BEA WebLogic Network Gatekeeper™

Developer's Guide for Extended Web Services

Version 1.0
Document Revised: March 14, 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

4. Extended Web Services Examples

A. References

CHAPTER 1

Introduction and Roadmap

The following sections describe the audience for and organization of this document:

Document Scope and Audience

The purpose of this guide is to describe how to develop telecom-enabled applications based on the Extended Web Services APIs and how to access and use the APIs/interfaces as offered by the BEA WebLogic Network Gatekeeper.

This guide contains code fragments from example applications written in Java to illustrate different aspects of the usage of the interfaces.

The purpose of this guide is not to describe Web Service development in general, but rather how to use the specific interfaces.

All example code is Axis-specific.

Guide to this Document

The document contains the following chapters:

- Chapter 1, "Introduction and Roadmap," informs you about the structure and contents of this document, the used writing conventions, and related documentation.
- Chapter 2, "Introduction and Overall Workflow," gives an introduction to the two main types of WebLogic Network Gatekeeper Web services applications. It also tells you about the programming environment and development workflows.
- Chapter 3, "Using the Extended Web Services," describes the Access service used for session handling and the Extended Web Services service capabilities.
- Chapter 4, "Extended Web Services Examples," contains examples of usage of the Extended Web Services.

Terminology

The following terms and acronyms are used in the document:

API —Application Programming Interface

CORBA —Common Object Request Broker Architecture

HTML —Hypertext Markup Language

MMS —Multimedia Message Service

RPC —Remote Procedure Call

ORB —Object Request Broker

SMS —Short Message Service

SwA —SOAP with Attachments

WSDL —Web Services Definition Language

WSI-I —Web Services Interoperability

SPA —Service Provider API

XML —Extended Markup Language

Related Documentation

This Developer's Guide is a part of the WebLogic Network Gatekeeper documentation set. The following documents contain other types information:

- *API Description Extended Web Services for WebLogic Network Gatekeeper*

  The API description describes the Extended Web Services API.

CHAPTER 2

Introduction and Overall Workflow

The following sections introduce the development workflow for developing Extended Web Services:

About WebLogic Network Gatekeeper Web Services applications

WebLogic Network Gatekeeper Web Services applications are services offering their users access to telecom functionality. The applications can access the telecom functionality through two different Web services APIs. Which API to use depends on the application's needs for network functionality and means of access.

For applications that will provide standardized basic telecom functionality, it is recommended to use the Parlay X APIs.

For applications with a need for more granular control, the Extended Web Services interfaces can be used.

This guide describes how to develop Extended Web Services applications that connects to WebLogic Network Gatekeeper. WebLogic Network Gatekeeper acts as a gateway to the underlying telecom network.

Using the Extended Web Services Interfaces you can quickly develop powerful telecom-enabled applications using any programming environment supporting Web Services.

Architecture

Figure 2-2, "Extended Web Services interfaces and JS2SE applications," on page 2-4 illustrates different ways of using the Web Services APIs as provided by WebLogic Network Gatekeeper, as well as examples of different execution environments for applications.

