.6.3, "Testing the Sample"
- Section 18.6.4, "Creating a New Application Server Connection"

18.6.1 Overview

This sample demonstrates how to create a Web-based chat application to send and receive messages through e-mail, 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 must define an Application Server connection in 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.

18.6.1.1 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

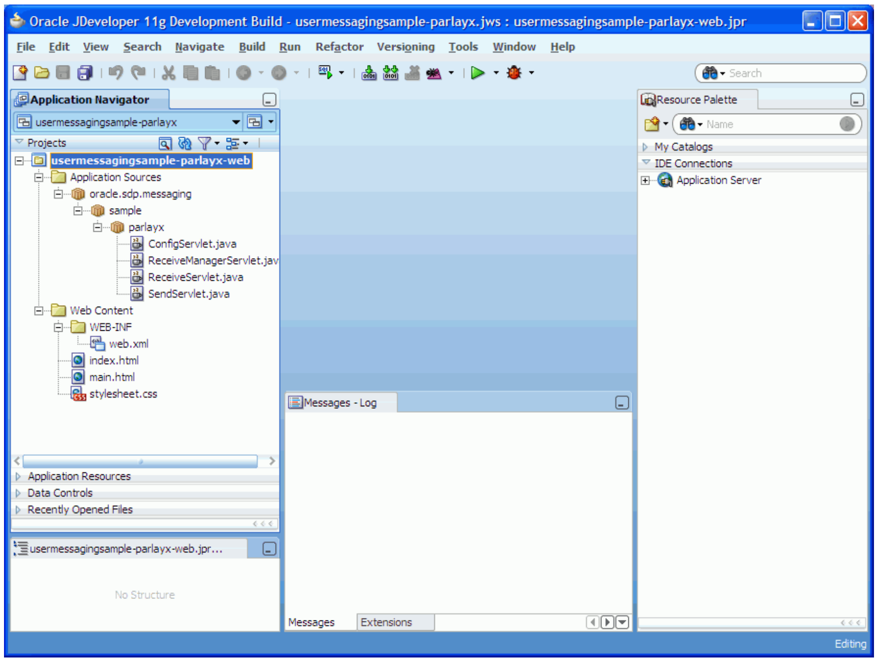
18.6.2 Running the Pre-Built Sample

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 18–1 Oracle JDeveloper Main Window



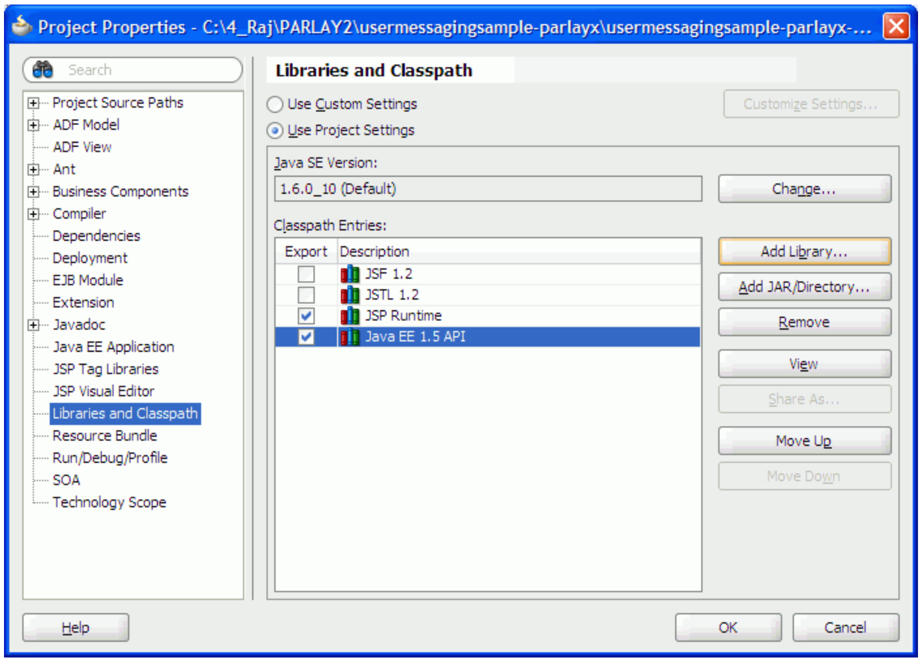
2. In Oracle JDeveloper, select **File** > **Open**..., then navigate to the directory above and open workspace file "usermessagingsample-parlayx.jws".

   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 adding a library to the Web module.

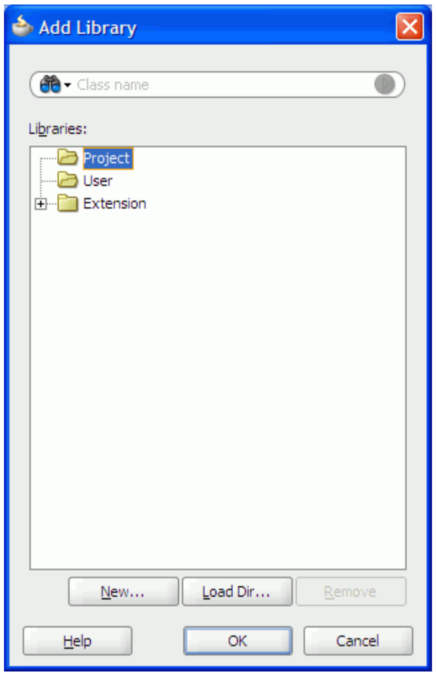
   1. In the Application Navigator, right click on web module usermessagingsample-parlayx-war, and select **Project Properties**.
   2. In the left pane, select **Libraries and Classpath**.

Figure 18–2 Adding a Library



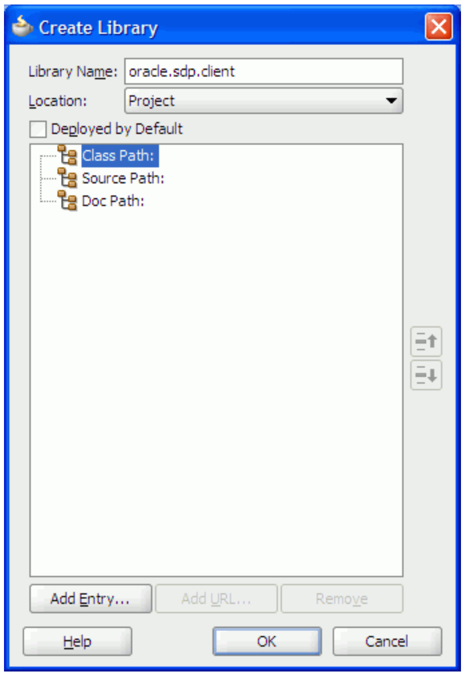
3. Click **Add Library**.

Figure 18–3 Adding a Library



4. Click **New** to define a new library.
5. For Library Name, enter "oracle.sdp.client".

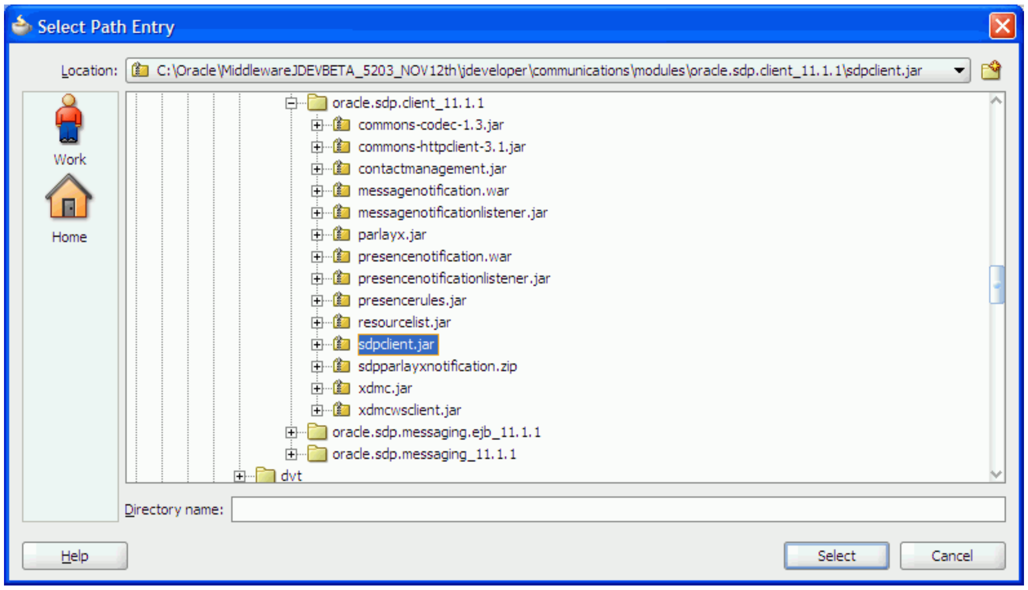
Figure 18–4 Defining the Library



6. With "Class Path" selected, select **Add Entry**.

7. Navigate to <JDeveloper_Base_Directory>/communications/modules/oracle.sdp.client_11.1.1, and select jar file "sdpclient.jar".

Figure 18–5 Selecting sdpclient.jar

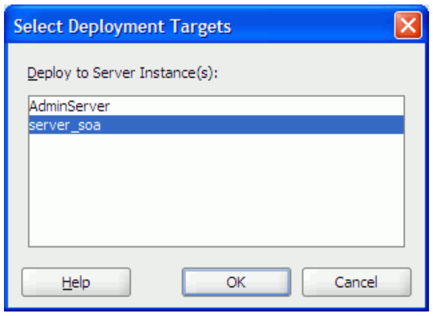

Select Path Entry
Location: C:\Oracle\MiddlewareJDEVBETA_5203_NOV12th\jdeveloper\communications\modules\oracle.sdp.client_11.1.1\sdpclient.jar
Work
Home
oracle.sdp.client_11.1.1
commons-codec-1.3.jar
commons-httpclient-3.1.jar
contactmanagement.jar
messagenotification.war
messagenotificationlistener.jar
parlayx.jar
presencenotification.war
presencenotificationlistener.jar
presencerules.jar
resourcelist.jar
sdpclient.jar
sdpparlayxnotification.zip
xdmc.jar
xdmcwsclient.jar
oracle.sdp.messaging.ejb_11.1.1
oracle.sdp.messaging_11.1.1
dvt
Directory name:
Help
Select
Cancel

8. Click **OK**/**Accept** in all popups to create the library and add it as a dependency to the sample Web module.

4. Create an Application Server Connection by right-clicking the project in the navigation pane and selecting New. Follow the instructions in Section 18.6.4, "Creating a New Application Server Connection".

5. Deploy the project by selecting the **usermessasgingsample-parlayx project**, **Deploy**, **usermessasgingsample-parlayx**, **to**, and **SOA_server** (Figure 18–6).

Figure 18–6 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 Messaging Preferences, as described in the following sections.

Note: Refer to "Configuring Notifications" in the *Oracle Fusion Middleware SOA Developer's Guide* for more information.

18.6.3 Testing the Sample

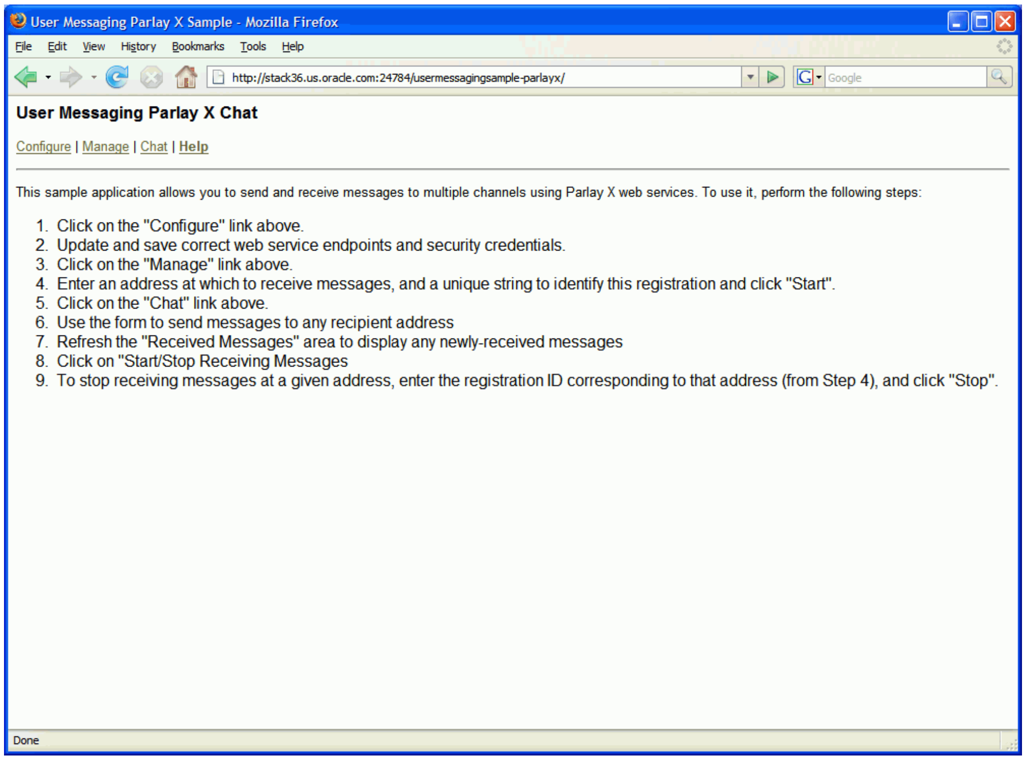
Perform the following steps to run and test the sample:

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 18–7). This page contains navigation tabs and instructions for the application.

Figure 18–7 Messaging Parlay X Sample Web Page


User Messaging Parlay X Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stack36.us.oracle.com:24784/usermessagingsample-parlayx/
Google

User Messaging Parlay X Chat

Configure | Manage | Chat | Help

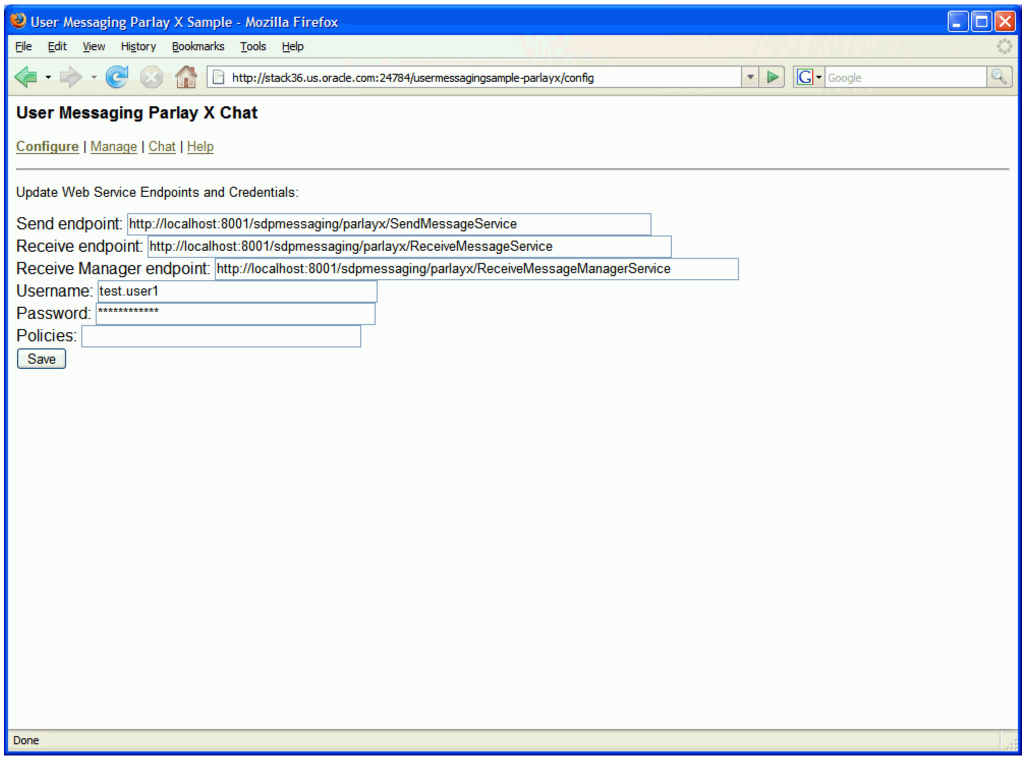
This sample application allows you to send and receive messages to multiple channels using Parlay X web services. To use it, perform the following steps:

1. Click on the "Configure" link above.
2. Update and save correct web service endpoints and security credentials.
3. Click on the "Manage" link above.
4. Enter an address at which to receive messages, and a unique string to identify this registration and click "Start".
5. Click on the "Chat" link above.
6. Use the form to send messages to any recipient address
7. Refresh the "Received Messages" area to display any newly-received messages
8. Click on "Start/Stop Receiving Messages
9. To stop receiving messages at a given address, enter the registration ID corresponding to that address (from Step 4), and click "Stop".

3. Click **Configure** and enter the following values (Figure 18–8):

   - 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/ReceiveMessageMessageServic e
   - Specify the Username and Password.
   - Specify a Policy (required if the User Messaging Service instance has WS security enabled).

Figure 18–8 Configuring the Web Service Endpoints and Credentials


User Messaging Parlay X Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stack36.us.oracle.com:24784/usermessagingsample-parlayx/config
Google
Configure | Manage | Chat | Help
Save
Done

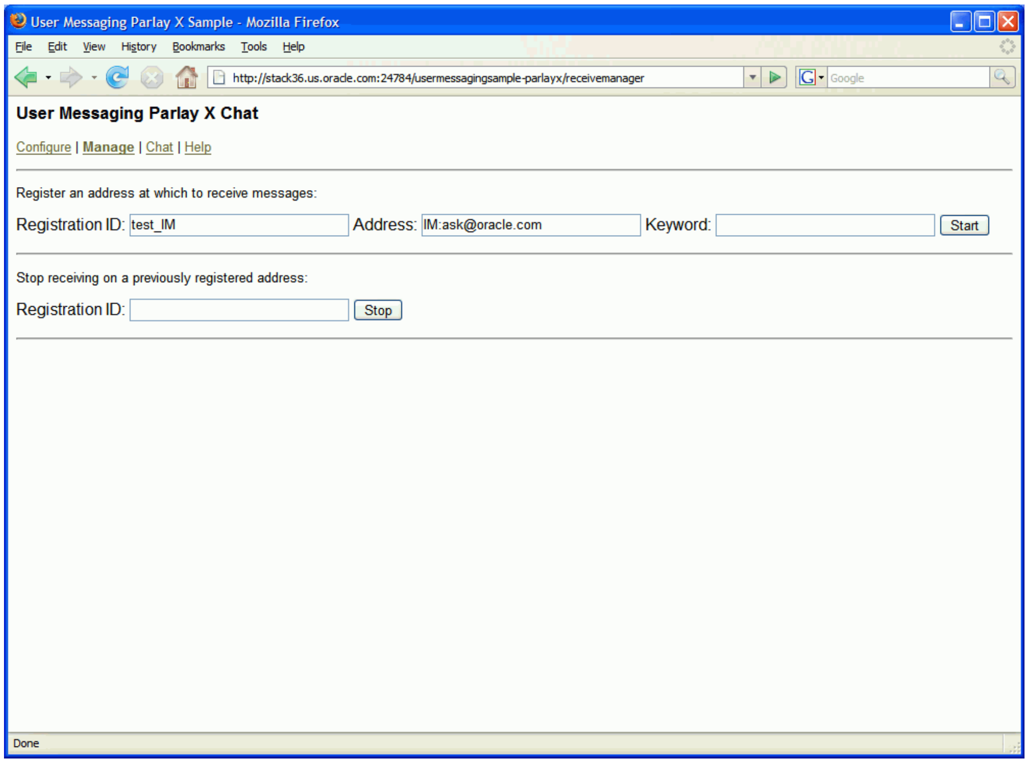
User Messaging Parlay X Chat

Update Web Service Endpoints and Credentials:

Send endpoint: http://localhost:8001/sdpmessaging/parlayx/SendMessageService
Receive endpoint: http://localhost:8001/sdpmessaging/parlayx/ReceiveMessageService
Receive Manager endpoint: http://localhost:8001/sdpmessaging/parlayx/ReceiveMessageManagerService
Username: test.user1
Password: ************
Policies:

4. Click **Save**.
5. Click **Manage**.
6. Enter a Registration ID to specify the registration and address at which to receive messages (Figure 18–9). You can also use this page to stop receiving messages at an address.

Figure 18–9 Specifying a Registration ID



User Messaging Parlay X Sample - Mozilla Firefox

http://stack36.us.oracle.com:24784/usermessagingsample-parlayx/receivemanager

User Messaging Parlay X Chat

Configure | **Manage** | Chat | Help

Register an address at which to receive messages:

Registration ID: test_IM Address: IM:ask@oracle.com Keyword: Start

Stop receiving on a previously registered address:

Registration ID: Stop

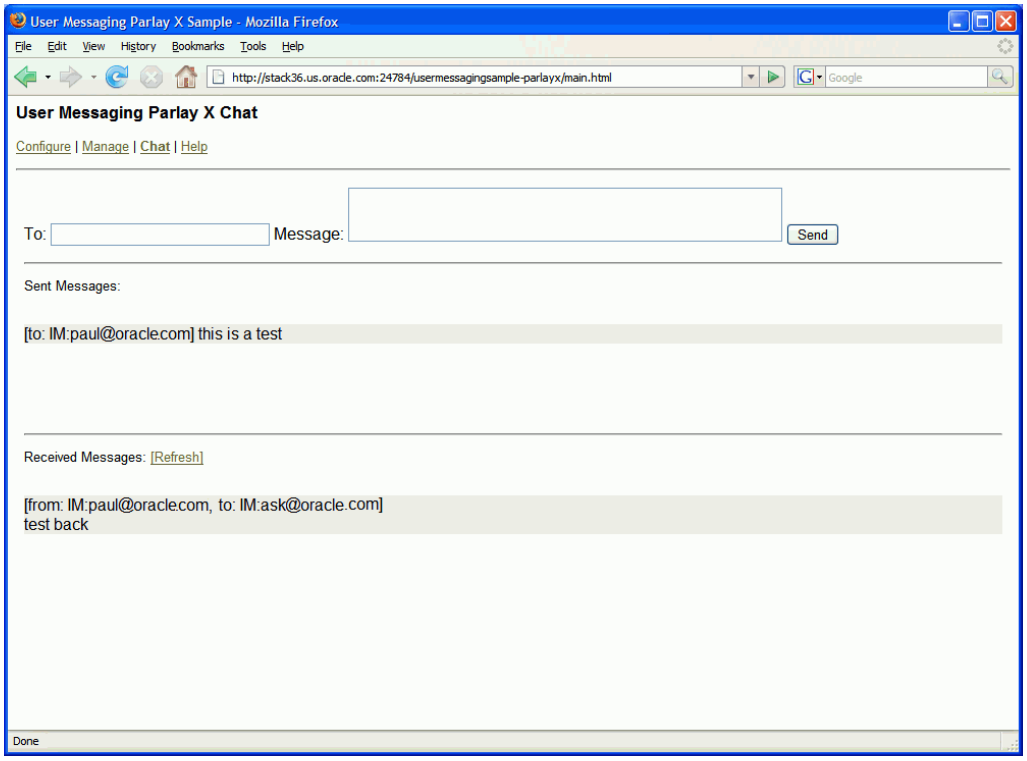
Done

7. Click **Start**.

   Verify that the message "Registration operation succeeded" appears.

8. Click **Chat** (Figure 18–10).
9. Enter recipients in the To: field in the format illustrated in Figure 18–10.
10. Enter a message.
11. Click **Send**.
12. Verify that the message is received.

Figure 18–10 Running the Sample


User Messaging Parlay X Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://stack36.us.oracle.com:24784/usermessagingsample-parlayx/main.html
Google
Configure | Manage | Chat | Help
To:
Message:
Send
Done

User Messaging Parlay X Chat

Sent Messages:

[to: IM:paul@oracle.com] this is a test

Received Messages: [Refresh]

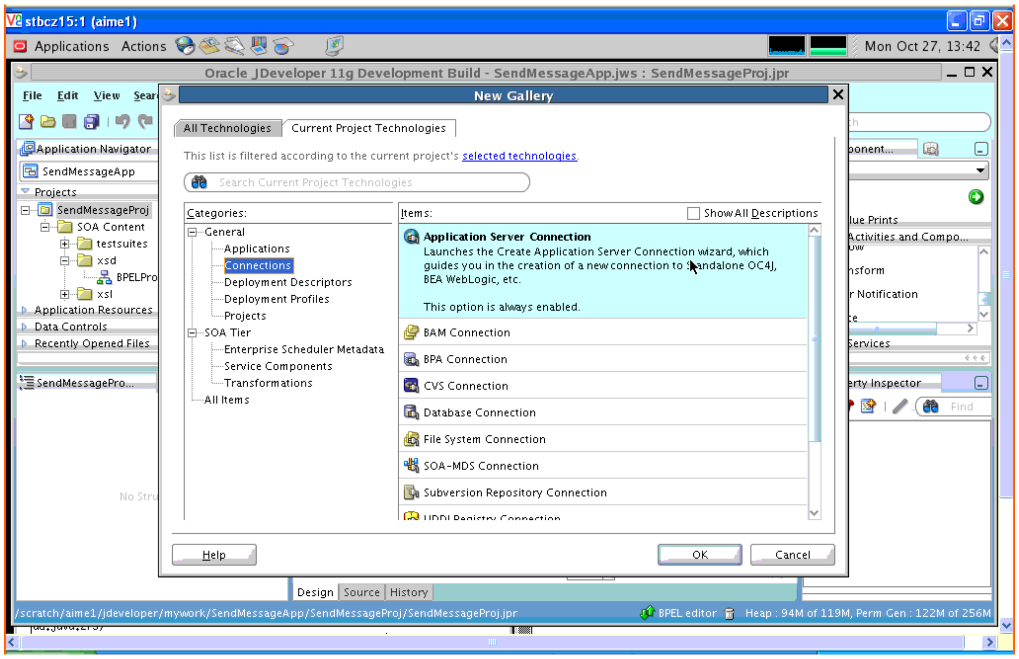
[from: IM:paul@oracle.com, to: IM:ask@oracle.com]
test back

18.6.4 Creating a New Application Server Connection

Perform the following steps to create a new Application Server Connection.

1. Create a new Application Server Connection by right-clicking the project and selecting **New**, **Connections**, and **Application Server Connection** (Figure 18–11).

Figure 18–11 New Application Server Connection



This list is filtered according to the current project's selected technologies.

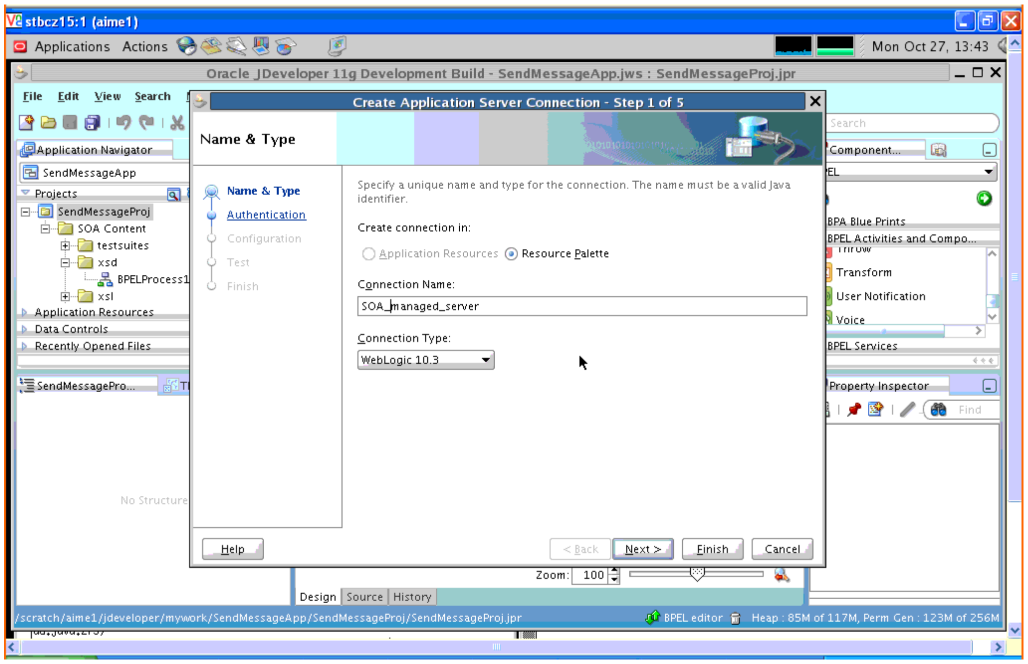
Launches the Create Application Server Connection wizard, which guides you in the creation of a new connection to standalone OC4J, BEA WebLogic, etc.

This option is always enabled.

/scratch/aime1/jdeveloper/mywork/SendMessageApp/SendMessageProj/SendMessageProj.jpr

2. Name the connection “SOA_server” and click **Next** (Figure 18–12).
3. Select "WebLogic 10.3" as the *Connection Type*.

Figure 18–12 New Application Server Connection



Create Application Server Connection - Step 1 of 5

Name & Type

Specify a unique name and type for the connection. The name must be a valid Java identifier.

Create connection in:

Application Resources [✓ Resource Palette]

Connection Name: SOA_managed_server

Connection Type: WebLogic 10.3

4. Enter the authentication information. The typical values are:

   Username: `weblogic`
   Password: `weblogic`

5. On the *Connection* screen, enter the hostname, port and SSL port for the SOA admin server, and enter the name of the domain for WLS Domain.
6. Click **Next**.
7. On the *Test* screen click **Test Connection**.
8. Verify that the message “Success!” appears.

   The Application Server Connection has been created.

19 User Messaging Preferences

This chapter describes the User Messaging Preferences that are packaged with Oracle User Messaging Service. It describes how to work with messaging channels and to create contact rules using messaging filters.

Note: To learn about the architecture and components of Oracle User Messaging Service, see *Oracle Fusion Middleware Getting Started with Oracle SOA Suite*.

This chapter contains the following sections:

- Section 19.1, "Introduction"
- Section 19.2, "How to Manage Messaging Channels"
- Section 19.3, "Creating Contact Rules using Filters"
- Section 19.4, "Configuring Settings"

19.1 Introduction

User Messaging Preferences allows a user who has access to multiple channels (delivery types) to control how, when, and where they receive messages. Users define filters, or delivery preferences, that specify which channel a message must be delivered to, and under what circumstances. Information about a user's devices and filters are stored in any database supported for use with Oracle Fusion Middleware.

For an application developer, User Messaging Preferences provide increased flexibility. Rather than an application needing business logic to decide whether to send an email or SMS message, the application can just send to the user, and the message gets delivered according to the user's preferences.

Since preferences are stored in a database, this information is shared across all instances of User Messaging Preferences in a domain.

The `oracle.sdp.messaging.userprefs` package contains the User Messaging Preferences API classes. For more information, refer to the Oracle Fusion Middleware User Messaging Service API Reference.

19.1.1 Terminology

User Messaging Preferences defines the following terminology:

- Channel: a physical channel, such as a phone, or PDA.
- Channel address: one of the addresses that a channel can communicate with.

- Filters: a set of notification delivery preferences.
- System term: a pre-defined business term that cannot be extended by the administrator.
- Business term: a rule term defined and managed by the system administrator through Enterprise Manager. Business terms can be added, defined, or deleted.
- Rule term: a system term or a business term.
- Operators: comparison operators *equals*, *does not equal*, *contains*, or *does not contain*.
- Facts: data passed in from the message to be evaluated, such as *time sent*, or *sender*.
- Rules Engine: the User Messaging Preferences component that processes and evaluates filters.
- Channel: the transport type, for example, e-mail, voice, or SMS.
- Comparison: a rule term and the associated comparison operator.
- Action: the action to be taken if the specified conditions in a rule are true, such as *Broadcast to All*, *Failover*, or *Do not Send to Any Channel*.

19.1.2 Configuration of Notification Delivery Preferences

User Messaging Preferences allows configuration of notification delivery preferences based on the following:

- a set of well-defined rule terms (system terms or business terms)
- a set of channel and the corresponding addresses supported by Oracle User Messaging Service
- a set of User Messaging Preferences filters that are transparently handled by a rules engine

One use case for notification delivery preference is for bugs entered into a bug tracking system. For example, user *Alex* wants to be notified through SMS and EMAIL channels for bugs filed against his product with priority = 1 by a customer type = Premium. For all other bugs with priority > 1, he only wants to be notified by EMAIL. Alex's preferences can be stated as follows:

Example 19–1 Notification Delivery Preferences

```
Rule (1): if (Customer Type = Premium) AND (priority = 1) then notify [Alex] using
 SMS and EMAIL.

Rule (2): if (Customer Type = Premium) AND (priority > 1) then notify [Alex] using
EMAIL.
```

A runtime service, the Oracle Rules Engine, evaluates the filters to process the notification delivery of user requests.

19.1.3 Delivery Preference Rules

A delivery preference rule consists of *rule comparisons* and *rule actions*. A rule comparison consists of a *rule term* (a system term or a business term) and the associated comparison operators. A rule action is the action to be taken if the specified conditions in a rule are true.

19.1.3.1 Data Types

Table 19–2 lists data types supported by User Messaging Preferences. Each system term and business term must have an associated data type, and each data type has a set of pre-defined comparison operators. Administrators cannot extend these operators.

Table 19–1 Data Types Supported by User Messaging Preferences

Data Type	Comparison Operators	Supported Values
Date	<, >, between, <=, >=	Date is accepted as a "java.util.Date" object or "String" representing the number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT (in essence, the value from java.util.Date.getTime() or java.util.Calendar.getTime()).
Time	==, !=, between	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.
Number (Decimal)	<, >, between, <=, >=	A java.lang.Double object or a String representing a floating decimal point number with double precision.
String	==, !=, contains, not contains	Any arbitrary string.

Note: The String data type does not support regular expressions.

The Time data type is only available to System Terms.

19.1.3.2 System Terms

Table 19–2 lists system terms, which are pre-defined business terms. Administrators cannot extend the system terms.

Table 19–2 System Terms Supported by User Messaging Preferences

System Term	Data Type	Supported Values
Date	Date	Date is accepted as a "java.util.Date" object or "String" representing the number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT (in essence, the value from java.util.Date.getTime() or java.util.Calendar.getTime()).
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.

19.1.3.3 Business Terms

Business terms are rule terms defined and managed by the system administrator through Oracle Application Server 11g Enterprise Manager. For more information on adding, defining, and deleting business terms, refer to *Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite*. A business term consists of a key, a data type, an optional description, and an optional List of Values (LOV).

Table 19–3 lists the pre-defined business terms supported by User Messaging Preferences.

Table 19–3 Pre-defined Business Terms for User Messaging Preferences

Business Term	Data Type
Organization	String
Time	Number (Decimal)
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

19.1.4 Rule Actions

For a given rule, a User Messaging Preferences user can define one of the following actions:

- **Broadcast to All**: send a broadcast message to all channels in the broadcast address list.
- **Failover**: Send a message serially to channels in the address list until one successful message is sent. This means performing a send to the next channel when the current channel returns a failure status. User Messaging Preferences does not allow a user to specify a channel-specific status code or expiration time.
- **Do not send to Any Channel**: Do not send a message to any channel.

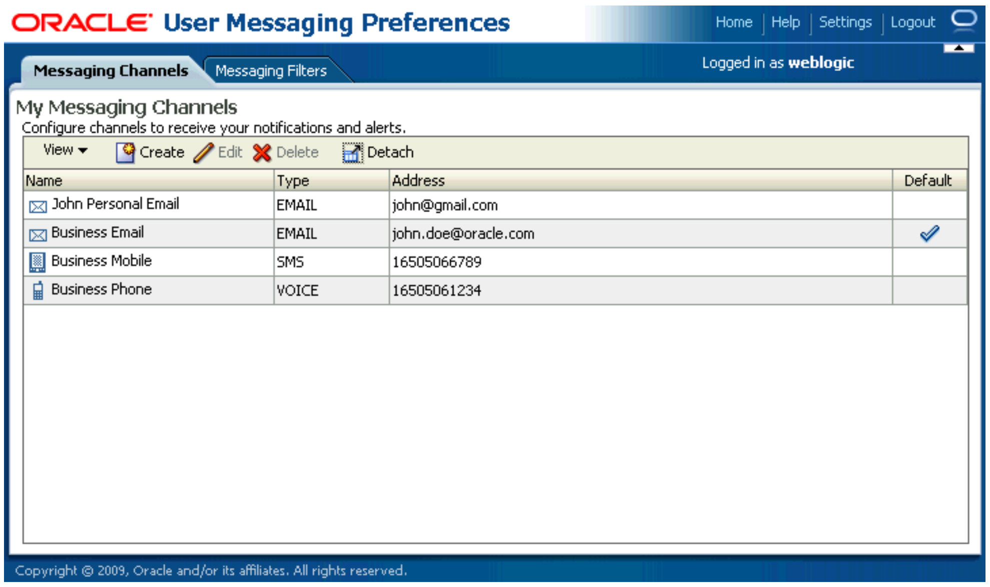
Tip: User Messaging Preferences does not provide a filter action that instructs "do not send to a specified channel." A best practice is to specify only positive actions, and not negative actions in rules.

- **Default address**: if no action is defined, a message is sent to a default address, as defined in the Messaging Channels page in Enterprise Manager.

19.2 How to Manage Messaging Channels

Any channel that a user creates is associated with that user's system ID. In Oracle User Messaging Service, channels represent both physical channels, such as mobile phones, and also e-mail client applications running on desktops, and are configurable on the The *Messaging Channels* tab (Figure 19–1).

Figure 19–1 Messaging Channels Tab



ORACLE User Messaging Preferences

Home | Help | Settings | Logout

Messaging Channels | Messaging Filters

Logged in as **weblogic**

My Messaging Channels

Configure channels to receive your notifications and alerts.

View ▾ | Create | Edit | Delete | Detach

Name	Type	Address	Default
John Personal Email	EMAIL	john@gmail.com	
Business Email	EMAIL	john.doe@oracle.com	✓
Business Mobile	SMS	16505066789	
Business Phone	VOICE	16505061234	

Copyright © 2009, Oracle and/or its affiliates. All rights reserved.

The *Messaging Channels* tab enables users to perform the following tasks:

19.2.1 Creating a Channel

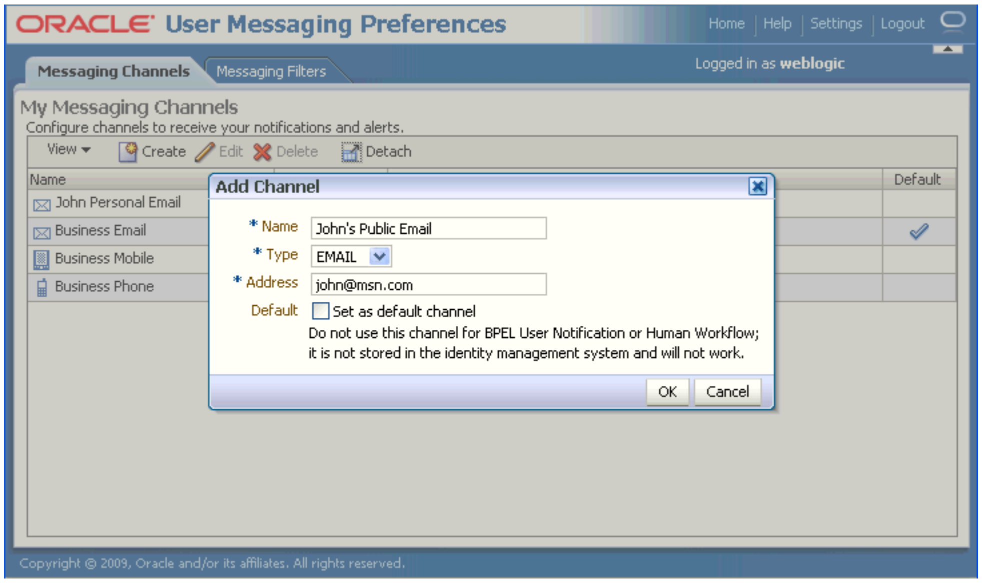
To create a channel:

1. Click **Create** (Figure 19–2).

Figure 19–2 The Create Icon

2. Enter a name for the channel in the *Name* field (Figure 19–3).
3. Select the channel's transport type from the *Type* drop-down menu.
4. Enter the number or address appropriate to the transport type you selected.
5. Select the *Default* checkbox to set the channel as the default channel.

Figure 19–3 Creating a Channel



* Name	John's Public Email
* Type	EMAIL
* Address	john@msn.com
Default	☐ Set as default channel

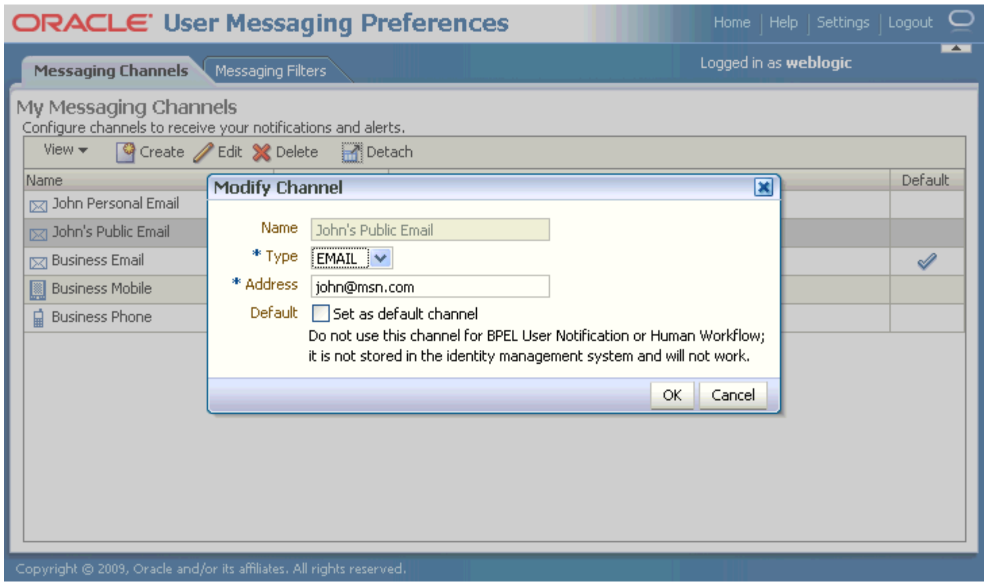
Do not use this channel for BPEL User Notification or Human Workflow; it is not stored in the identity management system and will not work.

6. Click **OK** to create the channel. The channel appears on the *Channels* page. The *Channels* page enables you to edit or delete the channel.

19.2.2 Editing a Channel

To edit a channel, select it and click **Edit** (Figure 19–4). The editing page appears for the channel, which enables you to add or change the channel properties described in Section 19.2.1, "Creating a Channel".

Figure 19–4 Edit a Channel

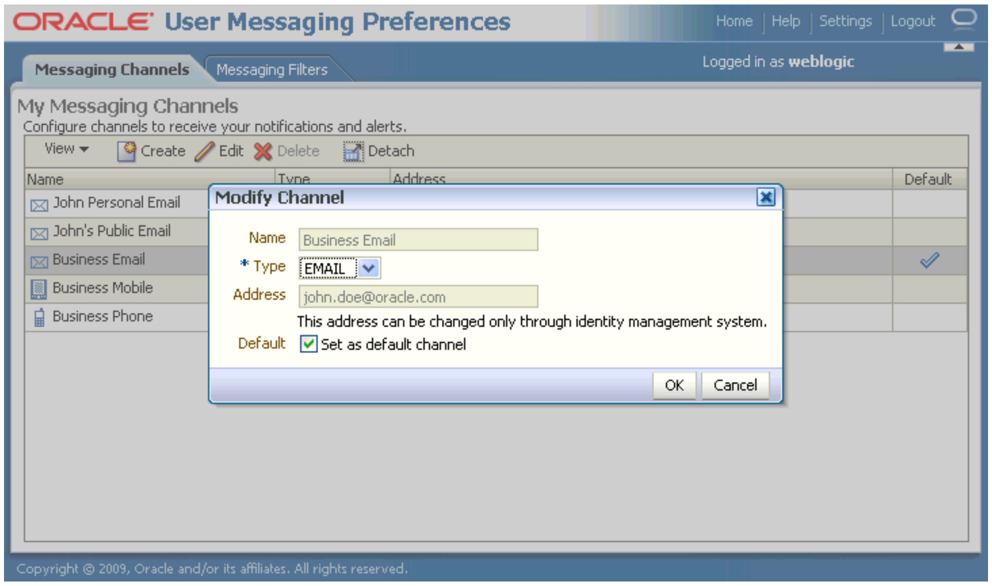


Name	John's Public Email
* Type	EMAIL
* Address	john@msn.com
Default	☐ Set as default channel

Do not use this channel for BPEL User Notification or Human Workflow; it is not stored in the identity management system and will not work.

Certain channels are based on information retrieved from your user profile in the identity store, and this address cannot be modified by User Messaging Preferences (Figure 19–5). The only operation that can be performed on such as channel is to make it the default.

Figure 19–5 Edit a Identity Store-Backed Channel


ORACLE User Messaging Preferences
Home | Help | Settings | Logout
Logged in as **weblogic**
Messaging Channels
Messaging Filters
View
Create
Edit
Delete
Detach

My Messaging Channels
Configure channels to receive your notifications and alerts.

Name	Type	Address	Default
John Personal Email			
John's Public Email			
Business Email			✓
Business Mobile			
Business Phone			

Modify Channel

Name: Business Email
* Type: EMAIL
Address: john.doe@oracle.com
This address can be changed only through identity management system.
Default: [✓] Set as default channel

OK Cancel

Copyright © 2009, Oracle and/or its affiliates. All rights reserved.

19.2.3 Deleting a Channel

To delete a channel, select it and click **Delete** (Figure 19–6).

Figure 19–6 The Delete Icon

19.2.4 Setting a Default Channel

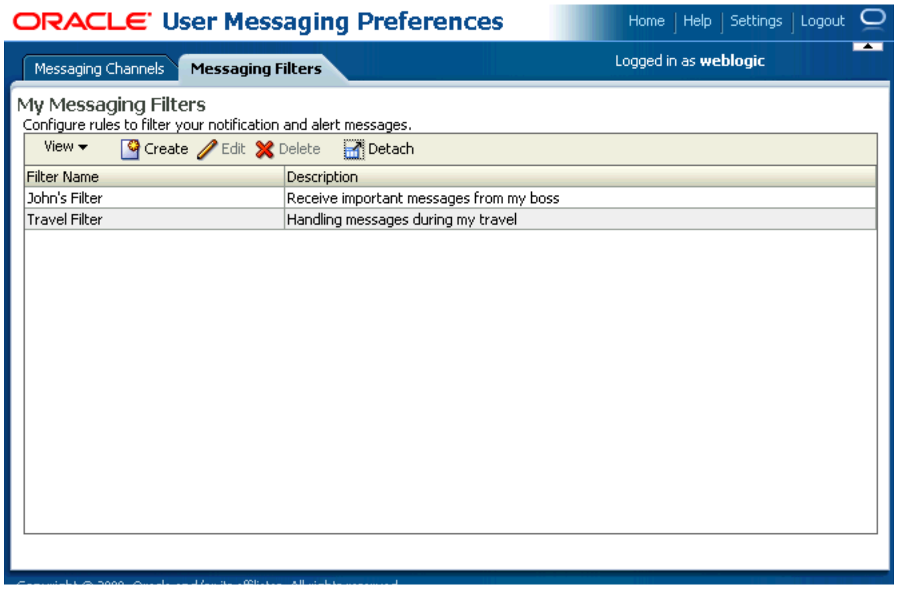
E-Mail is the default for receiving notifications. To set another channel as the default, select it, click **Edit**, and then click "Set as default channel." A check mark (Figure 19–7) appears next to the selected channel, designating it as the default means of receiving notifications.

Figure 19–7 The Default Icon

19.3 Creating Contact Rules using Filters

The *Messaging Filters* tab (Figure 19–8) enables users to build filters that specify not only the type of notifications they wish to receive, but also the channel through which to receive these notifications through a combination of comparison operators (such as *is equal to*, *is not equal to*), business terms that describe the notification type, content or source, and finally, the notification actions, which send the notifications to all channels, block channels from receiving notifications, or send notifications to the first available channel.

Figure 19–8 Messaging Filters Tab



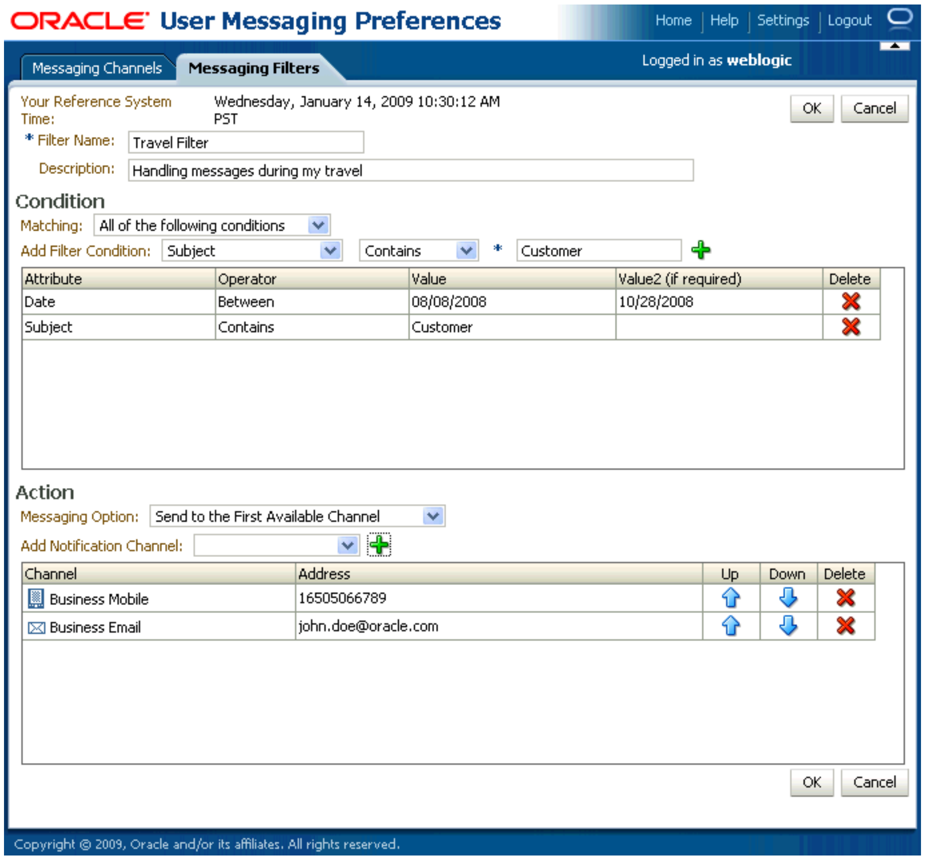
My Messaging Filters

Configure rules to filter your notification and alert messages.

Filter Name	Description
John's Filter	Receive important messages from my boss
Travel Filter	Handling messages during my travel

Figure 19–9 illustrates the creation of a filter called "Travel Filter", by a user named "weblogic", for handling notifications regarding Customers during his travel. Notifications that match all of the filter conditions are first directed to his "Business Mobile" channel. If this channel becomes unavailable, Oracle User Messaging Service transmits the notifications as e-mails since the next available channel selected is "Business Email".

Figure 19–9 Creating a Filter


ORACLE User Messaging Preferences
Home | Help | Settings | Logout
Logged in as weblogic
Messaging Channels
Messaging Filters
OK
Cancel

Your Reference System Time: Wednesday, January 14, 2009 10:30:12 AM PST

* Filter Name: Travel Filter

Description: Handling messages during my travel

Condition

Matching: All of the following conditions

Add Filter Condition: Subject | Contains | * Customer

Attribute	Operator	Value	Value2 (if required)	Delete
Date	Between	08/08/2008	10/28/2008	✗
Subject	Contains	Customer		✗

Action

Messaging Option: Send to the First Available Channel

Add Notification Channel:

Channel	Address	Up	Down	Delete
Business Mobile	16505066789	⇧	⇩	✗
Business Email	john.doe@oracle.com	⇧	⇩	✗

19.3.1 Creating Filters

To create a filter:

1. Click **Create** (Figure 19–2). The *Create Filter* page appears (Figure 19–9).
2. Enter a name for the filter in the *Filter Name* field.
3. If needed, enter a description of the filter in the *Description* field.
4. Define the filter conditions using the lists and fields of the *Condition* section as follows:
   - **a.** Select whether notifications must meet all of the conditions or any of the conditions by selecting either the **All of the following conditions** or the **Any of the following conditions** options.
   - **b.** Select the notification's attributes. These attributes, or business components, include
     - Organization
     - Time
     - Priority
     - Application
     - Application Type
     - Expiration Date

- From
- To
- Customer Name
- Customer Type
- Status
- Amount
- Due Date
- Process Type
- Expense Type
- Total Cost
- Processing Time
- Order Type
- Service Request Type
- Group Name
- Source
- Classification
- Duration
- User
- Role

5. Combine the selected condition type with one of the following comparison operators:

- Is Equal To
- Is Not Equal To
- Contains
- Does Not Contain

If you select the *Date* attribute, select one of the following comparison operators and then select the appropriate dates from the calendar application.

- Is Equal
- Is Not Equal
- Is Greater Than
- Is Greater Than or Equal
- Is Less Than
- Is Less Than or Equal
- Between
- Is Weekday
- Is Weekend

6. Add appropriate values describing the attributes or operators.

7. Click **Add** (Figure 19–6) to add the attribute and the comparison operators to the table.
8. Repeat these steps to add more filter conditions. To delete a filter condition, click **Delete** (Figure 19–6).
9. Select one of the following delivery rules:
   - **Send Messages to all Selected Channels** -- Select this option to send messages to every listed channel.
   - **Send to the 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 No Messages** -- Select this option to block the receipt of any messages that meet the filter conditions.
10. To set the delivery channels, select a channel from the *Add Notification Channel* list and then click **Add** (Figure 19–2). To delete a channel, click **Delete** (Figure 19–6).
11. 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 the First Available Channel.*
12. Click **OK** to create the filter. Clicking **Cancel** discards the filter.