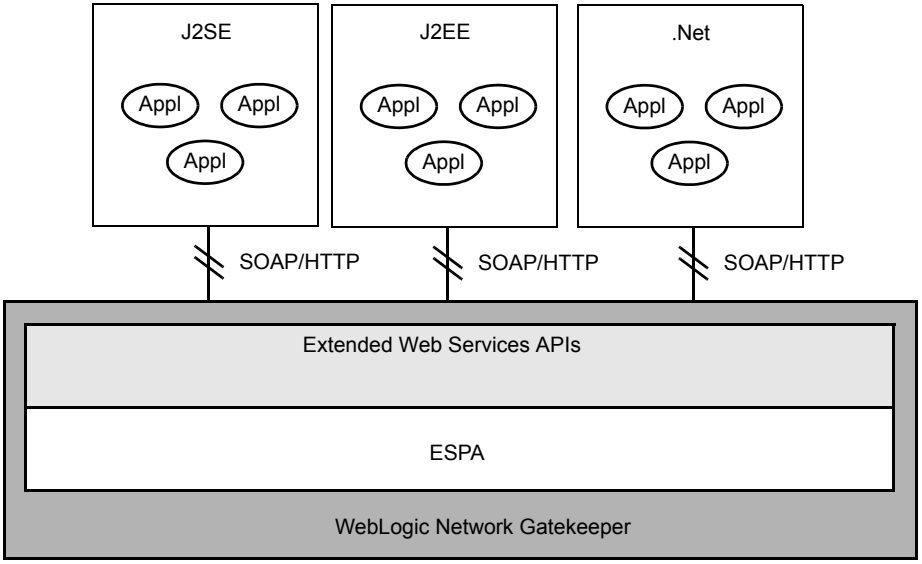
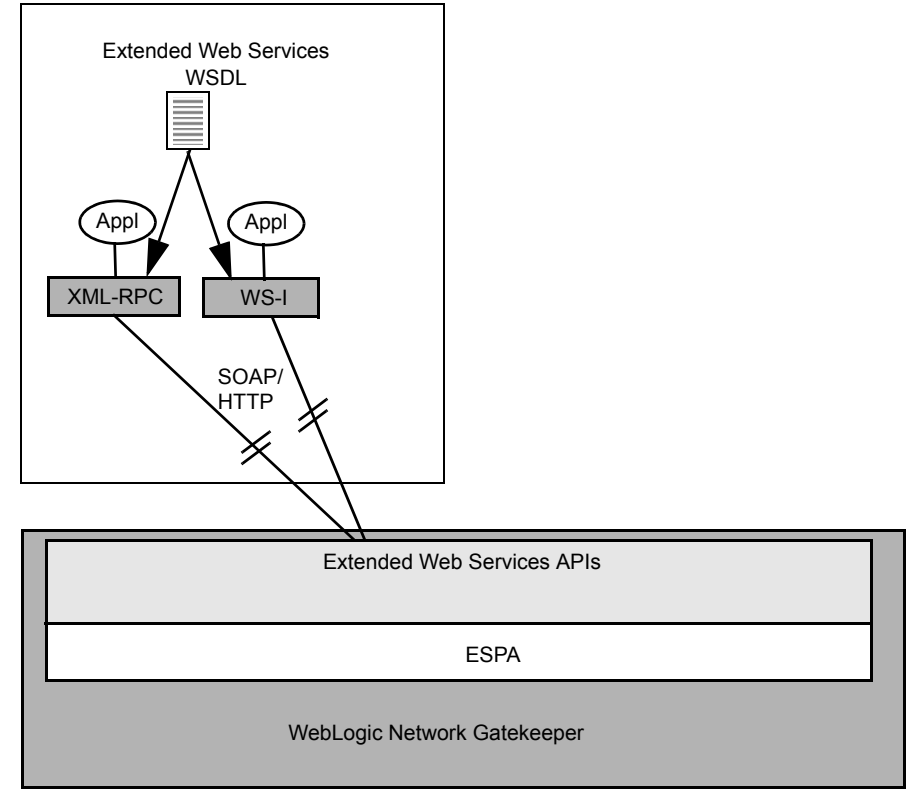


Figure 2-1 Extended Web Services interfaces on ESPA

There are two flavours of Web Services; XML-based RPC and WS-I.



Appl - Application
WS-I - Web Services Interoperability
XML RPC - Handles packaging of SOAP messages
WSDL - Web Services Definition Language

Figure 2-2 Extended Web Services interfaces and JS2SE applications

Web services applications

Web services applications executes in an environment capable of handling Web Services.

The Web Services applications communicates with WebLogic Network Gatekeeper using SOAP/HTTP.

Extended Web Services based applications

An Extended Web Services application

- uses an API that can be extended.
- benefits from a feature-rich API.
- is state-oriented.