19.3.2 Editing a Filter

To edit a filter, first select it and then click **Edit** (Figure 19–9). The editing page appears for the filter, which enables you to add or change the filter properties described in Section 19.3.1, "Creating Filters".

19.3.3 Deleting a Filter

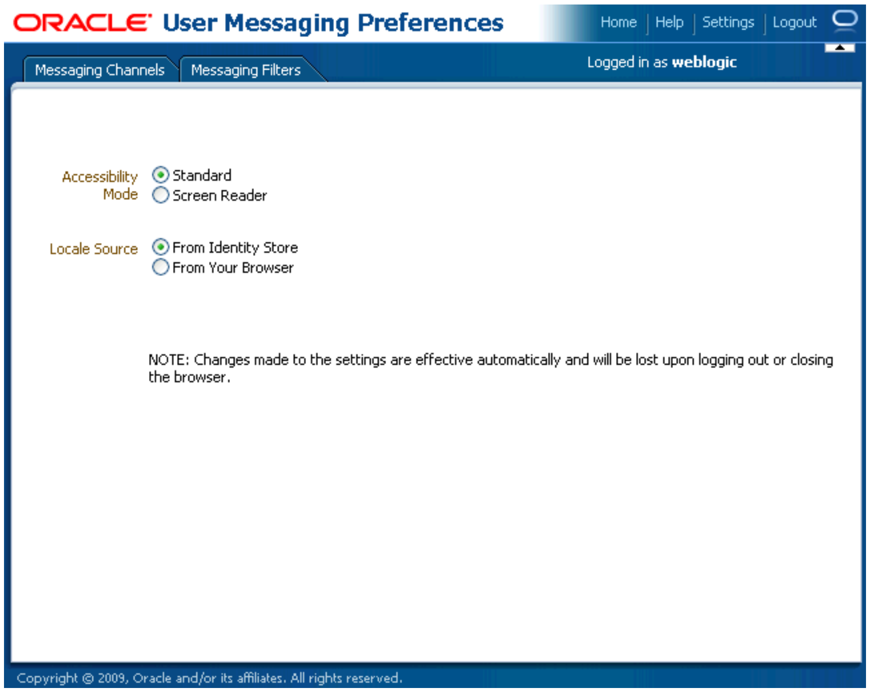
To delete a filter, first select it and then click **Delete** (Figure 19–6).

19.4 Configuring Settings

The *Settings* tab (Figure 19–10), accessed from the upper right area, enables users to set the following parameters:

- Accessibility Mode: select "Standard" or "Screen Reader."
- Locale Source: select "From Identity Store" or "From Your Browser."

Figure 19–10 Configuring Settings



Logged in as **weblogic**

Accessibility Mode: [✓ Standard] Screen Reader

Locale Source: [✓ From Identity Store] From Your Browser

NOTE: Changes made to the settings are effective automatically and will be lost upon logging out or closing the browser.

Copyright © 2009, Oracle and/or its affiliates. All rights reserved.

Part VII

Reference

This part contains reference information.

Part VIII contains the following appendices:

A

Oracle User Messaging Service Applications

This appendix describes how to create your own Oracle User Messaging Service applications using the procedures and code provided.

This chapter includes the following sections:

Note: For more information, and for code samples, refer to Oracle Technology Network (http://otn.oracle.com).

A.1 Send Message to User Specified Channel

This chapter describes how to build and run the Send Message to User Specified Channel application provided with Oracle User Messaging Service.

Note: To learn about the architecture and components of Oracle User Messaging Service, see *Oracle Fusion Middleware Getting Started with Oracle SOA Suite*.

This chapter contains the following sections:

A.1.1 Overview

The "Send Message to User Specified Channel" application demonstrates a BPEL process that allows a message to be sent to a user through a messaging channel specified in User Messaging Preferences. After you have configured a device and messaging channel addresses for each supported channel and the default device, Oracle User Messaging Service routes the message to the user based on the preferred channel setting that you configured.

A.1.1.1 Provided Files

The following files are included in the application:

- SendMessage.pdf – this document.
- Project – the directory containing Oracle JDeveloper project files.
- Readme.txt.
- Release notes

A.1.2 Installing and Configuring SOA and User Messaging Service

The installation of SOA and User Messaging Service has already been performed on your hosted instance, and the sample users have already been seeded. Perform the following steps to enable notifications in soa-infra, if not already done:

1. Using Enterprise Manager, go to "soa-infra" > (Menu) > Workflow Notification Properties, and set Notification Mode to ALL.
2. Configure the User Messaging drivers if required as described in "Configuring Drivers" in the Oracle Fusion Middleware SOA Administrator's Guide.
3. Set the email address for user "weblogic" by using the JXplorer LDAP browser. Refer to "Updating Addresses in Your LDAP User Profile".
4. Restart the server.

A.1.2.1 Updating Addresses in Your LDAP User Profile

Perform the following steps to set the email address for user "weblogic" by using the JXplorer LDAP browser:

A.1.2.1.1 Installing Download and install JXplorer from http://www.jxplorer.org.

A.1.2.1.2 Connecting **1.**Set the embedded LDAP server admin password as follows:

- Login to the WLS Admin Console.
- Click on the domain name link > Security > Embedded LDAP.
- Enter a new "Credential" and "Confirm Credential" (for example, "weblogic").
- Click **Save**.

2. Connect from JXplorer by specifying the fields in Table A–1:

Table A–1 JXplorer Connection Fields

Field	Value
Host	WLS AdminServer hostname
Port	WLS AdminServer port
Protocol	LDAP v3
Security Level	User + Password
User DN	cn=Admin
Password	<password> (for example, weblogic)

A.1.2.1.3 Setting User Messaging Device Addresses in LDAP The following example uses the user "weblogic". You may create and use additional users.

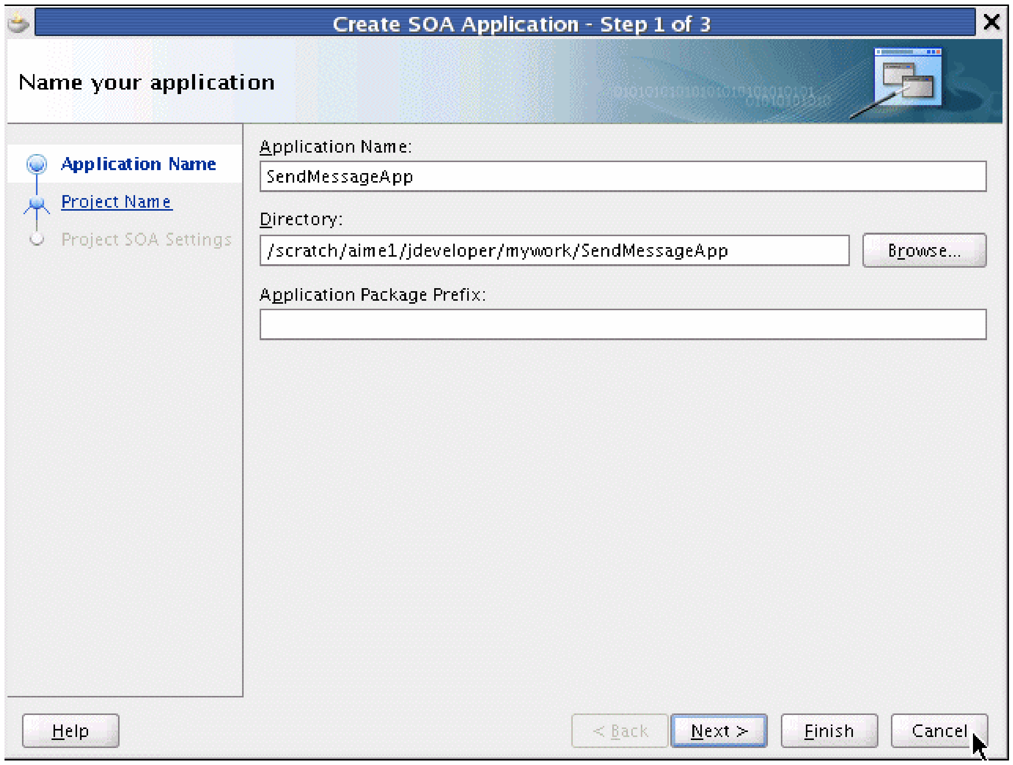
1. Expand the LDAP tree as follows: **domain** > **myrealm** > **people** > **weblogic**.
2. Click on the user entry.
3. Select the HTML view tab on the right.
4. Enter the desired Email Address and Mobile Phone Number.
5. Click **Submit**.

A.1.3 Building the Sample

Performing the following procedure of building the sample from scratch allows you to learn how to add messaging to your SOA Composite Applications, and use User Messaging Preferences.

1. Open Oracle JDeveloper 11g.
2. Create a new application by selecting **File, New**, **General**, **Applications**, and **SOA Applications**. Click **OK**.
3. Enter the *Application Name* and click **Next** (Figure A–1).

Figure A–1 Creating a New Application and Project (1 of 3)



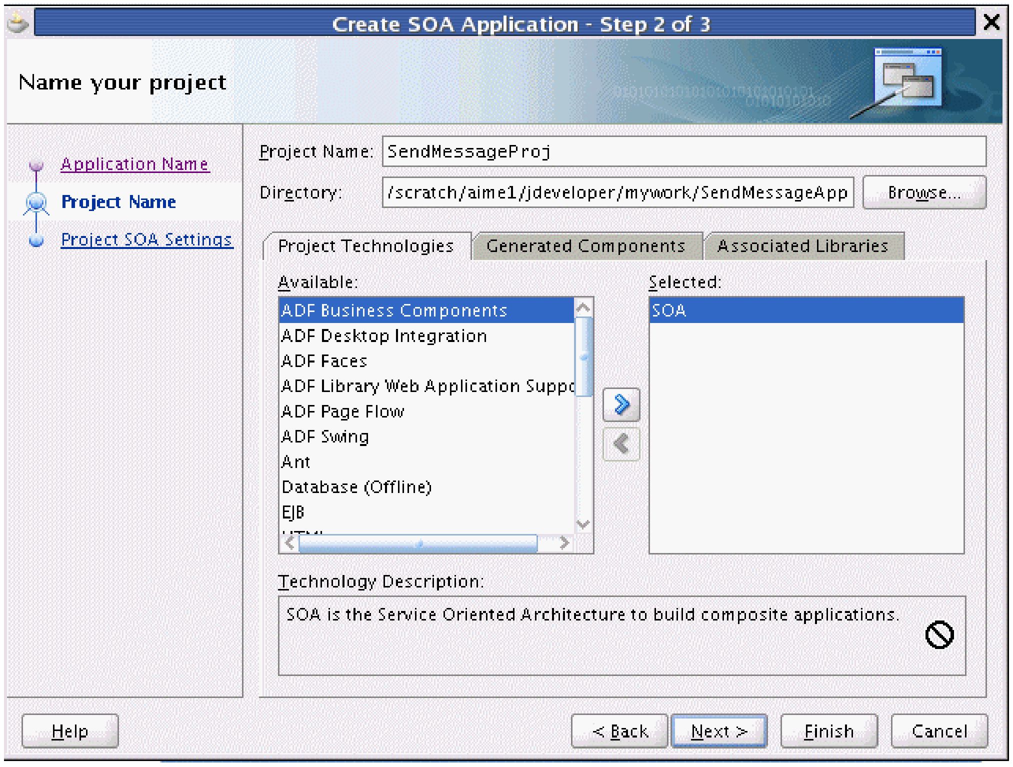
Application Name: SendMessageApp

Directory: /scratch/aime1/jdeveloper/mywork/SendMessageApp

Application Package Prefix:

4. Enter the name for the project and click **Next** (Figure A–2).

Figure A–2 Creating a New Application and Project (2 of 3)


Create SOA Application - Step 2 of 3

Name your project

Application Name
Project Name
Project SOA Settings

Project Name: SendMessageProj

Directory: /scratch/aime1/jdeveloper/mywork/SendMessageApp

Browse...

Project Technologies | Generated Components | Associated Libraries

Available:
ADF Business Components
ADF Desktop Integration
ADF Faces
ADF Library Web Application Suppo
ADF Page Flow
ADF Swing
Ant
Database (Offline)
EJB

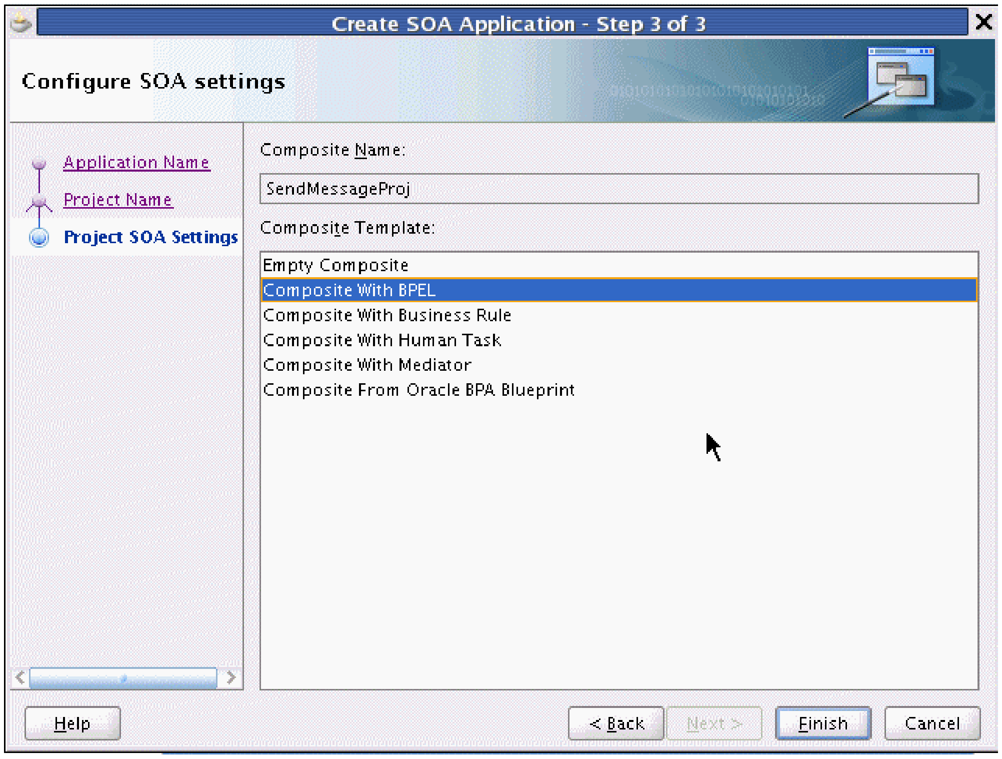
Selected:
SOA

Technology Description:
SOA is the Service Oriented Architecture to build composite applications.

Help | < Back | Next > | Finish | Cancel

5. Select the *Composite With BPEL* composite template (Figure A–3). Click **Finish**.

Figure A–3 Creating a New Application and Project (3 of 3)


Create SOA Application - Step 3 of 3

Configure SOA settings

Application Name
Project Name
Project SOA Settings

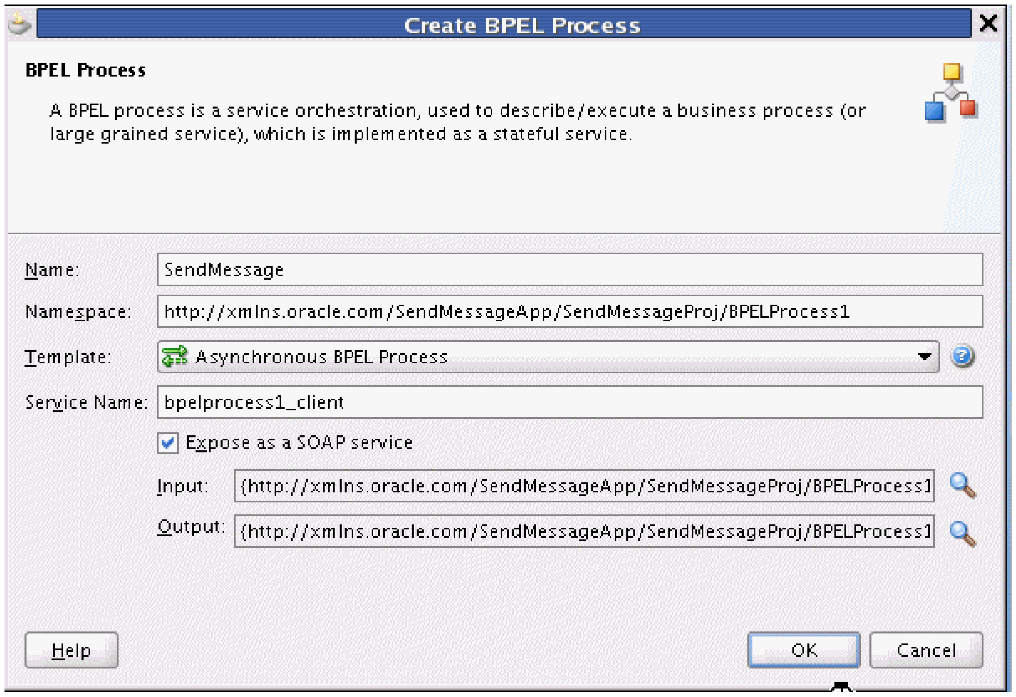
Composite Name:
SendMessageProj

Composite Template:
Empty Composite
Composite With BPEL
Composite With Business Rule
Composite With Human Task
Composite With Mediator
Composite From Oracle BPA Blueprint

Help | < Back | Next > | Finish | Cancel

6. In the *Create BPEL Process* window, enter the BPEL process name as "SendMessage" (Figure A–4). Click **OK**.

Figure A–4 Creating the BPEL Process



Create BPEL Process

BPEL Process

A BPEL process is a service orchestration, used to describe/execute a business process (or large grained service), which is implemented as a stateful service.

Name: SendMessage

Namespace: http://xmlns.oracle.com/SendMessageApp/SendMessageProj/BPELProcess1

Template: Asynchronous BPEL Process

Service Name: bpelprocess1_client

✓ Expose as a SOAP service

Input: {http://xmlns.oracle.com/SendMessageApp/SendMessageProj/BPELProcess1

Output: {http://xmlns.oracle.com/SendMessageApp/SendMessageProj/BPELProcess1

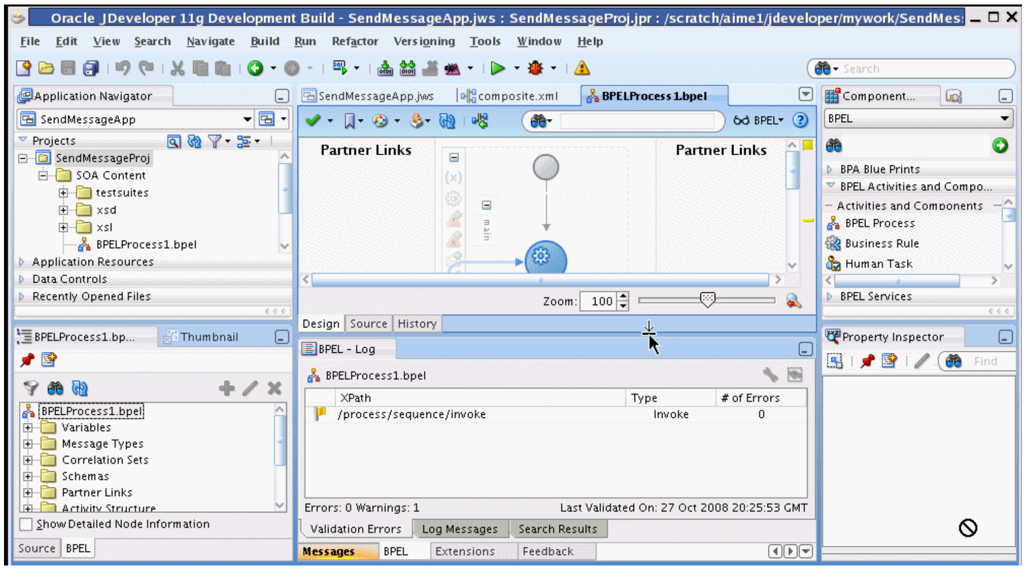
7. Verify that "Expose as a SOAP service" is checked. Click OK.
8. You have now created an empty and default BPEL application (Figure A–5).

   In the Oracle JDeveloper main window you can view the following components of the application under the *Composite.xml* tab.

   - The left box is the definition of a Web Service client that is used to initiate an application.
   - The middle box is a BPEL process that creates and formats the message and calls the messaging service.

Note: You must later create the messaging service resource that is used to send the message when you create the User Notification BPEL process (steps 13-19).

Figure A–5 Empty and Default BPEL Application



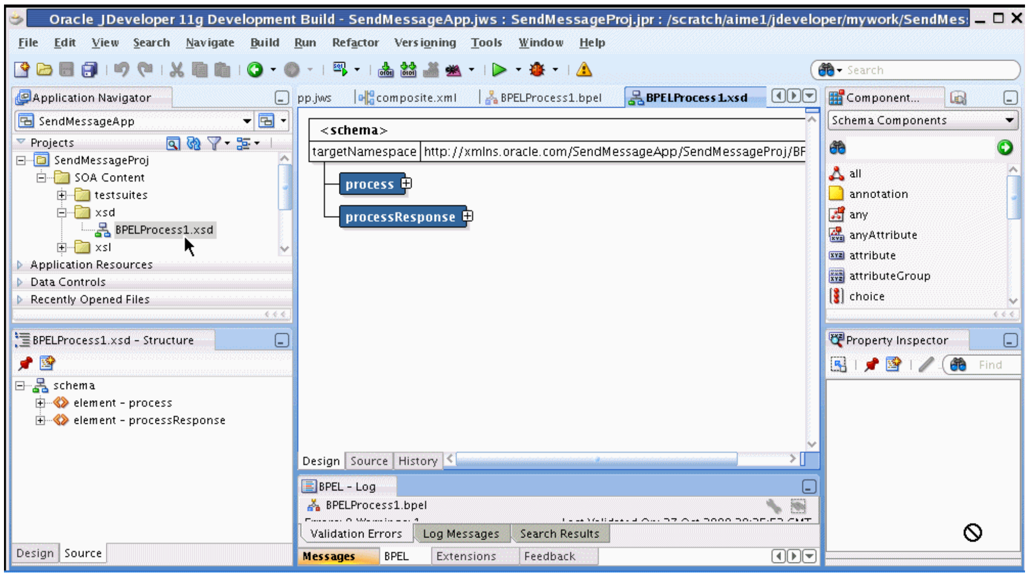
BPELProcess1.bpel

XPath	Type	# of Errors
/process/sequence/invoke	Invoke	0

Errors: 0 Warnings: 1   Last Validated On: 27 Oct 2008 20:25:53 GMT

9. Expand the *xsd* folder in the Application Navigator and open *BPELProcess1.xsd* by double-clicking it (Figure A–6).

Figure A–6 Accessing the BPELProcess1.xsd File



targetNamespace http://xmlns.oracle.com/SendMessageApp/SendMessageProj/BF

10. Click on the **Source** tab (Figure A–7).

11. Perform the following modifications to the inputs of this BPEL application:

    In the generated file, SendMessage.xsd, in the xsd folder in the application navigator under projects, the following element definition is created by default:

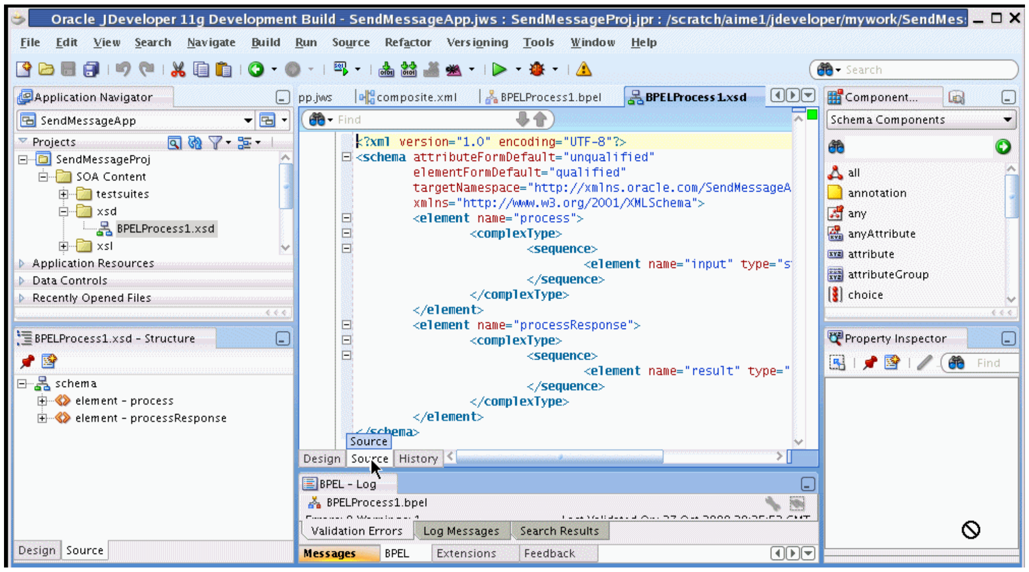
    ```
    <element name="input" type="string"/>
    ```

    This xsd element defines the input for the BPEL process.

    Select the *Source* tab (Figure A–7), and replace the line above with the following three lines:

```
<element name="to" type="string"/>
<element name="subject" type="string"/>
<element name="body" type="string"/>
```

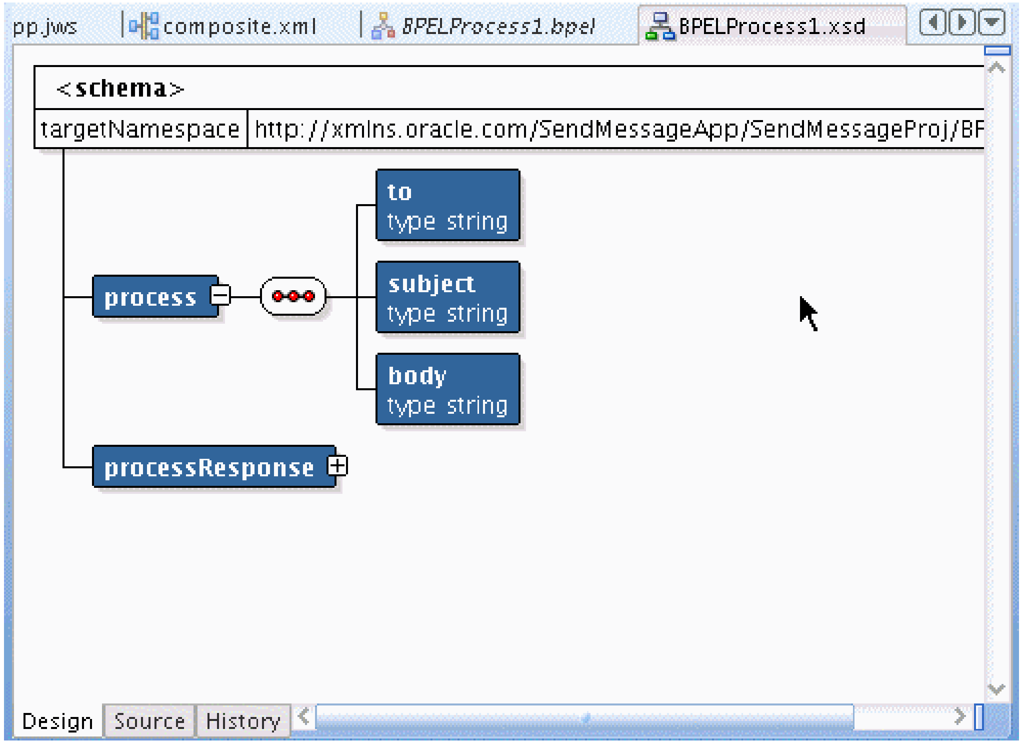
Figure A–7   Modifying the Inputs in the BPELProcess1.xsd File



```
<?xml version="1.0" encoding="UTF-8"?>
<schema attributeFormDefault="unqualified"
        elementFormDefault="qualified"
        targetNamespace="http://xmlns.oracle.com/SendMessageA
        xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="process">
                <complexType>
                        <sequence>
                                <element name="input" type="s
                        </sequence>
                </complexType>
        </element>
        <element name="processResponse">
                <complexType>
                        <sequence>
                                <element name="result" type="
                        </sequence>
                </complexType>
        </element>
</schema>
```

12. Perform a File, **Save All**.
13. View the expanded process element (Figure A–8).

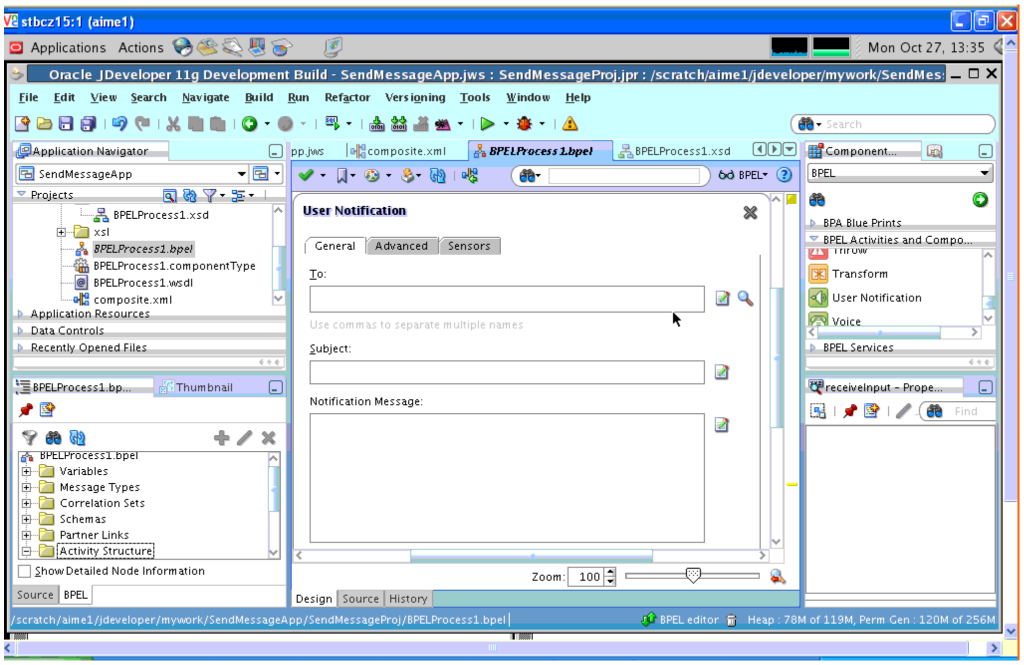
Figure A–8   Viewing the Expanded Process Element



14. To enable messaging in this process, drag and drop **User Notification** from *BPEL Activities and Components* located in the *Component Palette* between the *receiveInput* and *callbackClient* activities.

    The User Notification activity appears (Figure A–9).

Figure A–9 User Notification Activity Before Configuring the Inputs

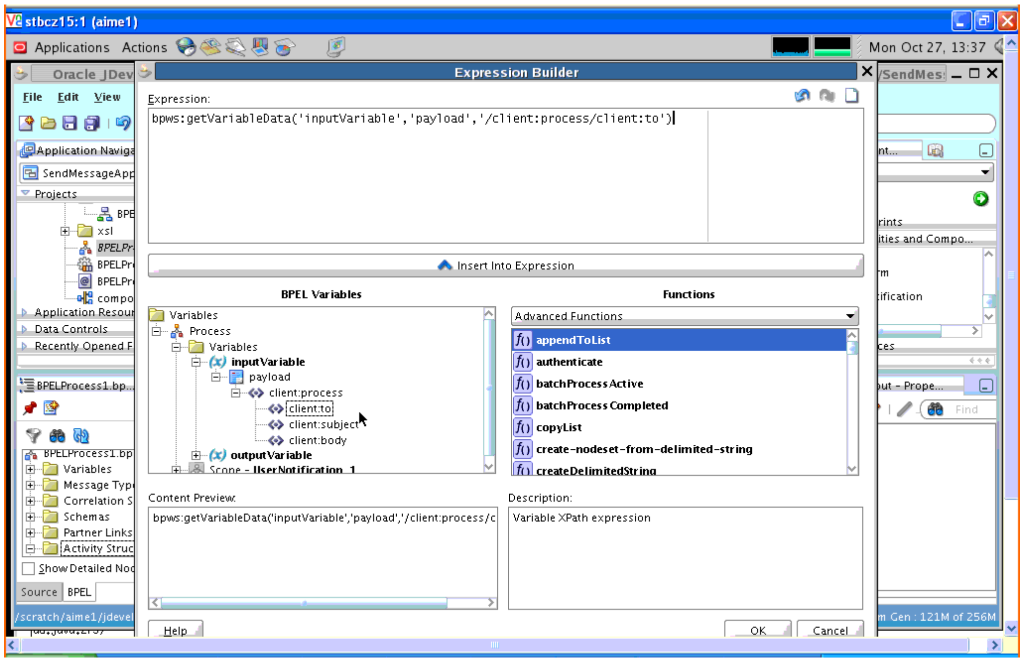


15. Click the XPath Expression Builder icon to the right of the "To:" input box.

16. Modify the expression for the recipient, "to", as follows:

    - In the BPEL Variables pane, select **Variables**, **inputVariable**, **Payload**, **clientprocess**, and **client:to** (Figure A–10).
    - Click **Insert Into Expression**.
    - Click **OK**.

Figure A–10 Defining the Recipient ("to") Expression

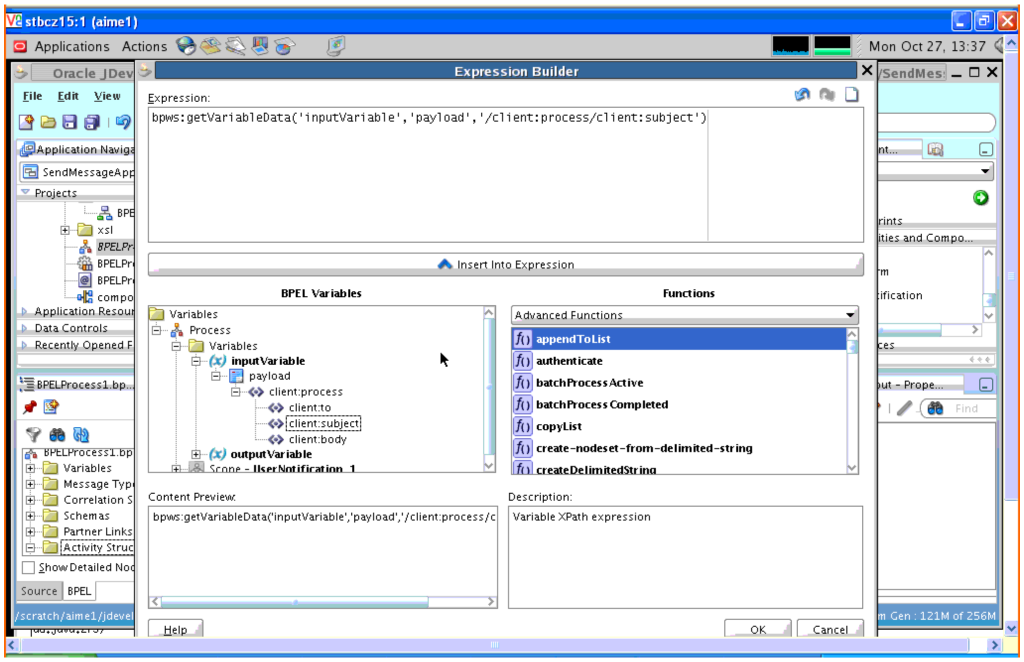


```
bpws:getVariableData('inputVariable','payload','/client:process/client:to')
```

Description: Variable XPath expression

17. Click the XPath Expression Builder icon to the right of the "subject:" input box.
18. Modify the expression for the *subject* as follows:
    - In the BPEL Variables pane, select **Variables**, **InputVariable**, **Payload**, **clientprocess**, and **client:subject** (Figure A–11).
    - Click **Insert Into Expression**.
    - Click **OK**.

Figure A–11 Defining the Subject Expression

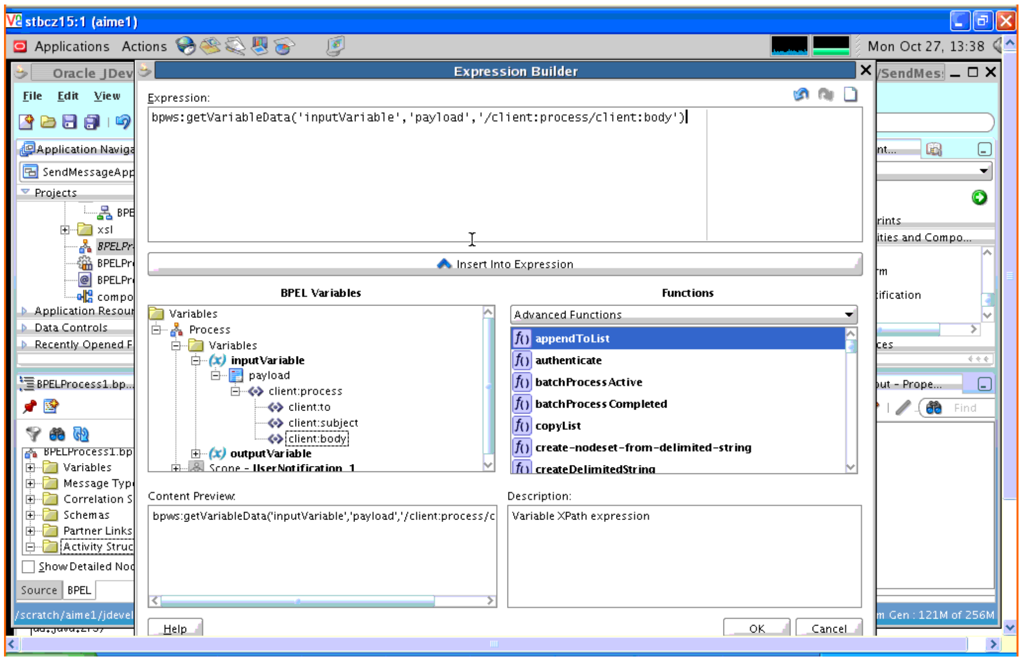


```
bpws:getVariableData('inputVariable','payload','/client:process/client:subject')
```

Description: Variable XPath expression

19. Click the XPath Expression Builder icon to the right of the "body:" input box.
20. Modify the expression for the *body* as follows:
    - In the BPEL Variables pane, select **Variables**, **InputVariable**, **Payload**, **clientprocess**, and **client:body** (Figure A–12).
    - Click **Insert Into Expression**.

Figure A–12 Defining the Body Expression



Expression:

```
bpws:getVariableData('inputVariable','payload','/client:process/client:body')
```

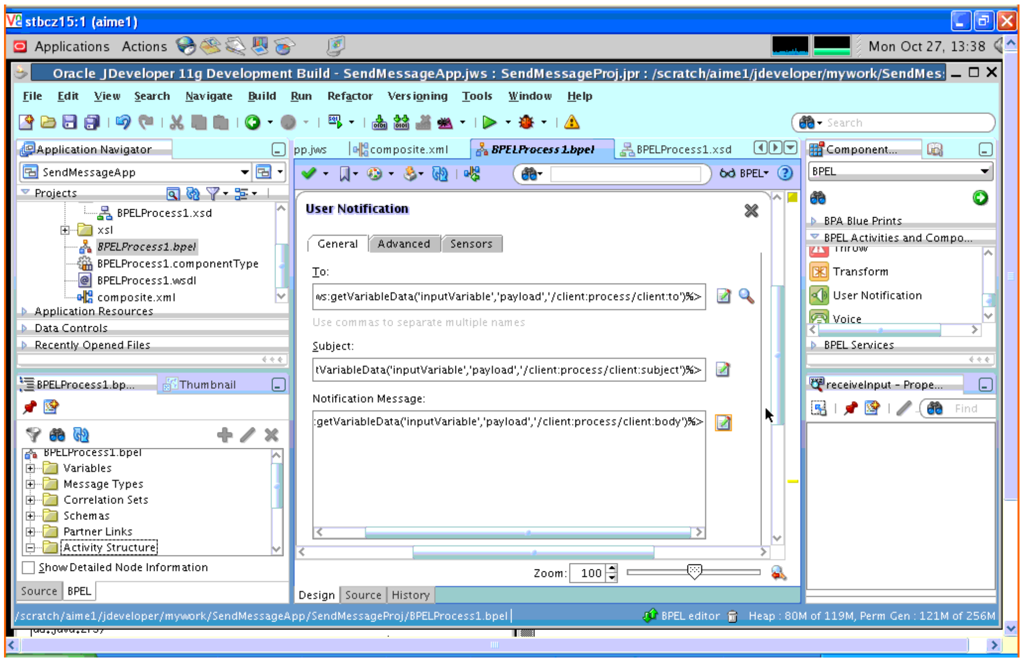
Content Preview:

```
bpws:getVariableData('inputVariable','payload','/client:process/c
```

Description: Variable XPath expression

- Click **OK**.
- Click **Apply and then OK** to apply the changes (Figure A–13).

Figure A–13 Confirming the Changes to the Inputs



To:

```
ws:getVariableData('inputVariable','payload','/client:process/client:to')%>
```

Subject:

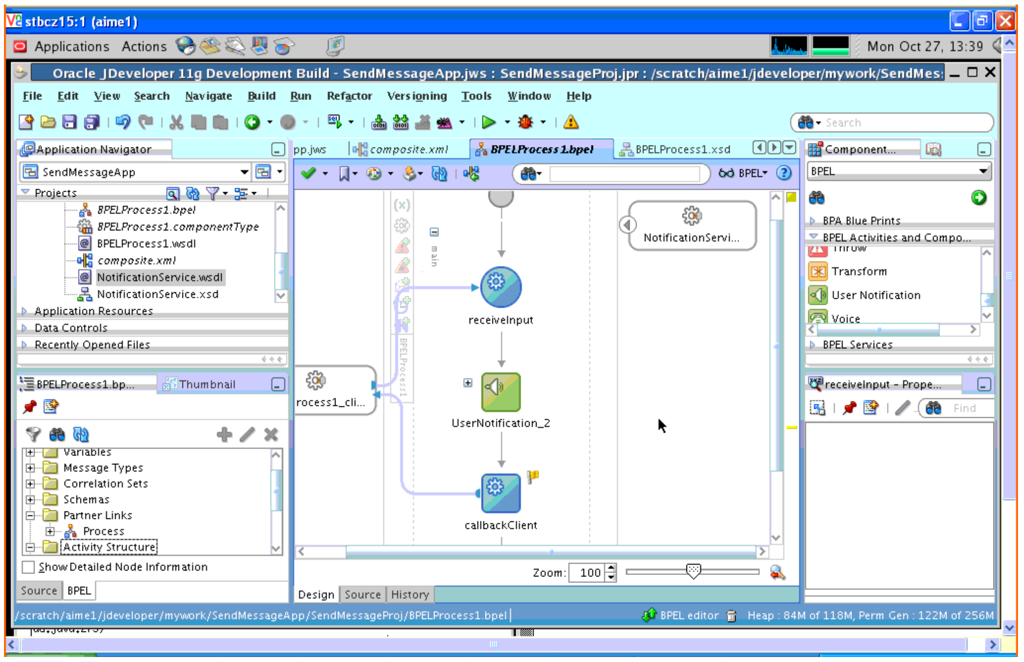
```
tVariableData('inputVariable','payload','/client:process/client:subject')%>
```

Notification Message:

```
:getVariableData('inputVariable','payload','/client:process/client:body')%>
```

The changes to the inputs are saved and the configuration of the User Notification Activity is complete. You can now see the User Notification Activity in the BPEL application (Figure A–14). The SOA Composite is complete.

Figure A–14 User Notification Activity After Configuration of Inputs

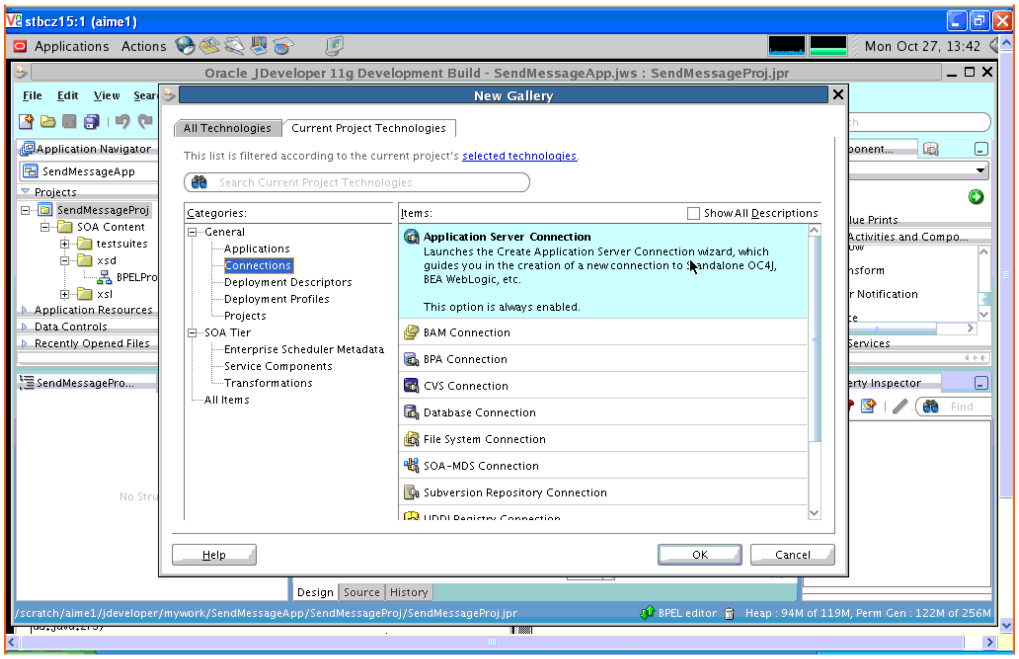


A.1.4 Creating a New Application Server Connection

Perform the following steps to create a new Application Server Connection.

1. Create a new Application Server Connection by right-clicking the project and selecting **New**, **Connections**, and **Application Server Connection** (Figure A–15).

Figure A–15 New Application Server Connection



This list is filtered according to the current project's selected technologies.

Application Server Connection
Launches the Create Application Server Connection wizard, which guides you in the creation of a new connection to standalone OC4J, BEA WebLogic, etc.

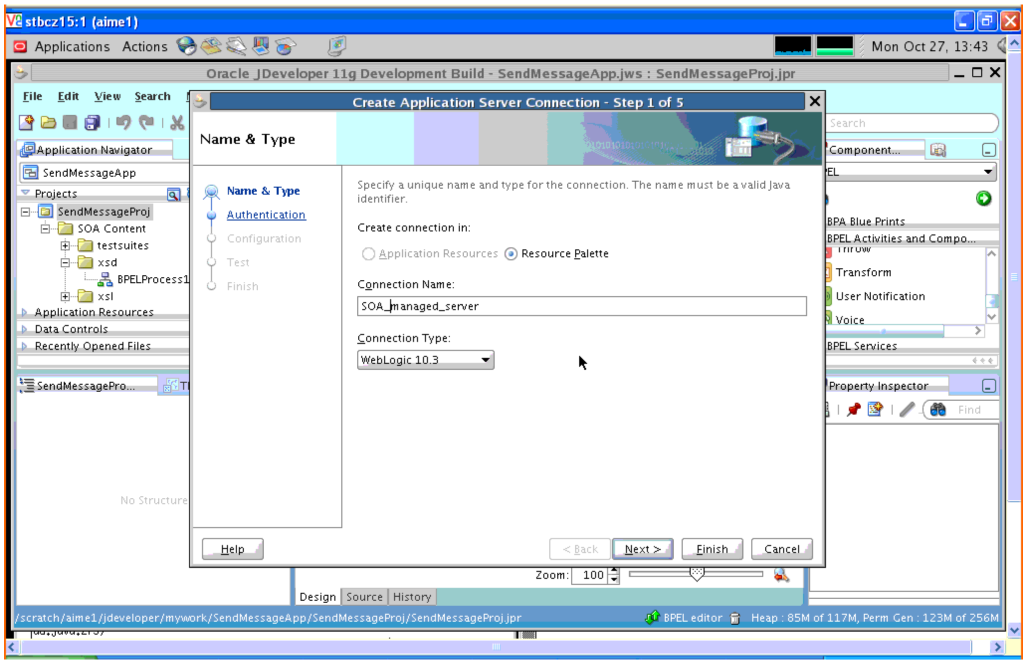
This option is always enabled.

/scratch/aime1/jdeveloper/mywork/SendMessageApp/SendMessageProj/SendMessageProj.jpr

Heap : 94M of 119M, Perm Gen : 122M of 256M

2. Name the connection “SOA_server” and click **Next** (Figure A–16).
3. Select "WebLogic 10.3" as the *Connection Type*.

Figure A–16 New Application Server Connection



Specify a unique name and type for the connection. The name must be a valid Java identifier.

Create connection in:

Application Resources [✓ Resource Palette]

Connection Name: SOA_managed_server

Connection Type: WebLogic 10.3

/scratch/aime1/jdeveloper/mywork/SendMessageApp/SendMessageProj/SendMessageProj.jpr

Heap : 85M of 117M, Perm Gen : 123M of 256M

4. Enter the authentication information. The typical values are:

Username: `weblogic`
Password: `weblogic`

5. On the *Connection* screen, enter the hostname, port and SSL port for the SOA admin server, and enter the name of the domain for WLS Domain.
6. Click **Next**.
7. On the *Test* screen click **Test Connection**.
8. Verify that the message "Success!" appears.

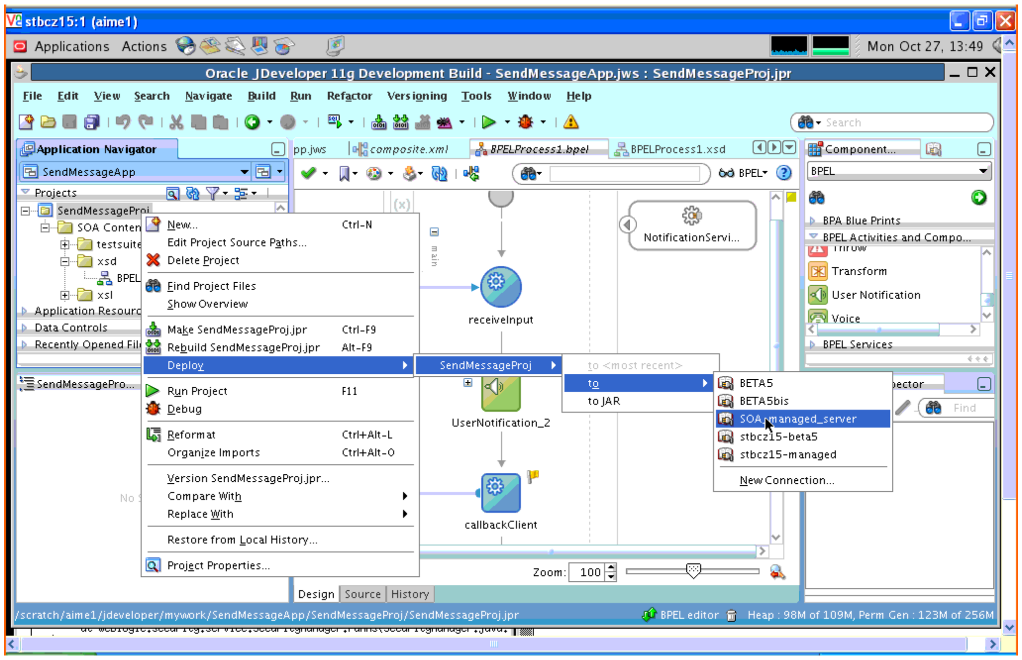
   The Application Server Connection has been created.

A.1.5 Deploying the Project

Perform the following steps to deploy the project:

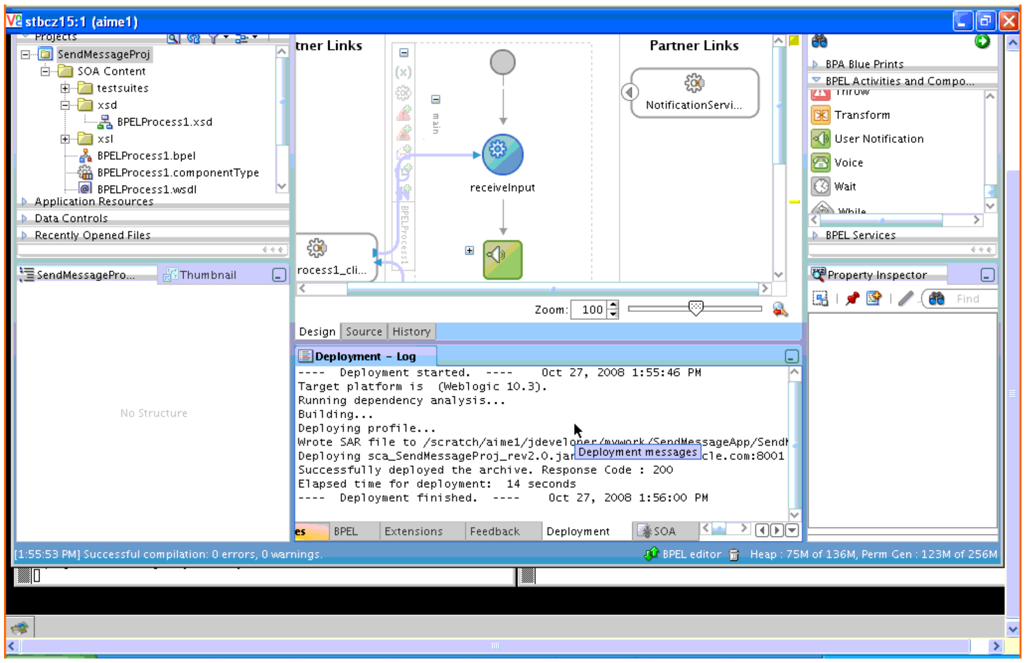
1. Deploy the project by selecting the **SendMessage project**, **Deploy**, **SendMessageProj**, **to**, and **SOA_server** (Figure A–17).