Development environment

Below is a description of an example development environment. Integrated programming environments, like Visual Studio .Net can be used for development of Web Services applications, but this guide uses a minimalistic approach. For the purpose of this guide, the following will do:

- an ordinary text editor.
- Java 2 SDK 1.4.2, see J2SE SDK, http://java.sun.com.
- Axis 1.1, see Apache Axis, http://ws.apache.org/axis/.
- JavaMail API 1.2, see JavaMail, http://java.sun.com, for messaging applications handling multimedia messages.

The following files from the Axis distribution are used:

- axis.jar
- axis-ant.jar
- commons-discovery.jar
- commons-logging.jar
- jaxrpc.jar
- saaj.jar
- wsdl4j.jar

The following JavaMail files are used:

- mail.jar
- activation.jar

Information exchange with the service provider

Before an application is developed, the application developer and the service provider must exchange information regarding resources.

The first step for the application developer is to define which resources to use, call, messaging, location, status, payment etc. and to map these requirements to an APIs that corresponds to these resources.

The next step is to exchange the information according to "Information exchange between application developer and WebLogic Network Gatekeeper operator" on page 2-7.

Table 2-1 Information exchange between application developer and WebLogic Network Gatekeeper operator

Module	Information to be provided by the	
	Application developer	Service provider
Access		Application ID. Service provider ID. Application Instance Group ID. Password for the Application Instance Group.
Call Control		For application initiated calls: Any eventual restrictions on allowed numbers. For network initiated calls: Access number to the application. Can be a range of numbers.
Call User Interaction		Details about IVRs, such as access number, announcement IDs, input capabilities and so on.
Content Based Charging		Currencies allowed, any restrictions on allowed maximum or minimum values.
Messaging		Access number to the application. Mailbox ID and corresponding password.

Table 2-1 Information exchange between application developer and WebLogic Network Gatekeeper operator

Module	Information to be provided by the	
	Application developer	Service provider
Message User Interaction		User information codes to be used. Language codes. User Interaction Codes for network-initaed user interaction sessions.
Subscriber Profile		Properties that can be set of get.
User Location		Supported uncertainty shapes.
User Status		-

The WebLogic Network Gatekeeper operator must also communicate which services and methods that are supported by the deployment.

In addition to this information, other information related to commercial, security, and privacy regulations must also be exchanged. Examples includes:

- Charging plans to use
- Number of concurrent application instances.
- Amount of usage of the different resources, for example allowed number of send SMS requests.
- Black/white listed addresses.
- Allow/deny lists for user status and user location requests.

Overall development workflows

Below you find overall workflows for development of Web Services applications based on the Extended Web Services interfaces.

Two main scenarios are identified:

- WebLogic Network Gatekeeper acts as a server and the application is the client. In this scenario, the application uses a Web Service provided by WebLogic Network Gatekeeper.
- the application acts as a server and WebLogic Network Gatekeeper is the client. In this scenario, the application in itself is the provider of a Web Service and WebLogic Network Gatekeeper invokes methods on this Web Service.

Often, an application acts as both server and client.

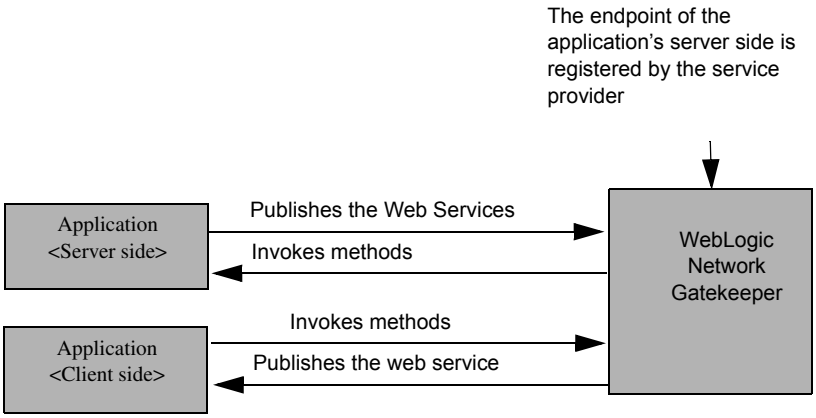


Figure 2-3 Subscribing for notifications

The methods that the application invokes on WebLogic Network Gatekeeper are defined in WSDL files, one for each service capability, where the name of the file reflects the capability. Likewise are the methods that WebLogic Network Gatekeeper invokes on the application defined in WSDL files, one for each service capability. The names of these files ends with "Listener".