Figure A–17 Deploying the Project



2. Verify that the message "Build Successful" appears in the log.
3. Enter the default revision and click **OK**.
4. Verify that the message "Deployment Finished" appears in the deployment log (Figure A–18).

Figure A–18 Verifying that the Deployment is Successful



```
----  Deployment started.  ----    Oct 27, 2008 1:55:46 PM
Target platform is  (Weblogic 10.3).
Running dependency analysis...
Building...
Deploying profile...
Wrote SAR file to /scratch/aime1/jdeveloper/mywork/SendMessageApp/SendM
Deploying sca_SendMessageProj_rev2.0.jar                          cle.com:8001
Successfully deployed the archive. Response Code : 200
Elapsed time for deployment:  14 seconds
----  Deployment finished.  ----    Oct 27, 2008 1:56:00 PM
```

```
[1:55:53 PM] Successful compilation: 0 errors, 0 warnings.
```

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 Messaging Preferences, as described in the following sections.

Note: Refer to "Configuring Notifications" in the *Oracle Fusion Middleware SOA Developer's Guide* for more information.

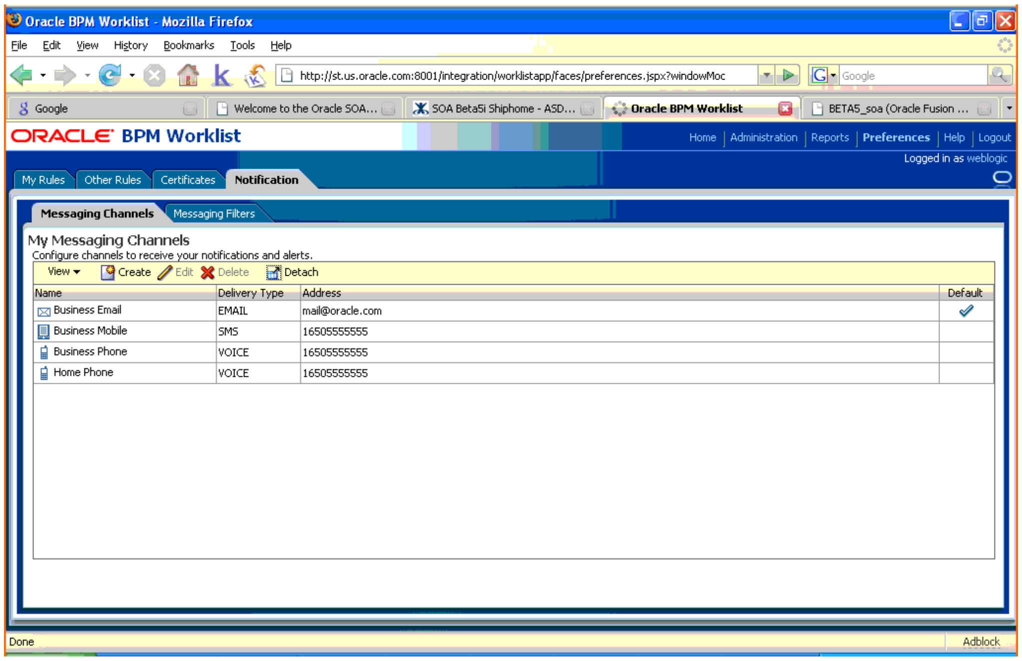
A.1.6 Configuring User Messaging Preferences

For users to receive the notifications, they must register the devices that they use to access messages through User Messaging Preferences. Perform the following steps:

1. Log into the User Messaging Preferences application at one of the following URLs:
   - Directly at http://<server>:<port>/sdpmessaging/userprefs-ui
   - Through the Worklist application's Preferences > Notification tab at: http://<server>:<port>/integration/worklistapp

   The User Messaging Preferences application appears.
2. Click on the *Messaging Channels* tab (Figure A–19).

Figure A–19 Messaging Channels Tab


Oracle BPM Worklist - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://st.us.oracle.com:8001/integration/worklistapp/faces/preferences.jspx?windowMoc
Google
Welcome to the Oracle SOA...
SOA Beta5i Shiphome - ASD...
Oracle BPM Worklist
BETA5_soa (Oracle Fusion ...
ORACLE BPM Worklist
Home | Administration | Reports | Preferences | Help | Logout
Logged in as weblogic
My Rules
Other Rules
Certificates
Notification
Messaging Channels
Messaging Filters
View
Create
Edit
Delete
Detach
Done
Adblock

My Messaging Channels

Configure channels to receive your notifications and alerts.

Name	Delivery Type	Address	Default
Business Email	EMAIL	mail@oracle.com	✓
Business Mobile	SMS	16505555555	
Business Phone	VOICE	16505555555	
Home Phone	VOICE	16505555555	

You are prompted for login credentials.

3. In the Messaging Channels tab, select a channel.

4. Set a channel as the default by expanding the device folder, and then clicking **Set as Default** adjacent to the selected channel.

   A checkmark appears next to the selected channel, designating it as the default means of receiving notifications. All messages sent to that user are sent to that channel.

A.1.7 Testing the Sample

The following steps describe how to perform a test message transmission through Enterprise Manager.

Perform the following steps to run and test the sample:

1. Open a Web browser window and login to Enterprise Manager for the SOA domain. For example, `http://<host>:<port>/em`.

2. In Enterprise Manager, expand the SOA folder in the navigation tree, and click the deployed *SendMessageProj* composite application. Click the **Test** button to launch the test client page.

3. In the **Input Arguments** section provide the input values for invoking *SendMessageProj*.

   Enter the following values:

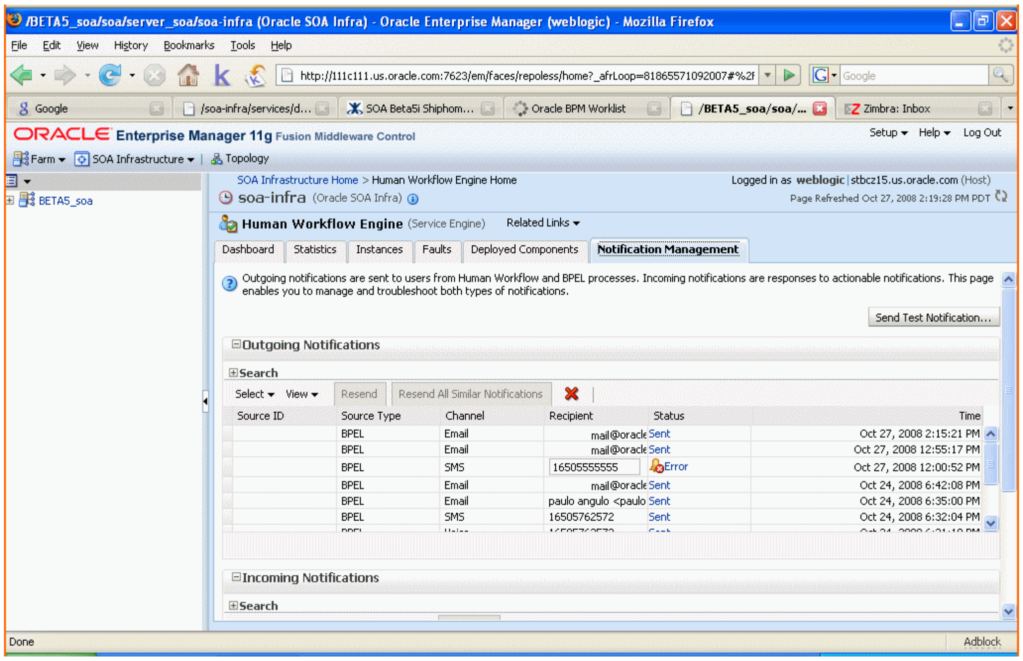
   - to: weblogic (the user)
   - subject: notification test (the subject)
   - body: the message content

4. Click **Test Web Service**.

A.1.7.1 Verifying the Execution of Sending the E-mail

Log in to the Human Workflow Engine. Verify the outgoing notifications and their statuses from the Notification Manager tab. (Figure A–20).

Figure A–20 Viewing Outgoing Notifications


/BETA5_soa/soa/server_soa/soa-infra (Oracle SOA Infra) - Oracle Enterprise Manager (weblogic) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://111c111.us.oracle.com:7623/em/faces/repoless/home?_afrLoop=81865571092007#%2f
Google
Google
/soa-infra/services/d...
SOA Beta5i Shiphom...
Oracle BPM Worklist
/BETA5_soa/soa/...
Zimbra: Inbox
ORACLE Enterprise Manager 11g Fusion Middleware Control
Setup Help Log Out
Farm SOA Infrastructure Topology
BETA5_soa
SOA Infrastructure Home > Human Workflow Engine Home
Logged in as weblogic | stbcz15.us.oracle.com (Host)
soa-infra (Oracle SOA Infra)
Page Refreshed Oct 27, 2008 2:19:28 PM PDT
Human Workflow Engine (Service Engine)
Related Links
Dashboard
Statistics
Instances
Faults
Deployed Components
Notification Management
Send Test Notification...
Outgoing Notifications
Search
Select View
Resend
Resend All Similar Notifications
Incoming Notifications
Search
Done
Adblock

Outgoing notifications are sent to users from Human Workflow and BPEL processes. Incoming notifications are responses to actionable notifications. This page enables you to manage and troubleshoot both types of notifications.

Source ID	Source Type	Channel	Recipient	Status	Time
	BPEL	Email	mail@oracle	Sent	Oct 27, 2008 2:15:21 PM
	BPEL	Email	mail@oracle	Sent	Oct 27, 2008 12:55:17 PM
	BPEL	SMS	16505555555	Error	Oct 27, 2008 12:00:52 PM
	BPEL	Email	mail@oracle	Sent	Oct 24, 2008 6:42:08 PM
	BPEL	Email	paulo angulo <paulo	Sent	Oct 24, 2008 6:35:00 PM
	BPEL	SMS	16505762572	Sent	Oct 24, 2008 6:32:04 PM

A.2 Send Email with Attachments

This chapter describes how to build and run the Send Email with Attachments application provided with Oracle User Messaging Service.

Note: To learn about the architecture and components of Oracle User Messaging Service, see *Oracle Fusion Middleware Getting Started with Oracle SOA Suite*.

This chapter contains the following sections:

- Section A.2.1, "Overview"
- Section A.2.2, "Installing and Configuring SOA and User Messaging Service"
- Section A.2.3, "Running the Pre-Built Sample"
- Section A.2.4, "Testing the Sample"
- Section A.2.5, "Building the Sample"
- Section A.2.6, "Creating a New Application Server Connection"

A.2.1 Overview

The "Send Email With Attachment" application demonstrates a BPEL process that sends an e-mail with an attached file.

A BPEL process looks up a user's e-mail address from the identity store, reads a file from the file system, creates e-mail content and then sends an email to the user.Section A.2.5, "Building the Sample" shows you how to add an e-mail with attachments to your SOA composite application, allowing your applications to be enabled with messaging.If you want to model the application from scratch, go to the section titled Building the Sample. Or, you can directly use the pre-built project provided with this tutorial.

Before you run the pre-built sample or build the application from scratch, you must install and configure the server as described in Section A.2.2, "Installing and Configuring SOA and User Messaging Service". By default, soa-infra does not send out notifications. The following steps describe installing and configuring the e-mail drivers needed to communicate with the e-mail server.

A.2.1.1 Provided Files

The following files are included in the sample application:

- ns_sendemail.pdf – this document.
- Project – the directory containing Oracle JDeveloper project files.
- Readme.txt.
- Release notes

A.2.2 Installing and Configuring SOA and User Messaging Service

The installation of SOA and User Messaging Service has already been performed on your hosted instance, and the sample user, "weblogic", has already been created. Perform the following steps to enable notifications in soa-infra, if not already done:

1. Using Enterprise Manager, go to "soa-infra" > (Menu) > Workflow Notification Properties, and set Notification Mode to ALL.
2. Configure the User Messaging drivers if required as described in "Configuring Drivers" in the Oracle Fusion Middleware SOA Administrator's Guide.
3. Set the email address for user "weblogic" by using the JXplorer LDAP browser. Refer to "Updating Addresses in Your LDAP User Profile".
4. Restart the server.

A.2.2.1 Updating Addresses in Your LDAP User Profile

Perform the following steps to set the email address for user "weblogic" by using the JXplorer LDAP browser:

A.2.2.1.1 Installing Download and install JXplorer from http://www.jxplorer.org.

A.2.2.1.2 Connecting **1.**Set the embedded LDAP server admin password as follows:

- Login to the WLS Admin Console.
- Click on the domain name link > Security > Embedded LDAP.
- Enter a new "Credential" and "Confirm Credential" (for example, "weblogic").
- Click **Save**.

2. Connect from JXplorer by specifying the fields in Table A–2:

Table A–2 JXplorer Connection Fields

Field	Value
Host	WLS AdminServer hostname
Port	WLS AdminServer port
Protocol	LDAP v3
Security Level	User + Password
User DN	cn=Admin
Password	<password> (for example, weblogic)

A.2.2.1.3 Setting User Messaging Device Addresses in LDAP The following example uses the user "weblogic". You may create and use additional users.

1. Expand the LDAP tree as follows: **domain** > **myrealm** > **people** > **weblogic**.
2. Click on the user entry.
3. Select the HTML view tab on the right.
4. Enter the desired Email Address and Mobile Phone Number.
5. Click **Submit**.

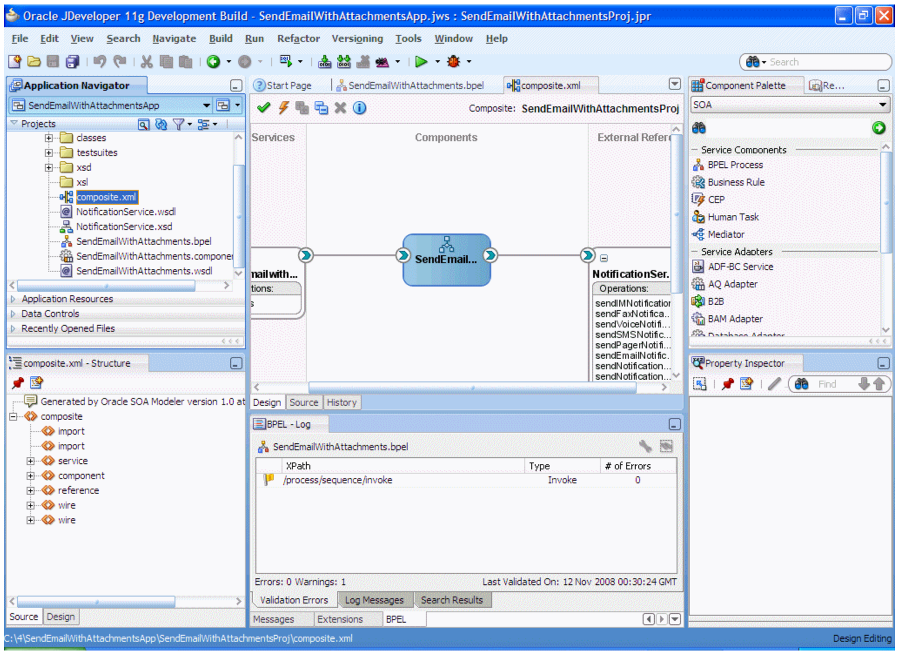
A.2.3 Running the Pre-Built Sample

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

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

   In the Oracle JDeveloper main window you can view the following components of the sample application under the *Composite.xml* tab.

Figure A–21 Oracle JDeveloper Main Window



XPath	Type	# of Errors
/process/sequence/invoke	Invoke	0

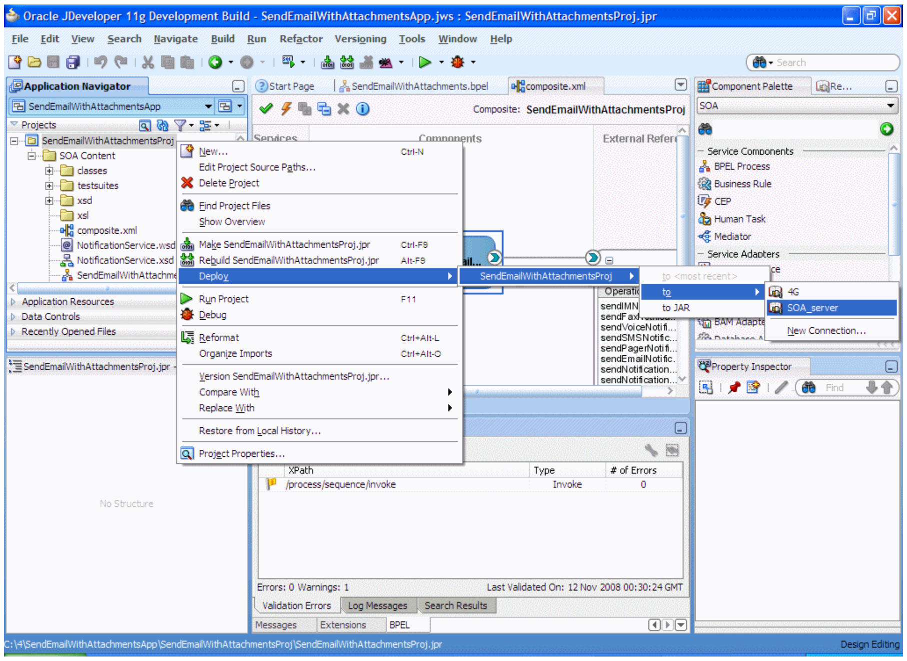
Errors: 0 Warnings: 1 Last Validated On: 12 Nov 2008 00:30:24 GMT

C:\4\SendEmailWithAttachmentsApp\SendEmailWithAttachmentsProj\composite.xml

- The left box is the definition of a Web Service client that is used to initiate an application.
- The middle box is a BPEL process that creates and formats the message and calls the messaging service.
- The right box is the messaging service resource that is used to send the message.

2. Create an Application Server Connection by right-clicking the project in the navigation pane and selecting New. Follow the instructions in Section A.2.6, "Creating a New Application Server Connection".
3. Deploy the project by selecting the **SendEmail project**, **Deploy**, **SendEmailProj**, **to**, and **SOA_server** (Figure A–22).

Figure A–22 Deploying the Project



4. Verify that the message "Build Successful" appears in the log.
5. Enter the default revision and click **OK**.
6. 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 Messaging Preferences, as described in the following sections.

Note: Refer to "Configuring Notifications" in the *Oracle Fusion Middleware SOA Developer's Guide* for more information.

A.2.4 Testing the Sample

The following steps describe how to perform a test message transmission through Enterprise Manager.

Perform the following steps to run and test the sample:

1. Open a Web browser window and login to Enterprise Manager for the SOA domain. For example, `http://<host>:<port>/em`.
2. In Enterprise Manager, expand the SOA folder in the navigation tree, and click the deployed *SendEmailWithAttachmentsProj* composite application. Click the **Test** button to launch the test client page.
3. In the **Input Arguments** section provide the input values for invoking *SendEmailWithAttachmentsProj*.

   Enter the following values:

   - to: weblogic (the user)
   - subject: notification test (the subject)
   - body: the message content
   - attachmentName: the name of the being attached, including extension.
   - attachmentMimeType: for example, image/gif.

     To send files such as pdf, doc, gif or jpeg files, the following values can be used for the attachmentMimeType entry:

     – file-name.doc – attachmentMimeType: application/msword

     – file-name.pdf – attachmentMimeType: application/pdf

     – file-name.jpg – attachmentMimeType: image/jpeg

     – file-name.gif – attachmentMimeType: image/gif
   - attachmentURI: the URI for the attachment
4. Click **Test Web Service**.

A.2.4.1 Verifying the Execution

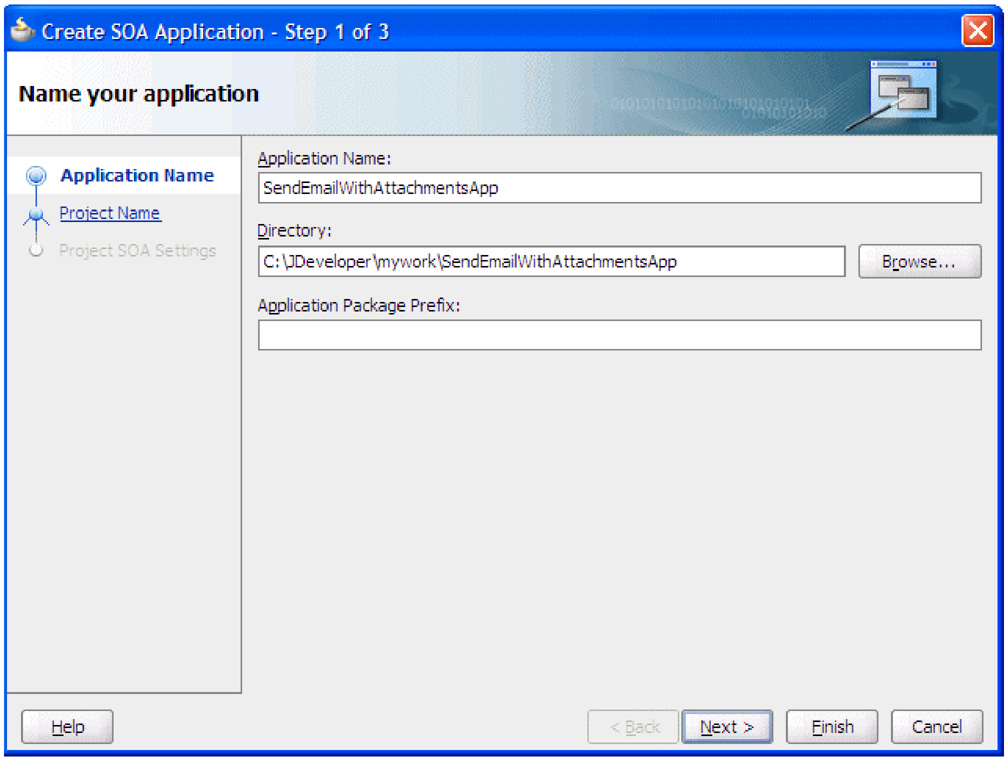
Check the weblogic e-mail account to verify it has received an email with attachment.

A.2.5 Building the Sample

Performing the following procedure of building the sample from scratch allows you to learn how to add messaging to your SOA Composite Applications, and use User Messaging Preferences.

1. Open Oracle JDeveloper 11g.
2. Create a new application by selecting **File, New**, **Applications**, and **SOA Application**. Click **OK**.
3. Enter the *Application Name* and click **Next** (Figure A–23).

Figure A–23 Creating a New Application and Project (1 of 3)


Create SOA Application - Step 1 of 3

Name your application

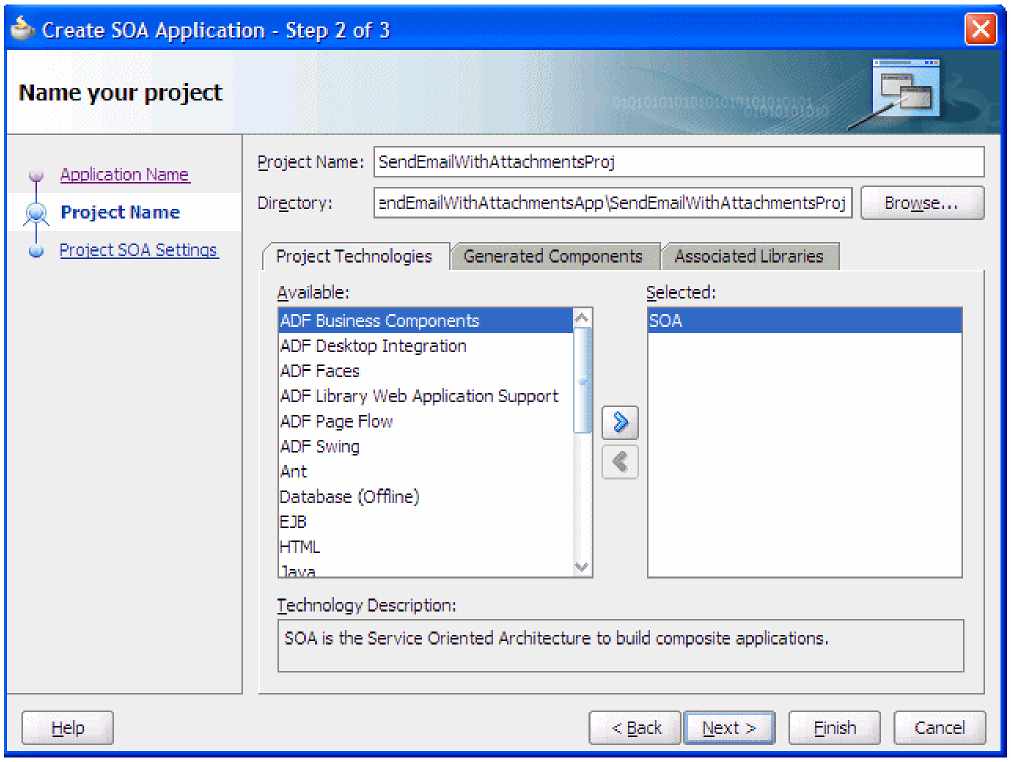
Application Name: SendEmailWithAttachmentsApp

Directory: C:\JDeveloper\mywork\SendEmailWithAttachmentsApp

Application Package Prefix:

4. Enter the name for the project and click **Next** (Figure A–24).

Figure A–24 Creating a New Application and Project (2 of 3)


Create SOA Application - Step 2 of 3

Name your project

Project Name: SendEmailWithAttachmentsProj

Directory: endEmailWithAttachmentsApp\SendEmailWithAttachmentsProj

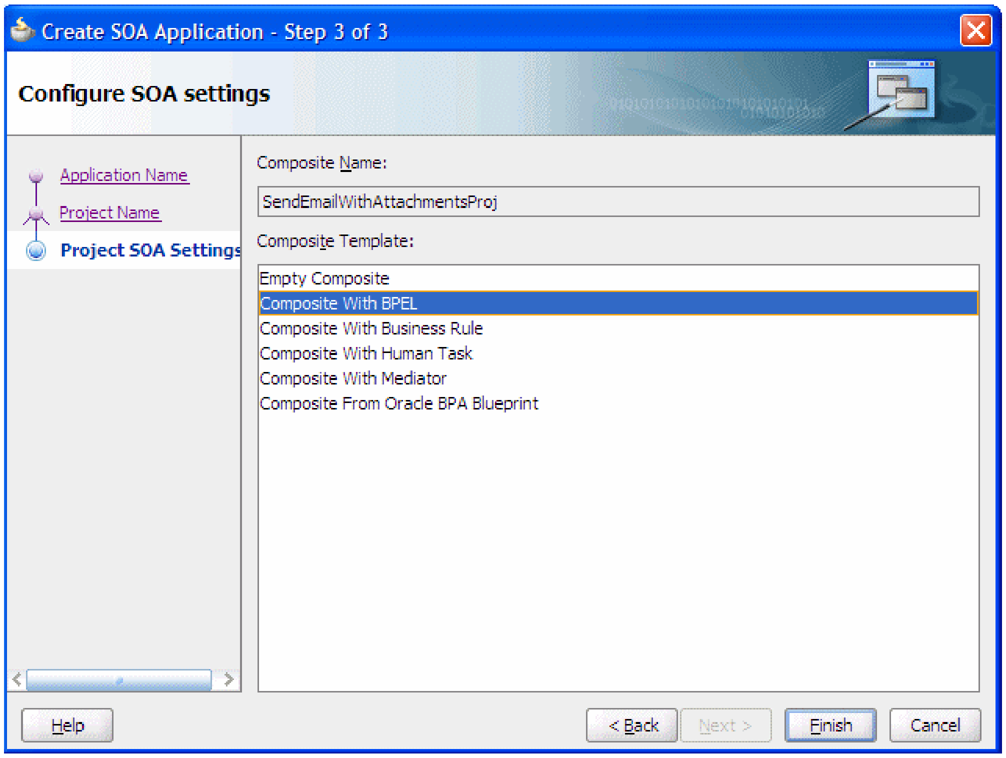
Available:
ADF Business Components
ADF Desktop Integration
ADF Faces
ADF Library Web Application Support
ADF Page Flow
ADF Swing
Ant
Database (Offline)
EJB
HTML
Java

Selected:
SOA

Technology Description:
SOA is the Service Oriented Architecture to build composite applications.