The method invocations are SOAP requests over HTTP, which means that the server part of the application must be capable of handling SOAP requests.

When using Axis, the Simple Axis Server can be used as a SOAP engine during test. In a production system can, for example, Axis in combination with Tomcat be used.

Client-side Web Services using XML based RPC

Below is an overall work sequence for developing telecom enabled Web Services using XML-based RPC:

1. Make sure to retrieve the necessary IDs for the resources the application will use from the service provider. Examples are mailbox IDs, short numbers for network triggered applications and so on.
2. Retrieve the WSDL file that handles user login.
3. Retrieve WSDL files for the telecom services to use.
4. Generate stubs/proxy classes for the language to implement the application in. Use a tool that converts the WSDL into stubs for the preferred language. Examples of such tools are WSDL2Java and Soap Toolkit.
5. Compile and create Jar-files from the Java stubs.
6. Use the generated APIs to add telecom functionality to the application.
7. Compile the application.
8. Test the application in a test environment, for example WebLogic Network Gatekeeper Application Test Environment.
9. Connect the application to a WebLogic Network Gatekeeper with a connection to a live telecom network.

Server-side Web Services using XML based RPC

Below is an overall work sequence for developing telecom enabled Web Services using XML-based RPC:

1. Make sure to retrieve the necessary IDs for the resources the application will use from the service provider. Examples are mailbox IDs, short numbers for network triggered applications and so on.
2. Retrieve WSDL files for the telecom services to use.
3. Generate skeleton classes for the language to implement the application in. Use a tool that converts the WSDL into stubs for the preferred language. Examples of such tools are WSDL2Java and Soap Toolkit.
4. Compile and create Jar-files from the Java stubs.
5. Implement the generated interfaces and add telecom functionality to the application.
6. Adapt the generated WSDD file to bind the SOAP requests to the appropriate class.
7. Compile the application.

8. Deploy the application in an environment capable of decoding HTTP/SOAP messages. Examples of such environments includes Axis.
9. Test the application in a test environment, for example WebLogic Network Gatekeeper Application Test Environment.
10. Connect the application to a WebLogic Network Gatekeeper with a connection to a live telecom network.

Example: Server-side Web Service

The example below shows how to define a web service that takes care of notifications on new SMS:es from to the applications' server side. The web service is the Messaging Listener interface, containing the method newMessageAvailable(...).

The Simple Axis Server is used as deployment environment for the application.

Below is an outline on the procedure:

1. Generate Java skeletons from the WSDL files:

   ```
   %java org.apache.axis.wsdl.WSDL2Java --server-side
   --skeletonDeploy true MessagingListener.wsdl
   ```

   Note: The Axis files must be in the classpath.

2. Compile and create Jar-files from the skeletons.
3. Move the empty implementation of the generated interfaces to the source directory of the application.

   In the example, the class is named `MessagingListenerSoapBindingImpl`. When generating skeletons using WSDL2Java, the empty interface implementations are named `<Name of API>BindingImpl`.

4. Adapt the generated Web Service Deployment Descriptor (WSDD) files to bind the SOAP request to the appropriate class.

   The WSDD files are used when deploying and undeploying services. Two files are generated: `deploy.wsdd` and `undeploy.wsdd`.

   In the example, the tag

   ```
   <parameter name="className"
   value="MessagingListenerSoapBindingSkeleton"/>
   ```

   is replaced with

```
<parameter name="className"
value="com.acme.apps.getmessageapp.MessageNotificationHandler"/>
```

in order to bind to the appropriate class.

5. Compile the application.
6. Verify that the application is deployed correctly by using a Web browser and point it to the URL of the web service. In the case of Simple Axis Server, the deployed Web Services can be found at the URL http://<host>:<port>/axis/services
7. Run the application. The adapted file deploy.wsdd is used when instantiating the Simple Axis Server.

Testing an application

Figure 2-4, “Application test flow,” on page 2-13 shows the application test flow, from the application developers’ functional test to deployment in a live network. An application developer can perform functional tests using Weblogic Network Gatekeeper Application Test Environment. The other tests in the flow are performed in cooperation between the application provider and the service provider.

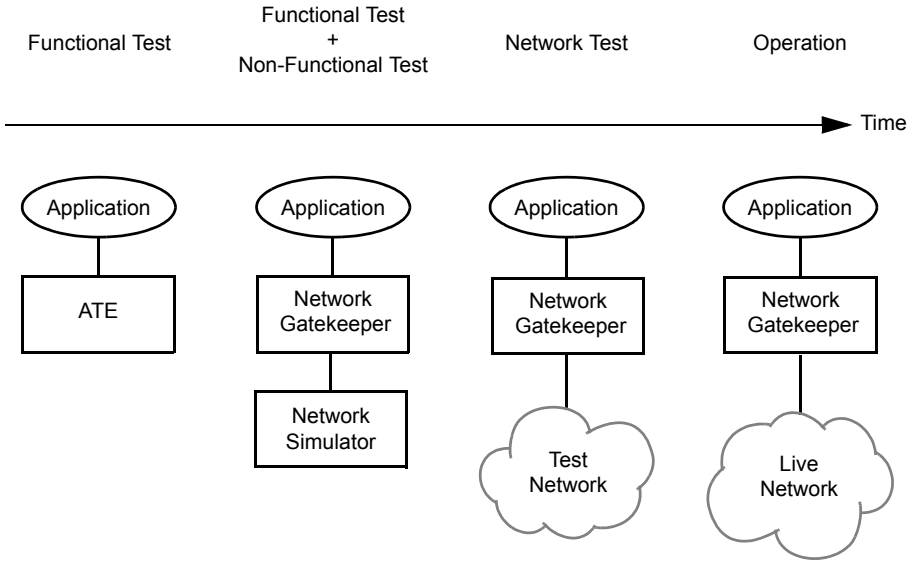


Figure 2-4 Application test flow

When an application shall be tested using Weblogic Network Gatekeeper Application Test Environment (ATE), the application is connected to ATE, which emulates WebLogic Network Gatekeeper. Before testing in a test telephony network, a network simulator can be used.

An overview of the relation between Weblogic Network Gatekeeper Application Test Environment and WebLogic Network Gatekeeper is shown in Figure 2-5, “Weblogic Network Gatekeeper Application Test Environment (ATE) in relation to WebLogic Network Gatekeeper,” on page 2-14.

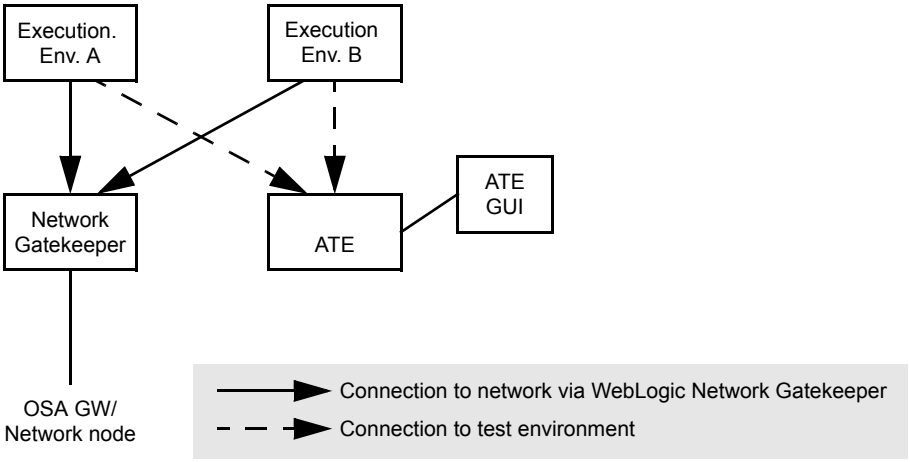


Figure 2-5 Weblogic Network Gatekeeper Application Test Environment (ATE) in relation to WebLogic Network Gatekeeper