5. Select the *Composite With BPEL* composite template (Figure A–25). Click **Finish**.

Figure A–25 Creating a New Application and Project (3 of 3)


Create SOA Application - Step 3 of 3

Configure SOA settings

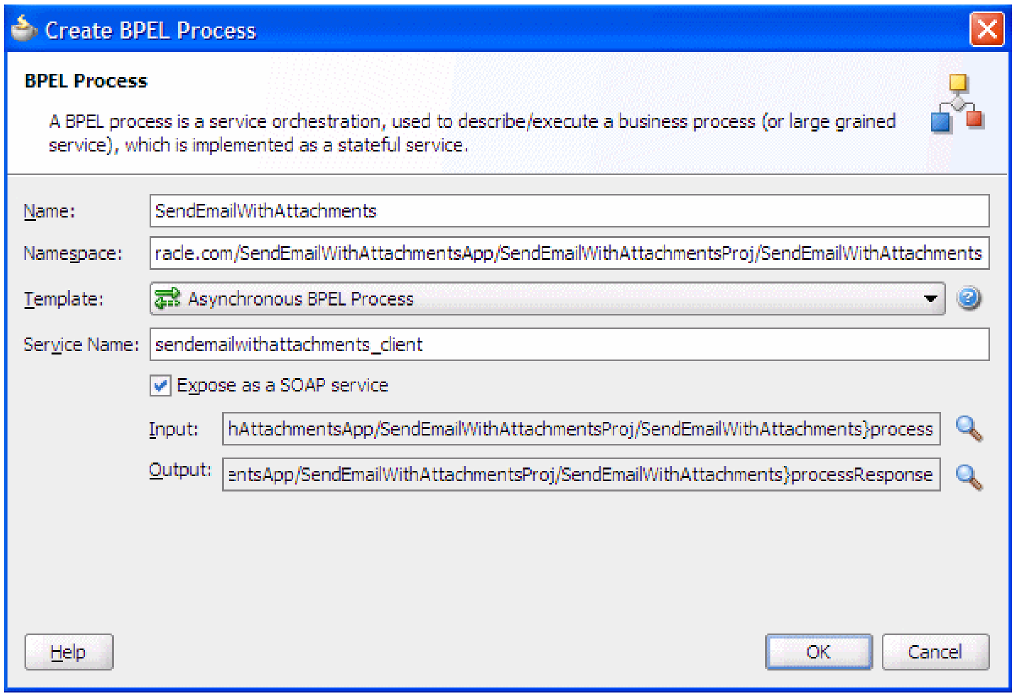
Application Name
Project Name
Project SOA Settings

Composite Name:
SendEmailWithAttachmentsProj

Composite Template:
Empty Composite
Composite With BPEL
Composite With Business Rule
Composite With Human Task
Composite With Mediator
Composite From Oracle BPA Blueprint

6. In the *Create BPEL Process* window, enter the BPEL process name as "SendEmailWithAttachments" (Figure A–26). Click **OK**.

Figure A–26 Creating the BPEL Process


Create BPEL Process

BPEL Process

A BPEL process is a service orchestration, used to describe/execute a business process (or large grained service), which is implemented as a stateful service.

Name: SendEmailWithAttachments
Namespace: racle.com/SendEmailWithAttachmentsApp/SendEmailWithAttachmentsProj/SendEmailWithAttachments
Template: Asynchronous BPEL Process
Service Name: sendemailwithattachments_client
[✓] Expose as a SOAP service
Input: hAttachmentsApp/SendEmailWithAttachmentsProj/SendEmailWithAttachments}process
Output: entsApp/SendEmailWithAttachmentsProj/SendEmailWithAttachments}processResponse

7. Verify that "Expose as a SOAP service" is checked. Click OK.

8. You have now created an empty and default BPEL application.

   In the Oracle JDeveloper main window you can view the following components of the sample application under the *Composite.xml* tab.

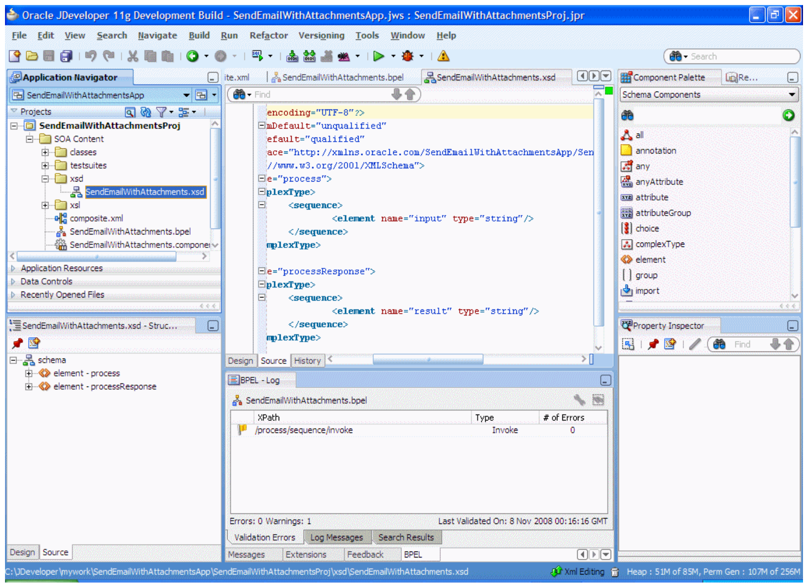
   - The left box is the definition of a Web Service client that is used to initiate an application.
   - The middle box is a BPEL process that creates and formats the message and calls the messaging service.

   ***Note*:** You must later create the messaging service resource that is used to send the message when you create the User Notification BPEL process (steps 13-19).

   ---

9. Expand the *xsd* folder in the Application Navigator and open *SendEmailWithAttachments.xsd* by double-clicking it (Figure A–27).

Figure A–27 Accessing the SendEmailWithAttachments.xsd File


Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr
File Edit View Search Navigate Build Run Refactor Versioning Tools Window Help
Application Navigator
SendEmailWithAttachmentsApp
Projects
SendEmailWithAttachmentsProj
SOA Content
classes
testsuites
xsd
SendEmailWithAttachments.xsd
xsl
composite.xml
SendEmailWithAttachments.bpel
SendEmailWithAttachments.componentType
Application Resources
Data Controls
Recently Opened Files
SendEmailWithAttachments.xsd - Structure
schema
element - process
element - processResponse
Design Source
ite.xml
SendEmailWithAttachments.bpel
SendEmailWithAttachments.xsd
Find
Design Source History
BPEL - Log
Validation Errors Log Messages Search Results
Messages Extensions Feedback BPEL
Component Palette
Schema Components
all
annotation
any
anyAttribute
attribute
attributeGroup
choice
complexType
element
group
import
Property Inspector
C:\JDeveloper\mywork\SendEmailWithAttachmentsApp\SendEmailWithAttachmentsProj\xsd\SendEmailWithAttachments.xsd
Xml Editing
Heap : 51M of 85M, Perm Gen : 107M of 256M

```
encoding="UTF-8"?>
mDefault="unqualified"
efault="qualified"
ace="http://xmlns.oracle.com/SendEmailWithAttachmentsApp/Sen
//www.w3.org/2001/XMLSchema">
e="process">
plexType>
    <sequence>
            <element name="input" type="string"/>
    </sequence>
mplexType>

e="processResponse">
plexType>
    <sequence>
            <element name="result" type="string"/>
    </sequence>
mplexType>
```

SendEmailWithAttachments.bpel

	XPath	Type	# of Errors
	/process/sequence/invoke	Invoke	0

Errors: 0 Warnings: 1 Last Validated On: 8 Nov 2008 00:16:16 GMT

10. Click on the **Source** tab (Figure A–27).
11. Perform the following modifications to the inputs of this BPEL application:

    In the generated file, SendEmailWithAttachments.xsd, in the xsd folder in the application navigator under projects, the following element definition is created by default:

```
<element name="process">
  <complexType>
    <sequence>
      <element name="input" type="string"/>
    </sequence>
  </complexType>
</element>
```

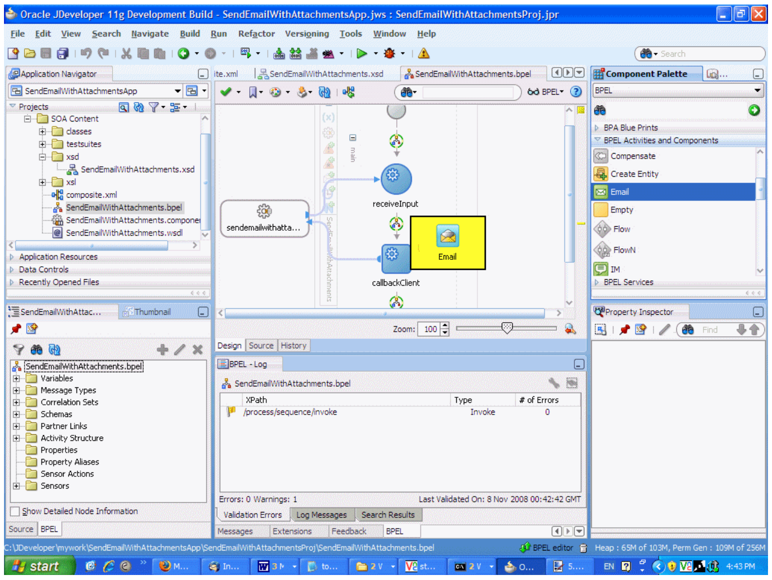
Select the *Source* tab, and replace the lines above with the following:

```
<element name="process">
```

```
<complexType>
     <sequence>
       <element name="to" type="string"/>
       <element name="subject" type="string"/>
       <element name="body" type="string"/>
       <element name="attachmentName" type="string"/>
       <element name="attachmentMimeType" type="string"/>
       <element name="attachmentURI" type="string"/>
     </sequence>
   </complexType>
 </element>
```

This xsd element defines the input for the BPEL process.

Figure A–28 Editing Email


Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr
File Edit View Search Navigate Build Run Refactor Versioning Tools Window Help
Application Navigator
SendEmailWithAttachmentsApp
Projects
SOA Content
classes
testsuites
xsd
SendEmailWithAttachments.xsd
xsl
composite.xml
SendEmailWithAttachments.bpel
SendEmailWithAttachments.componentType
SendEmailWithAttachments.wsdl
Application Resources
Data Controls
Recently Opened Files
ite.xml
SendEmailWithAttachments.xsd
SendEmailWithAttachments.bpel
sendemailwithatta...
receiveInput
Email
callbackClient
Zoom: 100
Design Source History
Component Palette
BPEL
BPA Blue Prints
BPEL Activities and Components
Compensate
Create Entity
Email
Empty
Flow
FlowN
IM
BPEL Services
Property Inspector
Find
SendEmailWithAttac...
Thumbnail
SendEmailWithAttachments.bpel
Variables
Message Types
Correlation Sets
Schemas
Partner Links
Activity Structure
Properties
Property Aliases
Sensor Actions
Sensors
Show Detailed Node Information
Source BPEL
BPEL - Log
Validation Errors Log Messages Search Results
Messages Extensions Feedback BPEL
BPEL editor
start
4:43 PM

SendEmailWithAttachments.bpel

XPath	Type	# of Errors
/process/sequence/invoke	Invoke	0

Errors: 0 Warnings: 1 Last Validated On: 8 Nov 2008 00:42:42 GMT

C:\JDeveloper\mywork\SendEmailWithAttachmentsApp\SendEmailWithAttachmentsProj\SendEmailWithAttachments.bpel

Heap : 65M of 103M, Perm Gen : 109M of 256M

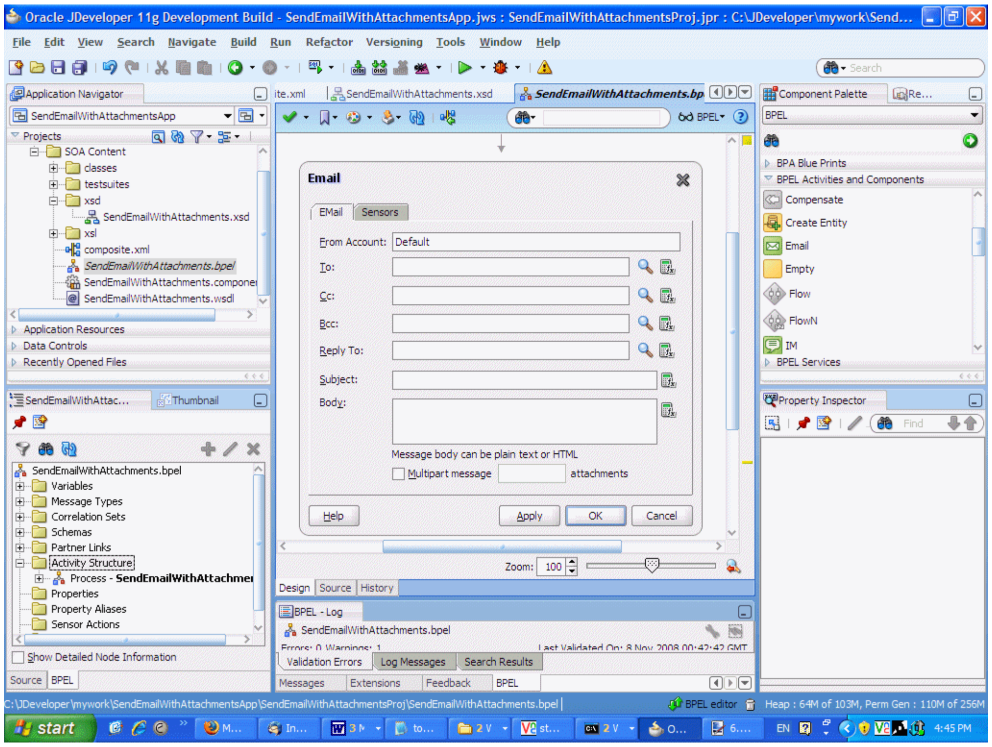
12. Save the project.

13. Select the SendEmailWithAttachments.bpel editor screen.

14. Drag and drop Email Activity from BPEL Activities and Components located in the Component Palette between the receiveInput and callbackClient activities (Figure A–28).

15. In the Edit Email window, leave the From account as “Default”.

Figure A–29 Edit Email Window



From Account: Default

Message body can be plain text or HTML

Zoom: 100

Errors: 0 Warnings: 1 Last Validated On: 8 Nov 2008 00:42:42 GMT

C:\JDeveloper\mywork\SendEmailWithAttachmentsApp\SendEmailWithAttachmentsProj\SendEmailWithAttachments.bpel

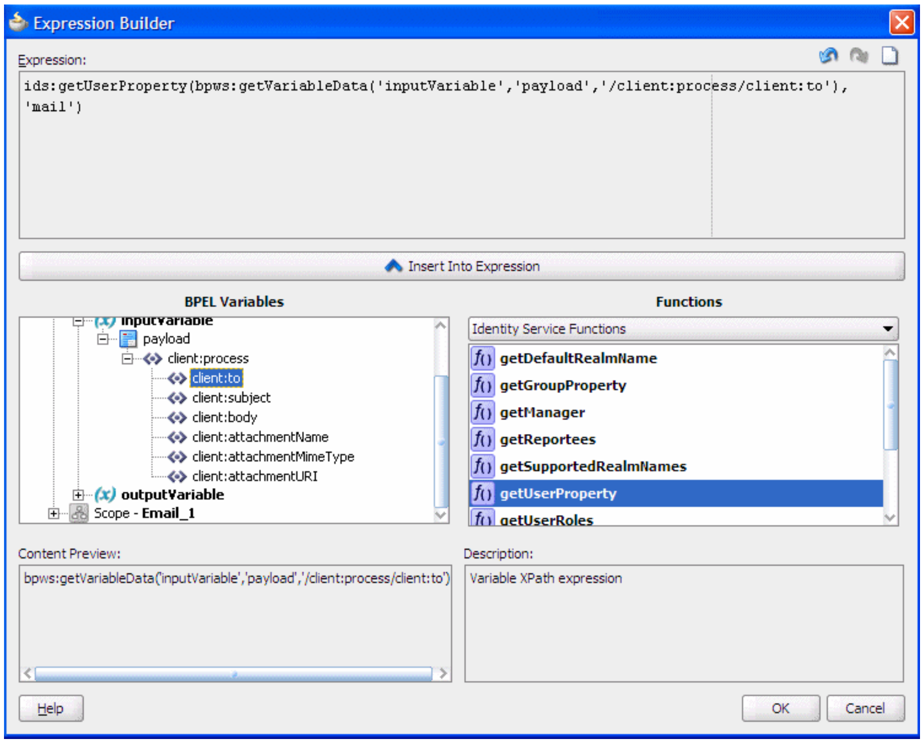
Heap : 64M of 103M, Perm Gen : 110M of 256M

16. To create the expression for To, select the Expression Builder (the second icon, Figure A–30) and perform the following steps:

- Select Identity Service Functions from the functions drop down list.
- Select the getUserProperty() function and select Insert into Expression.
- Under BPEL variables select Variables->Process->Variables->inputVariable ->payload-> client:process->client:to.
- Click Insert into Expression.
- Type the string 'mail' manually.
- Correct the parenthesis so they are matched.
- Click OK.

This expression (Figure A–30) takes the data from the Web Service and maps it to the business e-mail of the local SOA user.

Figure A–30 Expression Builder for the To Path



```
ids:getUserProperty(bpws:getVariableData('inputVariable','payload','/client:process/client:to'),
'mail')
```

Content Preview: bpws:getVariableData('inputVariable','payload','/client:process/client:to')

Description: Variable XPath expression

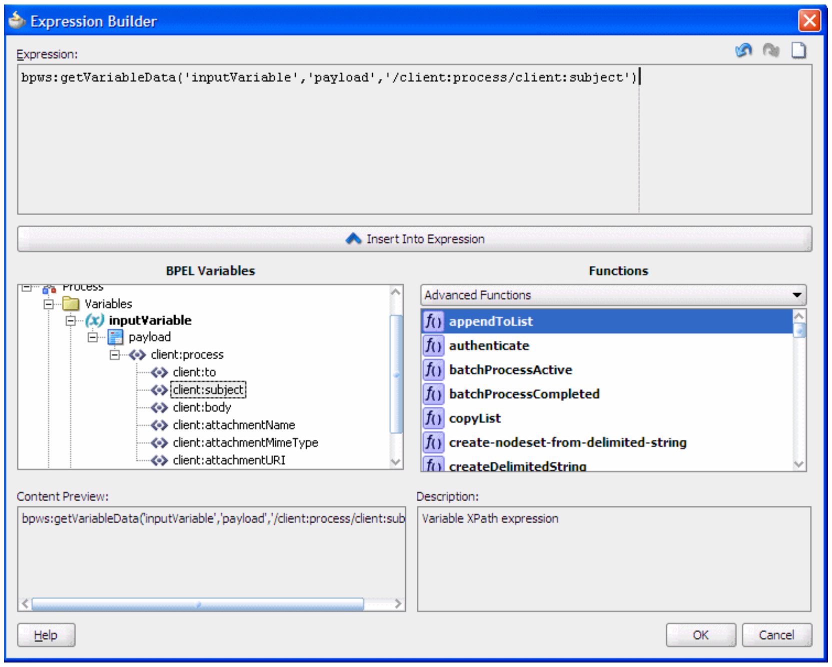
The expression must appear as follows:

```
ids:getUserProperty(  bpws:getVariableData('inputVariable',
'payload', '/client:process/client:to'),
'mail')
```

17. For Subject, select the Expression builder. Select getVariableData from Functions and click Insert Into Expression.

Figure A–31 shows the Expression Builder for the Subject.

Figure A–31 Expression Builder for the Subject



Expression:

```
bpws:getVariableData('inputVariable','payload','/client:process/client:subject')
```

BPEL Variables: Process > Variables > inputVariable > payload > client:process > client:to, client:subject, client:body, client:attachmentName, client:attachmentMimeType, client:attachmentURI

Functions: appendToList, authenticate, batchProcessActive, batchProcessCompleted, copyList, create-nodeset-from-delimited-string, createDelimitedString

Content Preview: bpws:getVariableData('inputVariable','payload','/client:process/client:sub

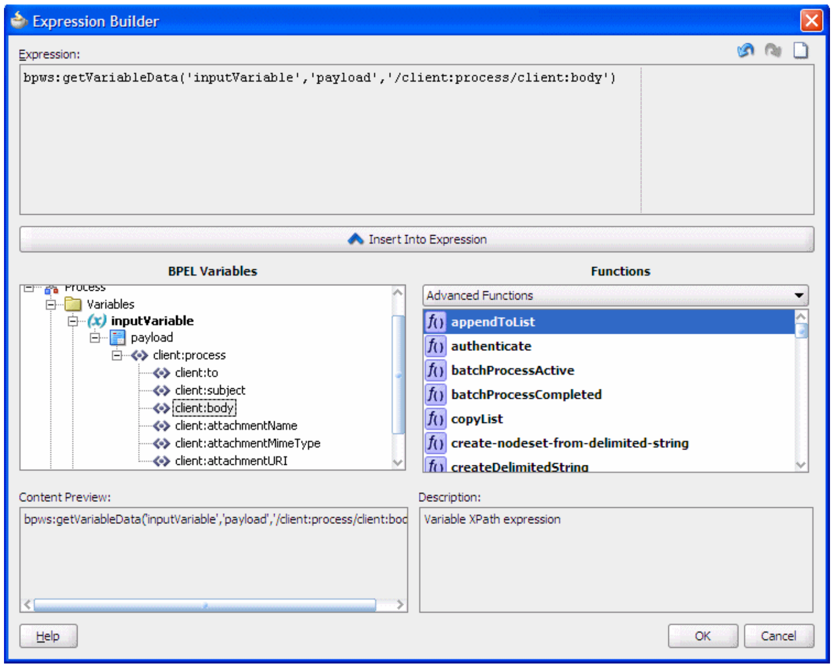
Description: Variable XPath expression

The expression must appear as follows:

```
bpws:getVariableData( 'inputVariable', 'payload',
'/client:process/
client:subject')
```

18. For “Body” select the Expression Builder and set the expression as shown in Figure A–32.

Figure A–32 Expression Builder for the Body



Expression:

```
bpws:getVariableData('inputVariable','payload','/client:process/client:body')
```

BPEL Variables: Process > Variables > inputVariable > payload > client:process > client:to, client:subject, client:body, client:attachmentName, client:attachmentMimeType, client:attachmentURI

Functions: appendToList, authenticate, batchProcessActive, batchProcessCompleted, copyList, create-nodeset-from-delimited-string, createDelimitedString

Content Preview: bpws:getVariableData('inputVariable','payload','/client:process/client:boc

Description: Variable XPath expression

The expression must appear as follows:

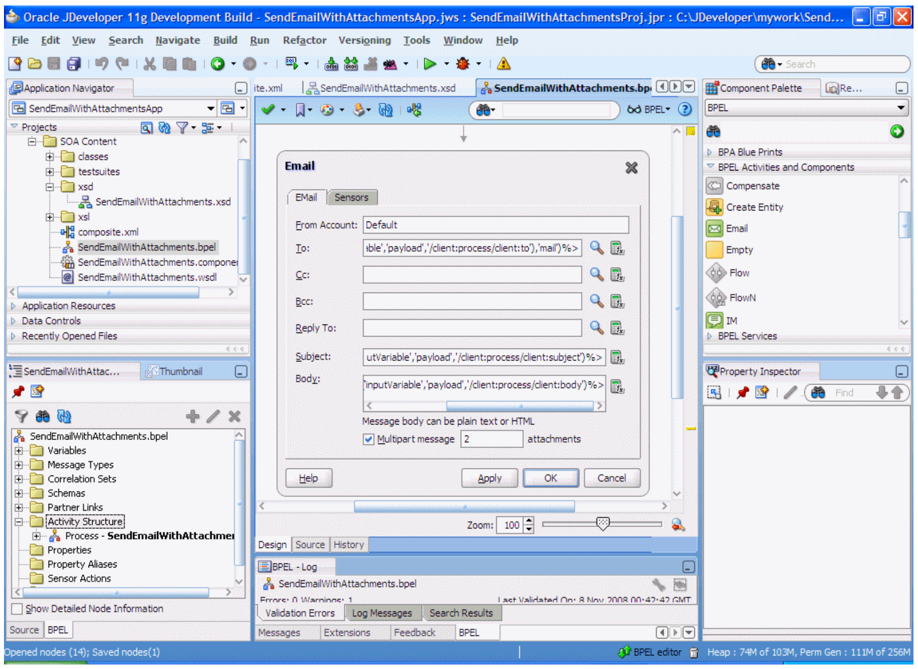
```
bpws:getVariableData('inputVariable','payload','/client:process/client:body')
```

19. In the Edit Email window (Figure A–33), ensure that the Multipart Message with attachments box is checked.

When an e-mail has multiple parts, the attachment count includes the body that is set with the Wizard above. The body specified by the Wizard above is set as the first body part.

For example, to represent a multipart mail with one (1) attached file, enter “2” as the number of body parts. When there is one attachment, enter '1' as the number of body parts.

Figure A–33 Edit Email Window



Email

From Account: Default

To: ıble','payload','/client:process/client:to'),'mail')%>

Cc:

Bcc:

Reply To:

Subject: utVariable','payload','/client:process/client:subject')%>

Body: 'inputVariable','payload','/client:process/client:body')%>

Message body can be plain text or HTML

[✓] Multipart message 2 attachments

Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr : C:\JDeveloper\mywork\Send...
EMail
Sensors
Help
Apply
OK
Cancel
Design
Source
History
BPEL - Log
Component Palette
Property Inspector

20. Set the attachments:

Each body part has three attributes: MimeType, BodyPartName and ContentBody. By default, the wizard generates default names, MIME types and contents for each of the attachments.