For a applications based on Web Services, the applications uses the endpoints provided by Weblogic Network Gatekeeper Application Test Environment during test. After successful verification, the application uses endpoints provided by WebLogic Network Gatekeeper.

CHAPTER 3

Using the Extended Web Services

The following sections describe how to use the Extended Web Services APIs:

About the Extended Web Services APIs

The Extended Web Services interface offers a high-level abstraction of OSA/Parlay for use in a web services environment. It offers a more granular control over resources in the telecom network than Parlay X, and yet it provides the same low degree of complexity.

The API is designed for rapid application development. From an architectural point of view, the implementation of the API resides on top of the service capabilitiy modules in WebLogic Network Gatekeeper.

All applications accessing WebLogic Network Gatekeeper through the Web Services interfaces uses a Kerberos type of service token-based authentication. The application is provided with a user name (the application instance group ID) and a password. When an application wants access to WebLogic Network Gatekeeper the application instance logs in using the user name and password together with the application account ID and service provider ID to retrieve a service token. This mechanism may be extended, as an option using, for example Passport or other extended Kerberos Key Distribution Centre (KDC) authentication solutions. This is according to the WSSE (Web Services Security) standard.

To run an Extended Web Services application, access to either WebLogic Network Gatekeeper or Weblogic Network Gatekeeper Application Test Environment is needed, together with a set of WSDL files defining the API, login credentials and IDs of resources to use. These are provided by the service provider.

The interfaces are separated in different modules. Each main component is contained in a specific module. The modules are:

Table 3-1 Information exchange between application developer and WebLogic Network Gatekeeper operator

Module	Defines
Access	Methods for session handling between an application and WebLogic Network Gatekeeper.
Call Control	Methods for controlling and routing telephony calls.
Call User Interaction	Methods for dialogue handling of between IVRs and telephony users.
Content Based Charging	Methods for handling charging based on content.

Table 3-1 Information exchange between application developer and WebLogic Network Gatekeeper operator

Module	Defines
Messaging	Methods for handling sending and reception of SMS:es, MMS:es and other messages.
Messaging User Interaction	Methods for dialogue handling using SMS or USSD.
Subscriber Profile	Methods for setting and getting application or end user data, such as terminal capabilities and preferred currency.
User Location	Methods for retrieving the geographical position of a mobile terminal.
User Status	Methods for retrieving information on the status of mobile terminals.

For detailed information on individual methods, see API Description Extended Web Services for WebLogic Network Gatekeeper.

WSDL files

WebLogic Network Gatekeeper supports Extended Web Services interfaces using the SOAP encoding (RPC encoding) approach.

By default, the WSDL files can be fetched from:

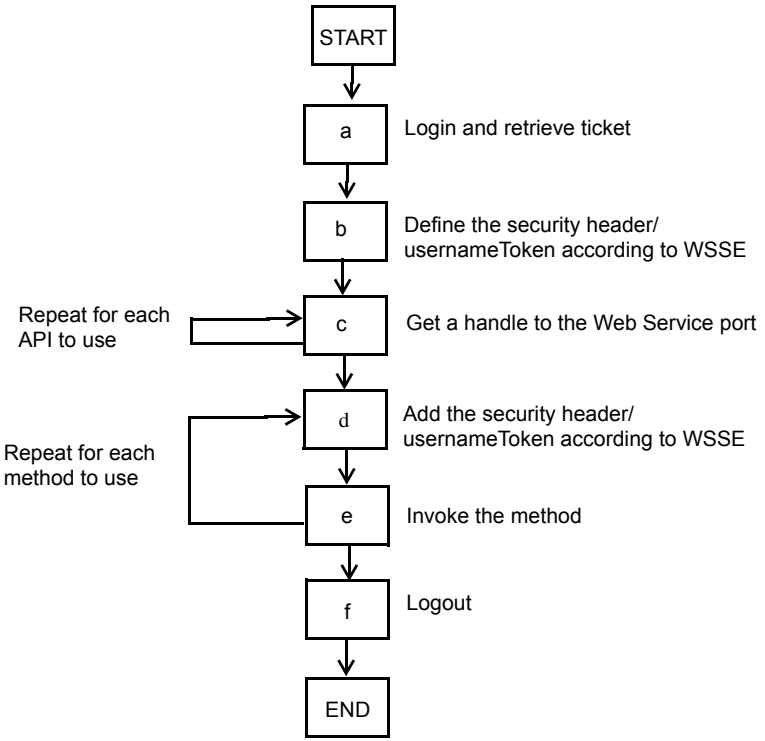
- http://<URL to WebLogic Network Gatekeeper>/wespa/services (The Web Services, also the endpoints)
- The WSDL files for the northbound (Listener) interfaces can be fetched from http://<URL to WebLogic Network Gatekeeper>/wespa/wsdl (definitions of the listener interfaces)

Module	WSDL-file
Access	Access
Call Control	CallControl
Call Control listener	CallControlListener
Call User Interaction	CallUserInteraction
Call User Interaction listener	CallUserInteractionListener
Content Based Charging	ContentBasedCharging
Content Based Charging listener	ContentBasedChargingListener
Messaging	Messaging
Messaging listener	MessagingListener
Messaging User Interaction	MessagingUserInteraction
Messaging User Interaction listener	UserInteractionListener
Network triggered User Interaction listener	UserInteractionNetworkListener
Subscriber Profile	SubscriberProfile
Subscriber Profile	SubscriberProfileListener
User Location	UserLocation
User Location listener	UserLocationListener
User Status	UserStatus
User Status listener	UserStatusListener

For a description of the methods in each API, see API Description Extended Web Services for WebLogic Network Gatekeeper.

Workflow

The main program control flow when executing applications based on the Extended Web Services interfaces is described in pseudo code in Figure 3-1, “Application execution workflow,” on page 3-6.



a See “Login and retrieve login ticket” on page 3-6.

b See “Define the security header” on page 3-7.

c See “Get hold of a Port” on page 3-8.

d See “Add security header” on page 3-8.

e See “Invoke a method” on page 3-9.

f “Logout” on page 3-9

Figure 3-1 Application execution workflow

Login and retrieve login ticket

To use any Web Service provided by WebLogic Network Gatekeeper a login ticket is needed. A login is needed to retrieve the ticket. The login ticket identifies the login session. This ticket is

valid until a logout is performed. The ticket is sent in each consecutive method invocation to identify the originator of the invoker.

Details about locators, endpoints, so on are explained later in this section

Listing 3-1 Login

```
AccessService accessService = new AccessServiceLocator();
java.net.URL endpoint = new java.net.URL(wsdlUrl);
Access access = accessService.getAccess(endpoint);
String loginTicket = access.applicationLogin(spID,
                                             appID,
                                             appInstGroupID,
                                             appInstGroupPassword);
```

The login ticket ID retrieved when invoking applicationLogin is used in each consecutive invocation towards WebLogic Network Gatekeeper. See “Define the security header” on page 3-7.

The login credentials; spID, appID, appInstGroupId, and appinstGoupPassword are provided by the service provider.

Define the security header

The login ticket ID, as retrieved when logging in, is sent in the SOAP header together with a username/password combination for each invocation of a web service method.