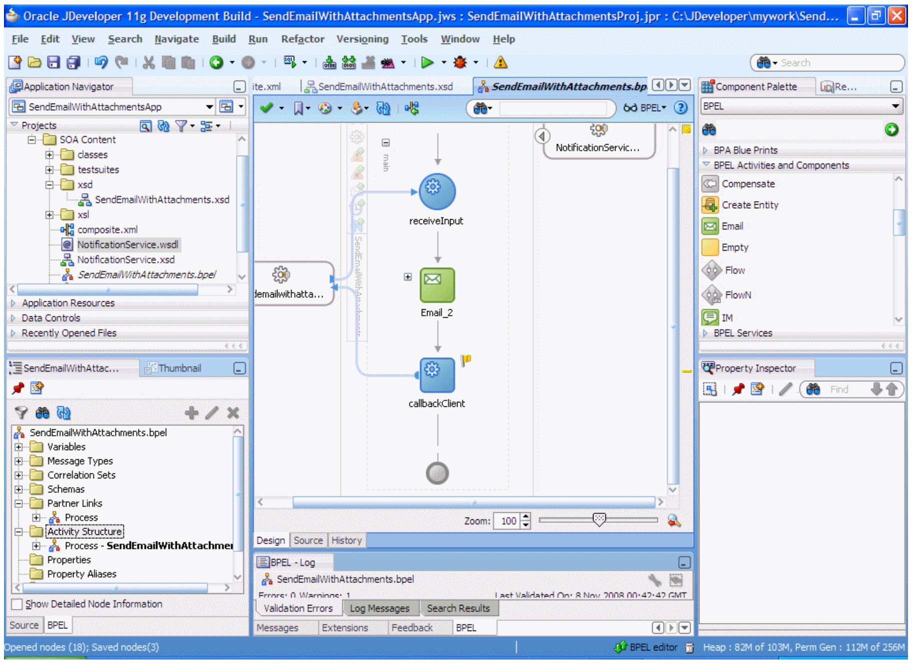
The assignment of these body parts has to be changed to set the correct data by modifying the copy rules in the assign activity in the notification scope. The copy rules (specified in the Copy Operation tab) are grouped for each assignment in the following order (the copy-to constructs are also listed):

```
MimeType - <to variable="varNotificationReq" part="EmailPayload"

query="/EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/
ns1:MimeType"/>

Name - <to variable="varNotificationReq" part="EmailPayload"

query="/EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/
ns1:BodyPartName"/>

Contents - <to variable="varNotificationReq" part="EmailPayload"
```

```
query="/EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/
ns1:ContentBody"/>
```

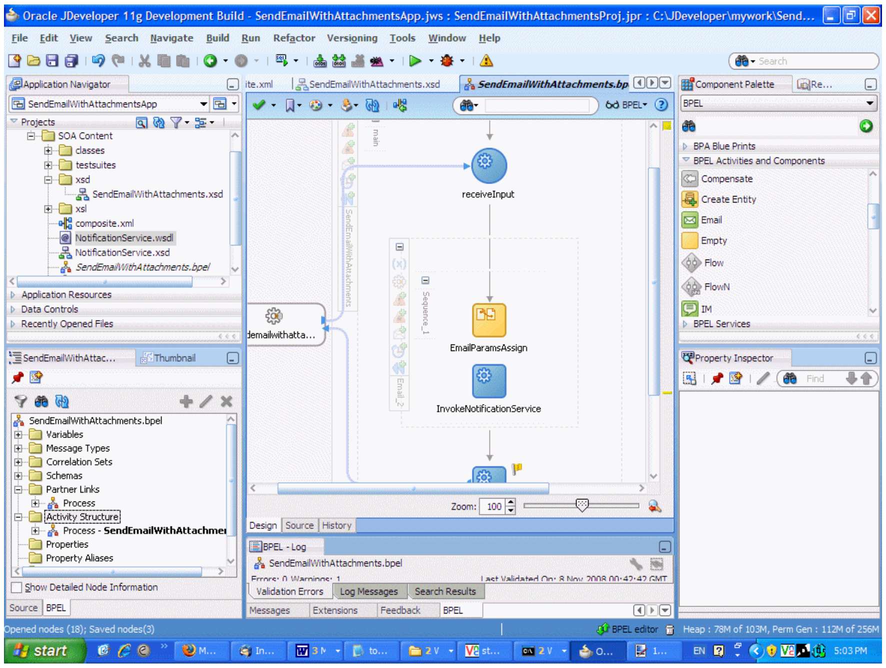
1. Expand the Email Node by selecting the plus sign icon (Figure A–34).

Figure A–34 Expanding the Email Node


Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr : C:\JDeveloper\mywork\Send...
File Edit View Search Navigate Build Run Refactor Versioning Tools Window Help
Application Navigator
SendEmailWithAttachmentsApp
Projects
SOA Content
classes
testsuites
xsd
SendEmailWithAttachments.xsd
xsl
composite.xml
NotificationService.wsdl
NotificationService.xsd
SendEmailWithAttachments.bpel
Application Resources
Data Controls
Recently Opened Files
receiveInput
Email_2
callbackClient
NotificationServic...
Zoom: 100
Design Source History
Component Palette
BPEL
BPA Blue Prints
BPEL Activities and Components
Compensate
Create Entity
Email
Empty
Flow
FlowN
IM
BPEL Services
Property Inspector
SendEmailWithAttachments.bpel
Variables
Message Types
Correlation Sets
Schemas
Partner Links
Process
Activity Structure
Process - SendEmailWithAttachmen
Properties
Property Aliases
Show Detailed Node Information
BPEL - Log
Errors: 0 Warnings: 1
Last Validated On: 8 Nov 2008 00:42:42 GMT
Validation Errors Log Messages Search Results
Messages Extensions Feedback BPEL
Opened nodes (18); Saved nodes(3)
BPEL editor Heap : 82M of 103M, Perm Gen : 112M of 256M

2. Double-click the EmailParamAssign node (Figure A–35).

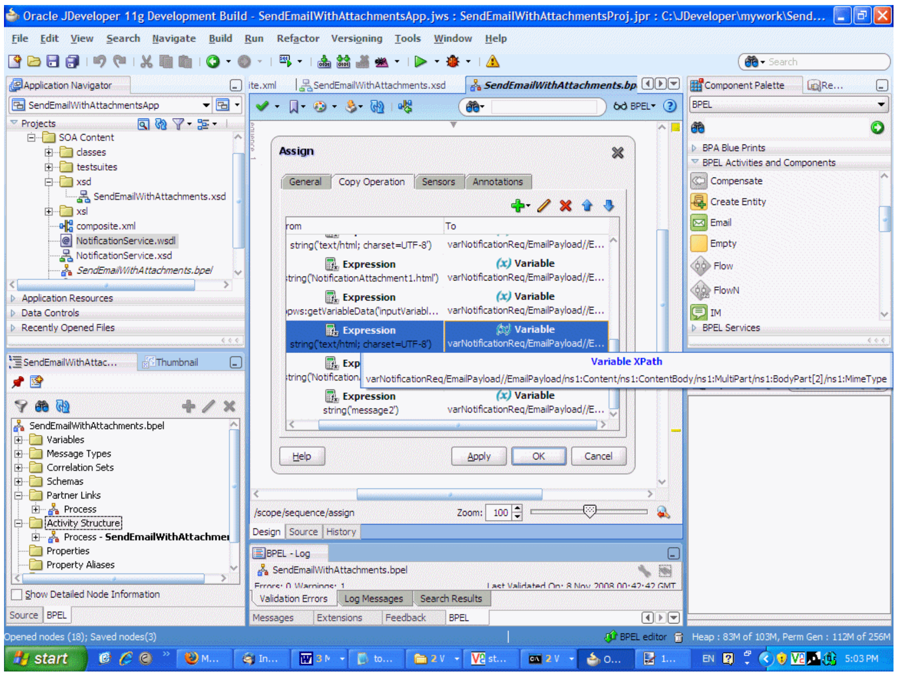
Figure A–35 Email ParamAssign Node


Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr : C:\JDeveloper\mywork\Send...
File Edit View Search Navigate Build Run Refactor Versioning Tools Window Help
Application Navigator
receiveInput
EmailParamsAssign
InvokeNotificationService
Sequence_1
Email_2
Zoom: 100
Component Palette
Property Inspector
BPEL - Log
Errors: 0 Warnings: 1
Last Validated On: 8 Nov 2008 00:42:42 GMT
Opened nodes (18); Saved nodes(3)
BPEL editor Heap : 78M of 103M, Perm Gen : 112M of 256M
start
5:03 PM

When making changes in the EmailParamAssign node (for example, editing the XPath variables), perform a Save All from the File menu after making each change. This ensures that the changes are reflected in the .bpel file.

3. To edit the mimeType of the second body part (the first body part is the contents set in the wizard) select the second body part variable ending with "MimeType" by double-clicking it (Figure A–36).

Figure A–36 Editing the mimeType of the Second Body Part



Oracle JDeveloper 11g Development Build - SendEmailWithAttachmentsApp.jws : SendEmailWithAttachmentsProj.jpr : C:\JDeveloper\mywork\Send...

Assign

From	To
string('text/html; charset=UTF-8')	varNotificationReq/EmailPayload//E...
Expression string('NotificationAttachment1.html')	Variable varNotificationReq/EmailPayload//E...
Expression bpws:getVariableData('inputVariabl...	Variable varNotificationReq/EmailPayload//E...
Expression string('text/html; charset=UTF-8')	Variable varNotificationReq/EmailPayload//E...
Exp string('Notification	
Expression string('message2')	Variable varNotificationReq/EmailPayload//E...

Variable XPath

varNotificationReq/EmailPayload//EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/ns1:MimeType

/scope/sequence/assign

Zoom: 100

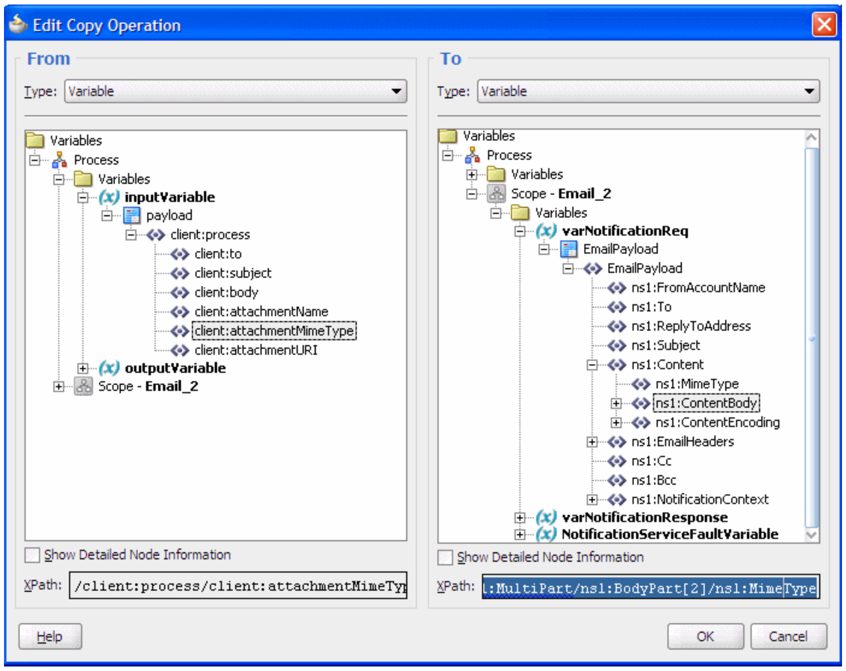
Opened nodes (18); Saved nodes(3)

Heap : 83M of 103M, Perm Gen : 112M of 256M

4. Edit the XPath as shown below (Figure A–37):

```
From: /client:process/client:attachmentMimeType,

To: /EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/
ns1:BodyPart[2]/ns1:MimeType
```

Figure A–37 Editing the XPath for mimeType



From:

```
Variables
  Process
    Variables
      (x) inputVariable
        payload
          client:process
            client:to
            client:subject
            client:body
            client:attachmentName
            client:attachmentMimeType
            client:attachmentURI
      (x) outputVariable
    Scope - Email_2
```

Show Detailed Node Information

XPath: /client:process/client:attachmentMimeTy

To:

```
Variables
  Process
    Variables
    Scope - Email_2
      Variables
        (x) varNotificationReq
          EmailPayload
            EmailPayload
              ns1:FromAccountName
              ns1:To
              ns1:ReplyToAddress
              ns1:Subject
              ns1:Content
                ns1:MimeType
                ns1:ContentBody
                ns1:ContentEncoding
              ns1:EmailHeaders
              ns1:Cc
              ns1:Bcc
              ns1:NotificationContext
        (x) varNotificationResponse
        (x) NotificationServiceFaultVariable
```

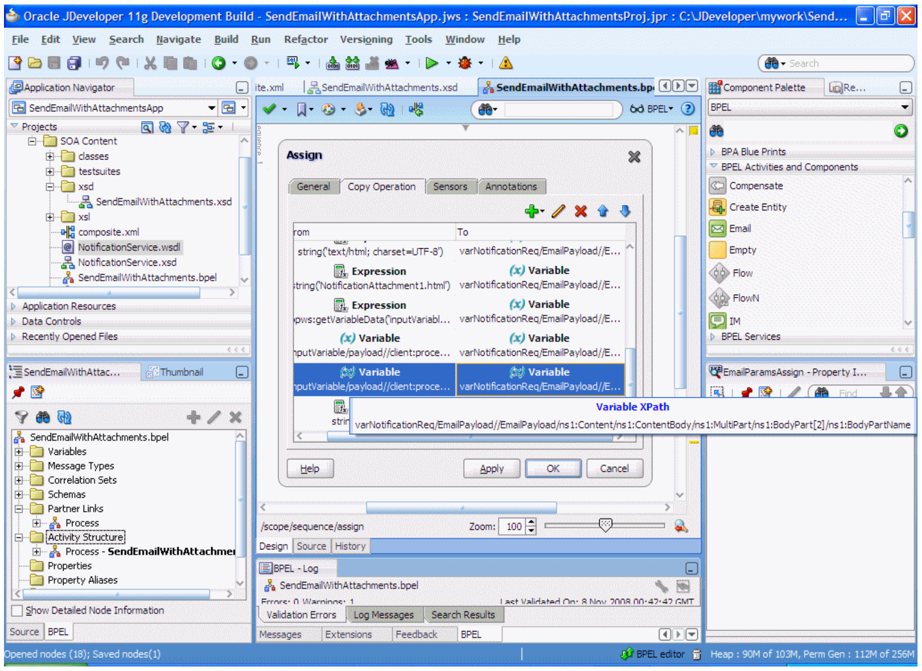
Show Detailed Node Information

XPath: :MultiPart/ns1:BodyPart[2]/ns1:MimeType

5. Save the project.

6. To edit the attachment name for the second attachment, select the second body part variable ending with "BodyPartName" by double-clicking it (Figure A–38).

Figure A–38 Editing the Attachment Name for the Second Attachment



From	To
string('text/html; charset=UTF-8')	varNotificationReq/EmailPayload//E...
Expression	(x) Variable
tring('NotificationAttachment1.html')	varNotificationReq/EmailPayload//E...
Expression	(x) Variable
ows:getVariableData('inputVariabl...	varNotificationReq/EmailPayload//E...
(x) Variable	(x) Variable
putVariable/payload//client:proce...	varNotificationReq/EmailPayload//E...
(x) Variable	(x) Variable
putVariable/payload//client:proce...	varNotificationReq/EmailPayload//E...

Variable XPath

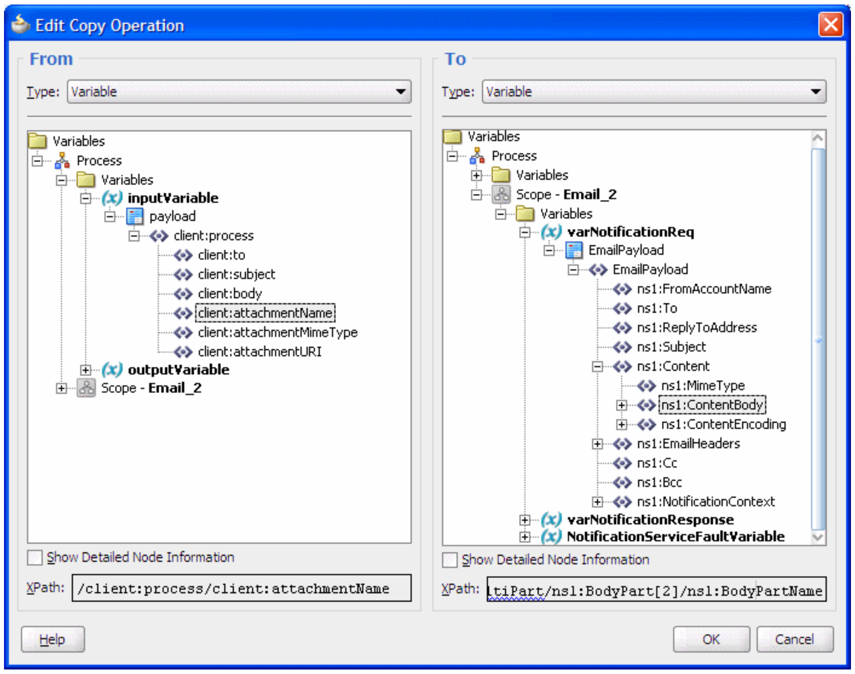
varNotificationReq/EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/ns1:BodyPartName

7. Edit the XPath as shown below:

```
From: /client:process/client:attachmentName
```

```
To: /EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart
/ns1:BodyPart[2]/ns1:BodyPartName
```

Figure A–39   Editing the XPath for BodyPartName



From — Type: Variable

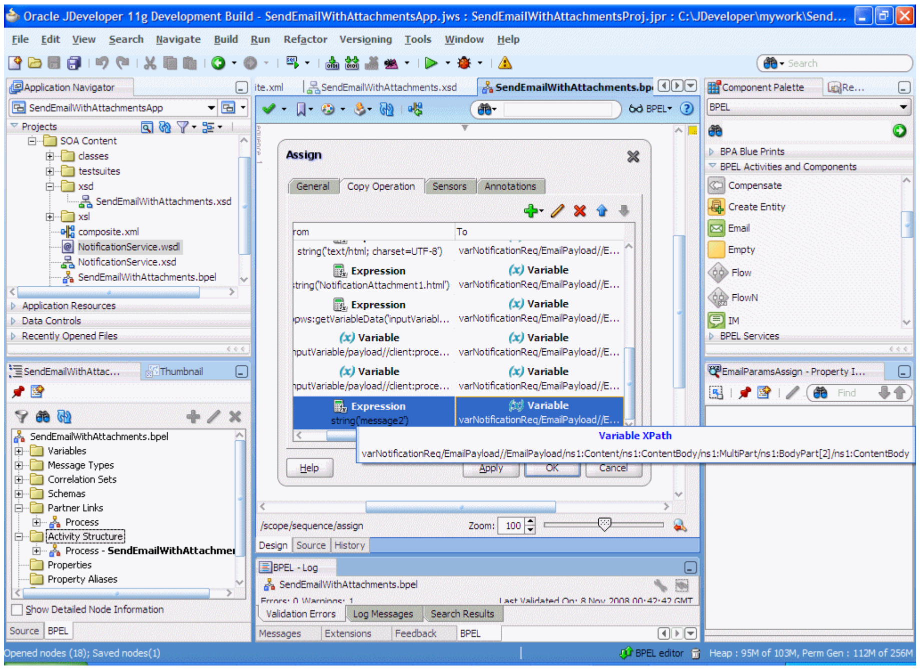
XPath: `/client:process/client:attachmentName`

To — Type: Variable

XPath: `tiPart/ns1:BodyPart[2]/ns1:BodyPartName`

8. Save the project.
9. To edit the attachment contents of the second attachment, select the second body part variable ending with "ContentBody" by double-clicking it (Figure A–40).

Figure A–40   Editing the Attachment Contents of the Second Attachment



From	To
string('text/html; charset=UTF-8')	varNotificationReq/EmailPayload//E...
Expression string('NotificationAttachment1.html')	Variable varNotificationReq/EmailPayload//E...
Expression bpws:getVariableData('inputVariabl...	Variable varNotificationReq/EmailPayload//E...
Variable inputVariable/payload//client:proce...	Variable varNotificationReq/EmailPayload//E...
Variable inputVariable/payload//client:proce...	Variable varNotificationReq/EmailPayload//E...
Expression string('message2')	Variable varNotificationReq/EmailPayload//E...

Variable XPath
varNotificationReq/EmailPayload//EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/ns1:BodyPart[2]/ns1:ContentBody

/scope/sequence/assign   Zoom: 100

SendEmailWithAttachments.bpel
Errors: 0 Warnings: 1   Last Validated On: 8 Nov 2008 00:42:42 GMT

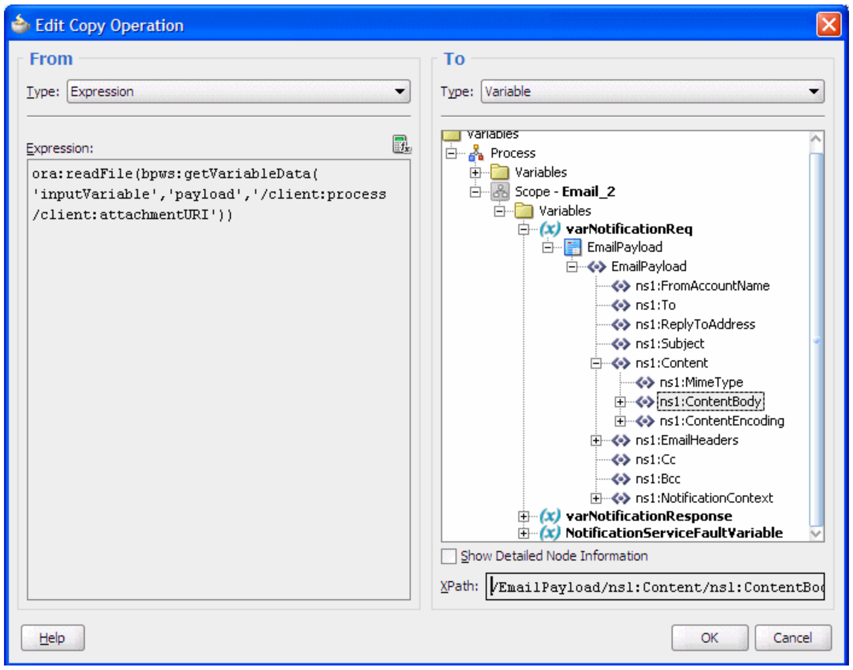
Opened nodes (18); Saved nodes(1)   BPEL editor   Heap : 95M of 103M, Perm Gen : 112M of 256M

10. Edit the XPath as shown below (Figure A–41):

```
From: ora:readFile(bpws:getVariableData('inputVariable','payload','/client:
process/client:attachmentURI'))

To:
/EmailPayload/ns1:Content/ns1:ContentBody/ns1:MultiPart/
ns1:BodyPart[2]/ns1:ContentBody
```

ora:readFile() xpath function is available under “BPEL Xpath Extension Functions”.

Figure A–41 Editing the XPath from the ContentBody


Edit Copy Operation

From
Type: Expression

Expression:
```
ora:readFile(bpws:getVariableData(
'inputVariable','payload','/client:process
/client:attachmentURI'))
```

To
Type: Variable

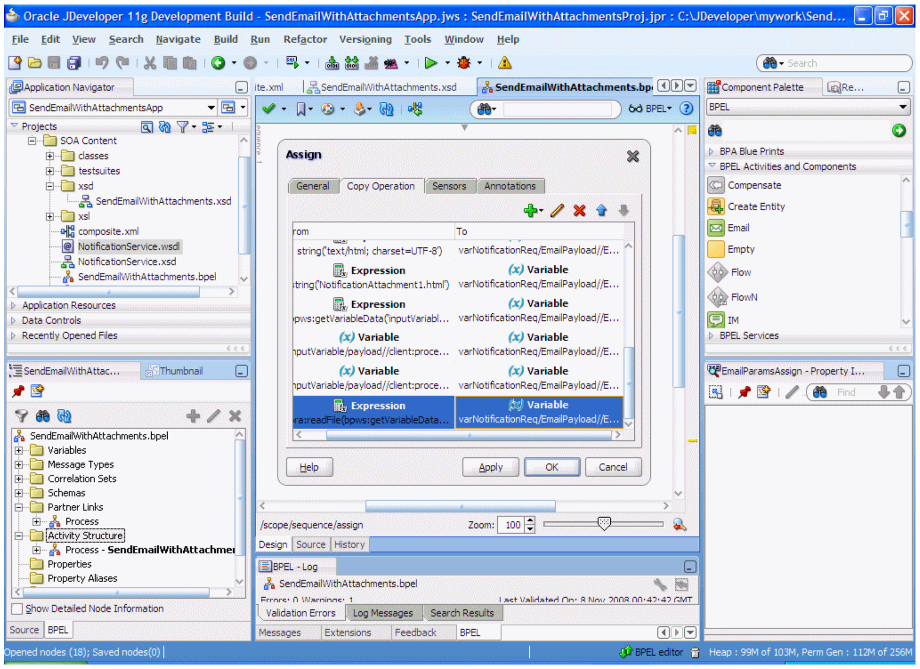
Variables
Process
Variables
Scope - Email_2
Variables
varNotificationReq
EmailPayload
EmailPayload
ns1:FromAccountName
ns1:To
ns1:ReplyToAddress
ns1:Subject
ns1:Content
ns1:MimeType
ns1:ContentBody
ns1:ContentEncoding
ns1:EmailHeaders
ns1:Cc
ns1:Bcc
ns1:NotificationContext
varNotificationResponse
NotificationServiceFaultVariable

Show Detailed Node Information

XPath: /EmailPayload/ns1:Content/ns1:ContentBo

11. Click OK in the Edit Copy Operation screen.

Figure A–42 Copy Operations Tab



rom	To
string('text/html; charset=UTF-8')	varNotificationReq/EmailPayload//E...
Expression string('NotificationAttachment1.html')	Variable varNotificationReq/EmailPayload//E...
Expression bpws:getVariableData('inputVariabl...	Variable varNotificationReq/EmailPayload//E...
Variable inputVariable/payload//client:proce...	Variable varNotificationReq/EmailPayload//E...
Variable inputVariable/payload//client:proce...	Variable varNotificationReq/EmailPayload//E...
Expression ora:readFile(bpws:getVariableData...	Variable varNotificationReq/EmailPayload//E...

12. Click OK in the assign activity. Save the project.

 The Process Modeling procedure is complete. You can use the information in this procedure to add notification with attachments to your SOA composite application.

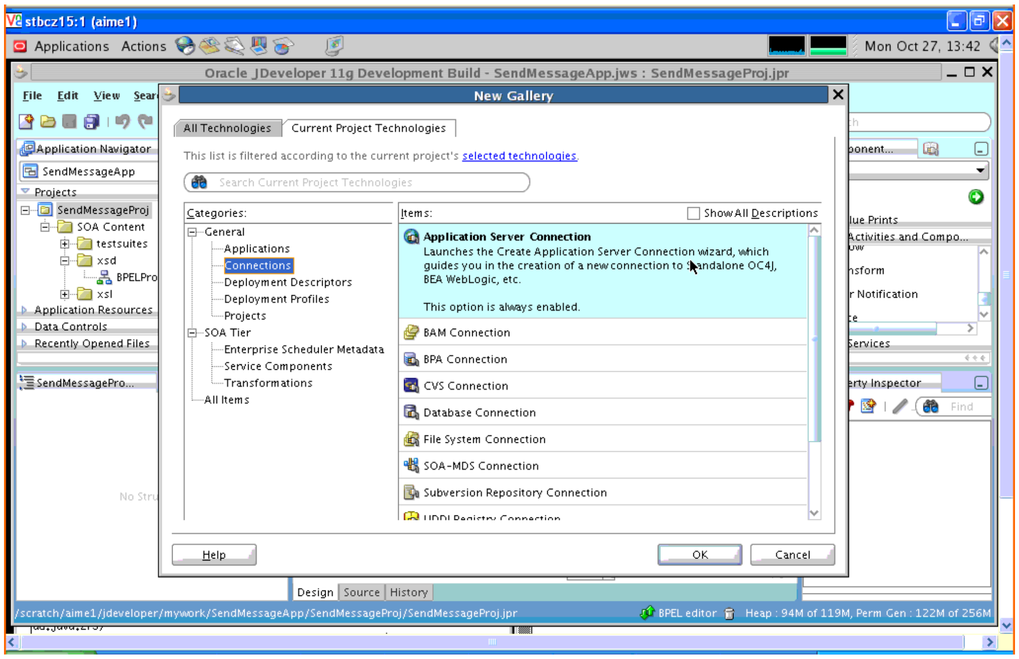
 You can now deploy and run the application as described in Section A.2.3, "Running the Pre-Built Sample".

A.2.6 Creating a New Application Server Connection

Perform the following steps to create a new Application Server Connection.

1. Create a new Application Server Connection by right-clicking the project and selecting **New**, **Connections**, and **Application Server Connection** (Figure A–43).

Figure A–43 New Application Server Connection



This list is filtered according to the current project's selected technologies.

Application Server Connection
Launches the Create Application Server Connection wizard, which guides you in the creation of a new connection to standalone OC4J, BEA WebLogic, etc.

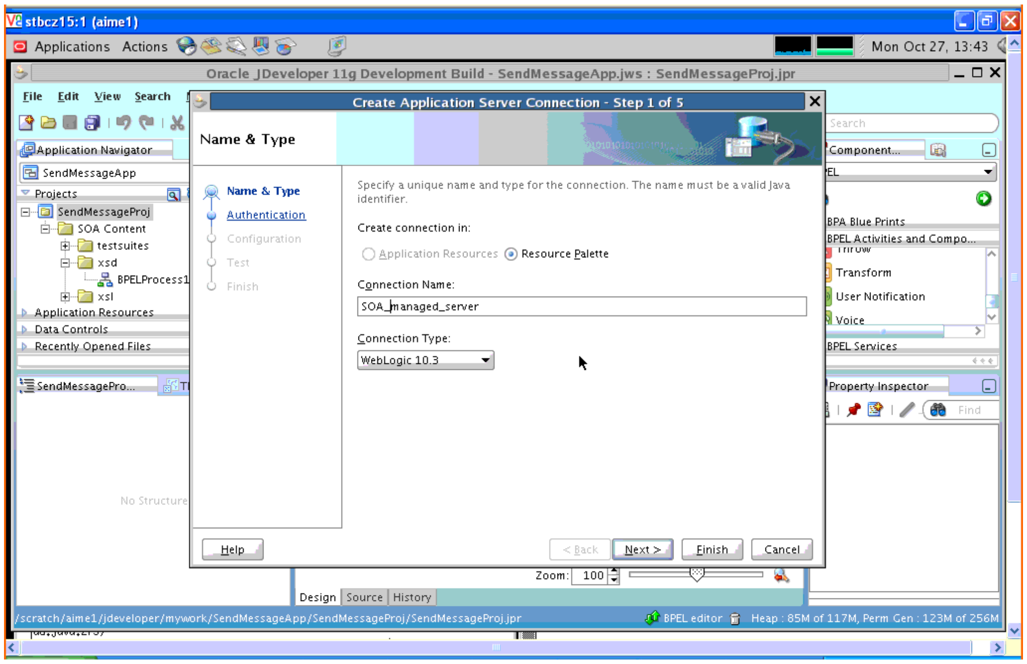
This option is always enabled.

/scratch/aime1/jdeveloper/mywork/SendMessageApp/SendMessageProj/SendMessageProj.jpr BPEL editor Heap : 94M of 119M, Perm Gen : 122M of 256M

2. Name the connection “SOA_server” and click **Next** (Figure A–44).

3. Select "WebLogic 10.3" as the *Connection Type*.

Figure A–44 New Application Server Connection



Specify a unique name and type for the connection. The name must be a valid Java identifier.

Create connection in:

( ) Application Resources [✓ Resource Palette]

Connection Name: SOA_managed_server

Connection Type: WebLogic 10.3

/scratch/aime1/jdeveloper/mywork/SendMessageApp/SendMessageProj/SendMessageProj.jpr BPEL editor Heap : 85M of 117M, Perm Gen : 123M of 256M

4. Enter the authentication information. The typical values are:

   Username: `weblogic`
   Password: `weblogic`

5. On the *Connection* screen, enter the hostname, port and SSL port for the SOA admin server, and enter the name of the domain for WLS Domain.

6. Click **Next**.
7. On the *Test* screen click **Test Connection**.
8. Verify that the message “Success!” appears.

   The Application Server Connection has been created.

B Profile Service Provider Configuration Reference (profile.xml)

The following chapter provides a complete reference to the profile provider configuration file, `profile.xml`, in the following sections:

- Section B.1, "Overview of profile.xml"
- Section B.2, "Graphical Representation"
- Section B.3, "Editing profile.xml"
- Section B.4, "XML Schema"
- Section B.5, "Example profile.xml File"
- Section B.6, "XML Element Description"

B.1 Overview of profile.xml

The `profile.xml` file configures attributes of a profile service provider, such as:

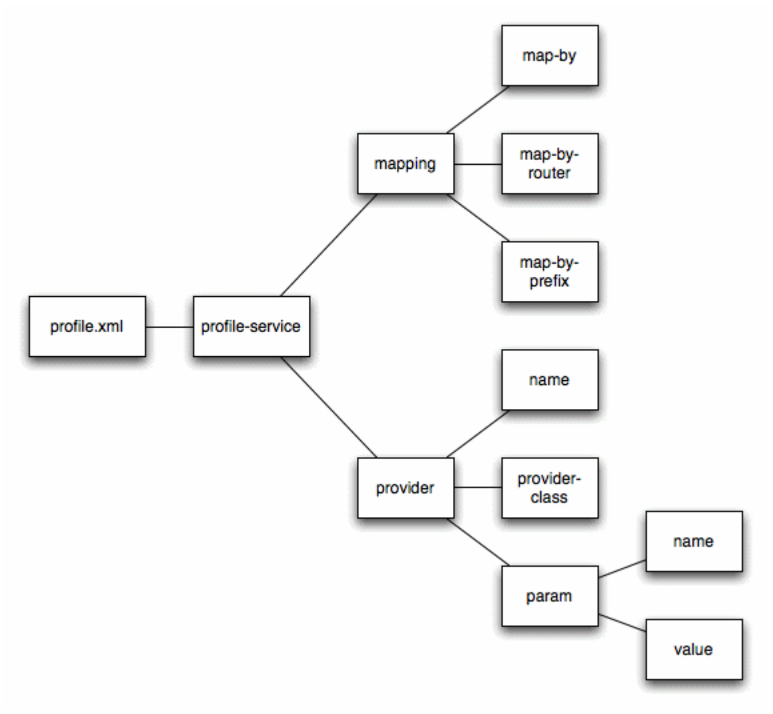
- The name of the provider
- The class name of the provider implementation
- Optional arguments passed to the provider
- Mapping rules for using the provider.

`profile.xml` is stored in the *`DOMAIN_DIR`*`/config/custom` subdirectory where *`DOMAIN_DIR`* is the root directory of the OWLCS domain.

B.2 Graphical Representation

Figure B–1 shows the element hierarchy of the `profile.xml` file.

Figure B–1 Element Hierarchy of profile.xml



B.3 Editing profile.xml

Oracle recommends using the Administration Console profile service extension to modify `profile.xml` indirectly, rather than editing the file by hand. Using the Administration Console ensures that the `profile.xml` document always contains valid XML. See Configuring Profile Providers Using the Administration Console in Developing Applications with OWLCS.

You may need to manually view or edit `profile.xml` to troubleshoot problem configurations, repair corrupted files, or to roll out custom profile provider configurations to a large number of machines when installing or upgrading OWLCS. When you manually edit `profile.xml`, you must reboot servers to apply your changes.

B.3.1 Steps for Editing profile.xml

If you need to modify `profile.xml` on a production system, follow these steps:

1. Use a text editor to open the *`DOMAIN_DIR`*`/config/custom/profile.xml` *`file, where DOMAIN_DIR`* is the root directory of the OWLCS domain.
2. Modify the `profile.xml` file as necessary. See "XML Element Description" for a full description of the XML elements.
3. Save your changes and exit the text editor.
4. Reboot or start servers to have your changes take effect:
5. Test the updated system to validate the configuration.

B.4 XML Schema

The full schema for the `profile.xml` file is bundled within the `profile-service-descriptor-binding.jar` library, installed in the *WLSS_HOME*`/server/lib/wlss` directory.

B.5 Example profile.xml File

See Developing Custom Profile Providers in *Developing SIP Applications* for sample listings of `profile.xml` configuration files.

B.6 XML Element Description

The following sections describe each XML element in `profile.xml`.

B.6.1 profile-service

The top level `profile-service` element contains the entire profile service configuration.

B.6.2 mapping

Specifies how requests for profile data are mapped to profile provider implementations.

B.6.2.1 map-by

Specifies the technique used for mapping documents to providers:

- `router` uses a custom router class, specified by `map-by-router`, to determine the provider.
- `prefix` uses the specified `map-by-prefix` entry to map documents to a provider.
- `provider-name` uses the specified `name` element in the `provider` entry to map documents to a provider.

B.6.2.2 map-by-prefix

Specifies the prefix used to map documents to profile providers when mapping by prefix.

B.6.2.3 map-by-router

Specifies the router class (implementing `com.bea.wcp.profile.ProfileRouter`) used to map documents to profile providers with router-based mapping.

B.6.3 provider

Configures the profile provider implementation and startup options.

B.6.3.1 name

Specifies a name for the provider configuration. The `name` element is also used for mapping documents to the provider if you specify the `provider-name` mapping technique.

B.6.3.2 provider-class

Specifies the profile provider class (implementing `com.bea.wcp.profile.ProfileServiceSpi`).

B.6.3.3 param

Uses the `name` and `value` elements to specify optional parameters to the provider implementation.

C

Developing SIP Servlets Using Eclipse

The following chapter describes how to use Eclipse to develop SIP Servlets for use with OWLCS, in the following sections:

- Section C.1, "Overview"
- Section C.2, "Setting Up the Development Environment"
- Section C.3, "Building and Deploying the Project"
- Section C.4, "Debugging SIP Servlets"

C.1 Overview

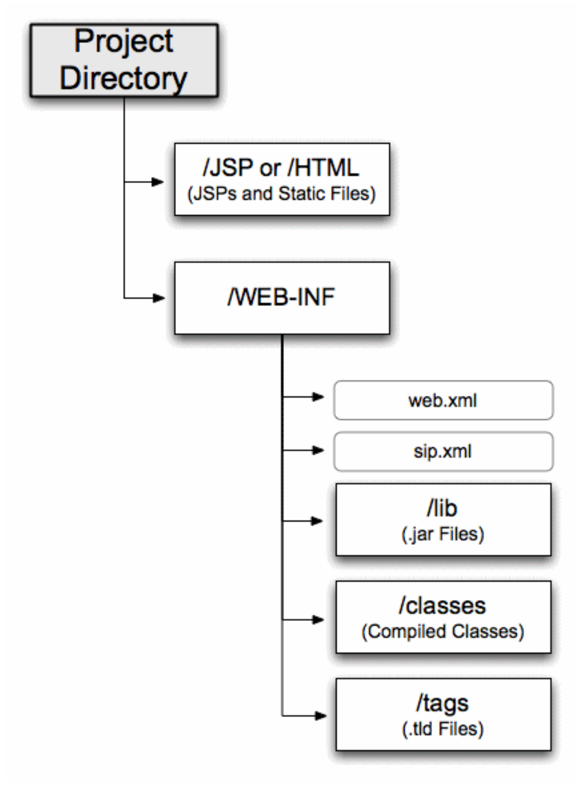
This document provides detailed instructions for using the Eclipse IDE as a tool for developing and deploying SIP Servlets with OWLCS. The full development environment requires the following components, which you must obtain and install before proceeding:

- OWLCS
- JDK 1.6.05
- Eclipse version 3.4 or Eclipse 3.3 Europe. This includes a CVS client and server (required only for version control).

C.1.1 SIP Servlet Organization

Building a SIP Servlet produces a Web Archive (WAR file or directory) as an end product. A basic SIP Servlet WAR file contains the subdirectories and contents described in Figure C–1.

Figure C–1 SIP Servlet WAR Contents



C.2 Setting Up the Development Environment

Follow these steps to set up the development environment for a new SIP Servlet project:

1. Create a new OWLCS Domain.
2. Create a new Eclipse project.

The sections that follow describe each step in detail.

C.2.1 Creating a OWLCS Domain

In order to deploy and test your SIP Servlet, you need access to a OWLCS domain that you can reconfigure and restart as necessary. Follow the instructions in *Oracle WebLogic Communication Services Installation Guide* to create a new domain using the Configuration Wizard. When generating a new domain:

- Select Development Mode as the startup mode for the new domain.
- Select Sun SDK 1.6.05 as the SDK for the new domain.

C.2.2 Verifying the Default Eclipse JVM

Eclipse 3.4 uses the required version Java 6 (1.6) by default. Follow these steps to verify the configured JVM:

1. Start Eclipse.
2. Select **Window** >**Preferences**.
3. Expand the Java category in the left pane, and select Installed JREs.

4. Verify that Java 6 (1.6) is configured. If it is, proceed to step 10.
5. If not configured correctly, click Add... to add a new JRE.
6. Enter a name to use for the new JRE in the JRE name field.
7. Click the Browse... button next to the JRE home directory field. Then navigate to the `MIDDLEWARE_HOME/jdk160_05` directory and click OK.
8. Click **OK** to add the new JRE.
9. Select the check box next to the new JRE to make it the default.
10. Click **OK** to dismiss the preferences dialog.

C.2.3 Creating a New Eclipse Project

Follow these steps to create a new Eclipse project for your SIP Servlet development, adding the OWLCS libraries required for building and deploying the application:

1. Start Eclipse.
2. Select **File** > **New** > **Project**...
3. Select Java Project and click **Next**.
4. Enter a name for your project in the Project Name field.
5. In the Location field, select "Create project in workspace" if you have not yet begun writing the SIP Servlet code. If you already have source code available in another location, select "Create project at external location" and specify the directory. Click Next.
6. Click the Libraries tab and follow these steps to add required JARs to your project:
   a. Click **Add External JARs**...
   b. Use the JAR selection dialog to add the `MIDDLEWARE_HOME/server/lib/weblogic.jar` file to your project.
   c. Repeat the process to add the `MIDDLEWARE_HOME/server/lib/wlss/sipservlet.jar` and `MIDDLEWARE_HOME/server/lib/wlss/wlssapi.jar` files to your project.
7. Add any additional JAR files that you may require for your project.
8. To enable deploying directly from eclipse, change the build folder from /src/build to /src/WebContent/WEB-INF/classes. This means that you do not have to package the application before deploying it.
9. Click **Finish** to create the new project. Eclipse displays your new project name in the Package Explorer.
10. Right-click on the name of your project and use the **New** >**Folder** command to recreate the directory structure shown in Figure C–1, "SIP Servlet WAR Contents".

C.3 Building and Deploying the Project

The `build.xml` file that you created compiles your code, packages the WAR, and copies the WAR file to the `/applications` subdirectory of your development domain. OWLCS automatically deploys valid applications located in the `/applications` subdirectory.

C.4 Debugging SIP Servlets

In order to debug SIP Servlets, you must enable certain debug options when you start OWLCS. Follow these steps to add the required debug options to the script used to start OWLCS, and to attachthe debugger from within Eclipse:

Note: On Linux, debug is enabled by default if you install in developer mode. However, the port is set to 8453.

---

1. Use a text editor to open the `StartWebLogic.cmd` script for your development domain.
2. Beneath the line that reads:

   ```
   set JAVA_OPTIONS=
   Enter the following line:
   set DEBUG_OPTS=-Xdebug -Xrunjdwp:transport=dt_
   socket,address=9000,server=y,suspend=n
   ```

3. In the last line of the file, add the `%DEBUG_OPTS%` variable in the place indicated below:

   ```
   "%JAVA_HOME%\bin\java" %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% %DEBUG_OPTS%
   -Dweblogic.Name=%SERVER_NAME% -Dweblogic.management.username=%WLS_USER%
   -Dweblogic.management.password=%WLS_PW%
   -Dweblogic.management.server=%ADMIN_URL%
   -Djava.security.policy="%MIDDLEWARE_HOME%\server\lib\weblogic.policy"
   weblogic.Server
   ```

4. Save the file and use the script to restart OWLCS.
5. To attach the debugger from within Eclipse select the **Run** > **Open debug** dialog.
6. Create a new "Remote Java Application".
7. Enter the host and port corresponding to the DEBUG_OPTS.

D

Porting Existing Applications to Oracle WebLogic Communication Services

This chapter describes guidelines and issues related to porting existing applications based on SIP Servlet v1.0 specification to Oracle WebLogic Communication Services and the SIP Servlet v1.1 specification. It contains the following sections:

D.1 Application Router and Legacy Application Composition

The SIP Servlet v1.1 specification describes a formal application selection and composition process, which is fully implemented in OWLCS. Use the SIP Servlet v1.1 techniques for all new development. Application composition techniques described in earlier versions of OWLCS are now deprecated.

OWLCS provides backwards compatibility for applications using version SIP Servlet 1.0 composition techniques, provided that:

- you *do not* configure a custom Application Router, and
- you *do not* configure Default Application Router properties.

D.2 SipSession and SipApplicationSession Not Serializable

The `SipSession` and `SipApplicationSession` interfaces are no longer serializable in the SIP Servlet v1.1 specification. OWLCS maintains binary compatibility for the earlier v1.0 specification, to allow any compiled applications that treat these interfaces as serializable objects to work. However, you must modify the source code of such applications before you can recompile them with OWLCS.

Version 1.0 Servlets that stored the `SipSession` as a serializable info object using the `TimerService.createTimer` API can achieve similar functionality by storing the `SipSession` ID as the serializable `info` object. On receiving the timer expiration callback, applications must use the `SipApplicationSession` and the serialized ID object returned by the `ServletTimer` to find the `SipSession` within the `SipApplicationSession` using the retrieved ID. See the SIP Servlet v1.1 API JavaDoc for more information.

D.3 SipServletResponse.setCharacterEncoding() API Change

`SipServletResponse.setCharacterEncoding()` no longer throws `UnsupportedEncodingException`. If you have an application that explicitly catches `UnsupportedEncodingException` with this method, the existing, compiled application can be deployed to OWLCS unchanged. However, the source code must be modified to not catch the exception before you can recompile.

D.4 Transactional Restrictions for SipServletRequest and SipServletResponse

SIP Servlet v1.1 acknowledges that `SipServletRequest` and `SipServletResponse` objects always belong to a SIP transaction. The specification further defines the conditions for committing a message, after which no application can modify or re-send the message. See *5.2 Implicit Transaction State* in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for a list of conditions that commit SIP messages.

As a result of this change, any attempt to modify (set, add, or remove a header) or send a committed message now results in an `IllegalStateException`. Ensure that any existing code checks for the committed status of a message using `SipServletMessage.isCommitted()` before modifying or sending a message.

D.5 Immutable Parameters for New Parameterable Interface

SIP Servlet v1.1 introduces a new `javax.servlet.sip.Parameterable` interface for accessing, creating, and modifying parameters in various SIP headers. Note that the system header parameters described in Table D–1 are immutable and cannot be modified using this new interface.

Table D–1 Immutable System Header Parameters

Header	Immutable Parameters
From	tag
To	tag
Via	branch, received, rport, wlsslport, wlssladdr, maddr, ttl
Record-Route	All parameters are immutable.
Route	For initial requests, the application that pushes the Route header can modify any of the header's parameters. In all other cases, the parameters of the Route header are immutable.
Path	For Register requests, the application that pushes the Path header can modify any of the header's parameters.In all other cases, the parameters of the Path header are immutable.

D.6 Stateless Transaction Proxies Deprecated

For applications in OWLCS, the Proxy function is always transactionally stateful, and setting the Proxy object to stateless has no effect.

The `Proxy.setStateful()` and `Proxy.getStateful()` methods are redundant: `Proxy.getStateful()` always returns true, and `Proxy.setStateful()` performs no operation.

D.7 Backward-Compatibility Mode for v1.0 Deployments

OWLCS automatically detects precompiled, v1.0 deployments and alters the SIP container behavior to maintain backward compatibility. The sections that follow describe differences in behavior that occur when deploying v1.0 SIP Servlets to OWLCS.

D.7.1 Validation Warnings for v1.0 Servlet Deployments

The SIP Servlet v1.1 specification requires more strict validation of Servlet deployments than the previous specification. In the following cases, v1.0 SIP Servlets can be successfully deployed to OWLCS, but a warning message gets displayed at deployment:

- If a listener is declared in the `listener-class` element of a v1.0 deployment descriptor but the corresponding class does not implement the `EventListener` interface, a warning is displayed during deployment. (Version 1.1 SIP Servlets that declare a listener *must* implement `EventListener`, or the application cannot be deployed).
- If a SIP Servlet is declared in the `servlet-class` element of a v1.0 deployment descriptor, but the corresponding class does extend the `SipServlet` abstract class, a warning is displayed. (Version 1.1 SIP Servlets *must* extend `SipServlet`, or the application cannot be deployed).

D.7.2 Modifying Committed Messages

The SIP Servlet v1.1 specification now recommends that the SIP container throw an `IllegalStateException` if an application attempts to modify a committed message. To maintain backward compatibility, OWLCS throws the `IllegalStateException` only when a version 1.1 SIP Servlet deployment modifies a committed message.

D.7.3 Path Header as System Header

The SIP Servlet v1.1 specification now defines the `Path` header as a system header, which cannot be modified by an application. Version 1.0 SIP Servlets can still modify the `Path` header, but a warning message is generated. Version 1.1 SIP Servlets that attempt to modify the `Path` header fail with an `IllegalArgumentException`.

D.7.4 SipServletResponse.createPrack() Exception

In OWLCS, `SipServletResponse.createPrack()` can throw `Rel100Exception` only for version 1.1 SIP Servlets. `createPrack()` does not throw the exception for version 1.0 SIP Servlets to maintain backward compatibility.

D.7.5 Proxy.proxyTo() Exceptions

For version 1.1 SIP Servlets, OWLCS throws an `IllegalStateException` if a version 1.1 SIP Servlet specifies a duplicate branch URI with `Proxy.proxyTo(uri)` or `Proxy.proxyTo(uris)`. To maintain backward compatibility, OWLCS ignores the duplicate URIs (and throws no exception) if a version 1.0 SIP Servlet specifies duplicate URIs with these methods.

D.7.6 Changes to Proxy Branch Timers

SIP Servlet v1.1 makes several protocol changes that effect the behavior of proxy branching for both sequential and parallel proxying.

For sequential proxying, the v1.1 specification requires that OWLCS start a branch timer using the maximum of the `sequential-search-timeout` value, which is configured in `sip.xml`, or SIP protocol Timer C (> 3 minutes). Prior versions of OWLCS always set sequential branch proxy timeouts using the value of `sequential-search-timeout`; this behavior is maintained for v1.0 deployments.

For parallel proxying, the v1.1 specification provides a new `proxyTimeout` value that controls proxying. The specification requires that OWLCS reset a branch timer using the configured `proxyTimeout` value, rather than using the Timer C value as required in the SIP Servlet v1.0 specification. The Timer C value is still used for v1.0 deployments.

D.8 Deprecated APIs

Earlier versions of WebLogic SIP Server provided proprietary APIs to support functionality and RFCs that were not supported in the SIP Servlet v1.0 specification. The SIP Servlet v1.1 specification adds new RFC support and functionality, making the proprietary APIs redundant. Table D–2 shows newly-available SIP Servlet v1.1 methods that must be used in place of now-deprecated WebLogic SIP Server methods. The deprecated methods are still available in this release to provide backward compatibility for v1.0 applications.

Table D–2 Deprecated APIs

Deprecated Methods (WebLogic SIP Server Proprietary)	Replacement Method (SIP Servlet v1.1)
`WlssSipServlet.doRefer(),` `WlssSipServlet.doUpdate(),` `WlssSipServlet.doPrack()`	`SipServlet.doRefer(),` `SipServlet.doUpdate(),` `SipServlet.doPrack()`
`WlssSipServletResponse.createPrack()`	`SipServletResponse.createPrack()`
`WlssProxy.getAddToPath(),` `WlssProxy.setAddToPath()`	`Proxy.getAddToPath(),` `Proxy.setAddToPath()`
`WlssSipServletMessage.setHeaderForm(),` `WlssSipServletMessage.getHeaderForm()`	`SipServletMessage.setHeaderForm(),` `SipServletMessage.getHeaderForm()`
`com.bea.wcp.util.Sessions`	See Table 6–1, "Sessions in a Converged Application".

D.9 SNMP MIB Changes

Previous versions of the OWLCS SNMP MIB definition did not follow the WebLogic MIB naming convention. Specifically, the MIB table column name label did not begin with the table name. OWLCS changes the SNMP MIB definition to prepend labels with `sipServer` in order to comply with the WebLogic naming convention and provide compatibility with WebLogic tools that generate the metadata file.

For example, in version 3.x the `SipServerEntry` MIB definition was:

```
SipServerEntry  ::=  SEQUENCE {
sipServerIndex  DisplayString,
t1TimeoutInterval  INTEGER,
t2TimeoutInterval  INTEGER,
t4TimeoutInterval  INTEGER,
....
}
```

In OWLCS, the definition is now:

```
SipServerEntry  ::=  SEQUENCE {
sipServerIndex  DisplayString,
sipServerT1TimeoutInterval  Counter64,
sipServerT2TimeoutInterval  INTEGER,
sipServerT4TimeoutInterval  INTEGER,
.....
}
```

This change in the MIB may cause backwards compatibility issues if an application or script uses the MIB table column name labels directly. All hard-coded labels, such as `iso.org.dod.internet.private.enterprises.bea.wlss.sipServerTable.t1TimeoutInterval` must be changed to prepend the table name (`iso.org.dod.internet.private.enterprises.bea.wlss.sipServerTable.sipServerT1TimeoutInterval`).

Note: Client-side SNMP tools generally load a MIB and issue commands to retrieve values based on the loaded MIB labels. These tools are unaffected by the above change.

The complete OWLCS MIB file is installed as $WLSS_HOME/server/lib/wlss/BEA-WLSS-MIB.asn1.

D.10 Renamed Diagnostic Monitors and Actions

The diagnostic monitors and diagnostic actions provided in OWLCS are now prefixed with `occas/`. For example, the SIP Server 3.1 `Sip_Servlet_Before_Service` monitor is now named `occas/Sip_Servlet_Before_Service`. You must update any existing diagnostic configuration files or applications that reference the non-prefixed names before they can work with OWLCS.

Index