Listing 3-2 Define the security header

```
org.apache.axis.message.SOAPHeaderElement header =
new org.apache.axis.message.SOAPHeaderElement(wsdlUrl, "Security", "");
header.setActor("wsse:PasswordToken");
header.addAttribute(wsdlUrl, "Username", ""+userName);
```

```
header.addAttribute(wsdlUrl, "Password", ""+sessionId);
header.setMustUnderstand(true);
```

The login ticket is supplied in the Password attribute. The userName attribute is defined by the service provider, normally in the format <myUserName>@<myapplication>.

Axis 1.1 does not contain WSSE helper classes, so this is performed manually.

The header is defined upon the object representing the Web Service port to use. Also see "Add security header" on page 3-8.

Get hold of a Port

Below is the code for getting hold of a port. The example is using the Messaging interface.

Listing 3-3 Get hold of a port

```
MessagingService messagingService = new MessagingServiceLocator();
java.net.URL endpoint = new java.net.URL(messagingWsdlUrl);
messaging = messagingService.getMessaging(endpoint);
```

The details on the parameters of the messaging API are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

Add security header

Adding the security header to a request is straightforward, as illustrated below. For information on how create the header, see "Define the security header" on page 3-7.

Listing 3-4 Add security header

```
((org.apache.axis.client.Stub)sendSms).setHeader(header);
```

Invoke a method

Below it is illustrated how to get hold of a port. The example is using the Send SMS API.

Listing 3-5 Invoke a method

```
String mailboxTicket = messaging.openMailbox(myMailbox,
                                                myMailboxPwd,
                                                  spID+appID+appInstGroupID);
```

The details on the parameters of the send SMS API are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

Logout

The logout destroys the login ticket.

Listing 3-6 Logout

```
access.applicationLogout(sessionId);
```

The login Ticket is destroyed and cannot be used in consecutive method invocations.

Access

The following functionality is provided:

- Login an application to WebLogic Network Gatekeeper.
- Logout an application from WebLogic Network Gatekeeper.
- Change password.

For a description on how to use this API, see "Login and retrieve login ticket" on page 3-6 and "Logout" on page 3-9. There is also support for changing the password.

Messaging

The messaging service capability makes it possible for an application to send, store, and receive SMSes and Multi Media (MMS) messages. In addition, the service supports send lists, smart messaging, EMS, and distribution of ring tones and logos. The send list feature allows for send list distribution of messages.

An administrator creates mailboxes with INBOX and OUTBOX folders for each subscriber or application. A mailbox structure is given in Figure 3-2.

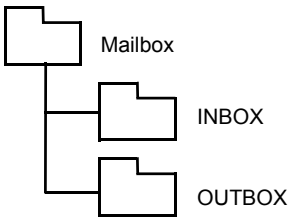


Figure 3-2 Mailbox structure

An application is notified when a sent message has been successfully delivered to a recipient and when the service receives a message in an INBOX related to the application. Old messages are automatically removed from the mailboxes. The cleanup interval and age of messages to be deleted are configurable.

Charging

The charging service capability makes it possible for an application to charge a subscriber based on the content (content based charging) of a used service, for example a music video, rather than based on the amount or time used. Reservation/payment in parts and immediate charging are supported.

In immediate charging the amount is withdrawn from the subscriber's account at the same time the service is ordered.

To make sure the subscriber does not have to pay for a service not delivered, reservation/payment in parts can be used. In this case, the charging service reserves the whole or a part of the amount in the subscriber's account. The amount is not withdrawn until it has been verified that the subscriber has received the whole or a defined part of the service paid for. Reservation/payment

in parts can also be used by an application before delivering a service to make sure that the amount to be charged is available on the subscriber's account.

The charging service also supports adding money to subscriber accounts.

Call

The call service capability provides applications with functions for call routing, call management, and call leg management. More than two call legs can be connected to a call simultaneously.

Two main usage scenarios for call control are identified; application initiated and network triggered calls.

Network triggered calls

Network triggered call is used for applications where call set up is triggered from the network, see Figure 3-3, "Example of a network triggered call," on page 3-11.

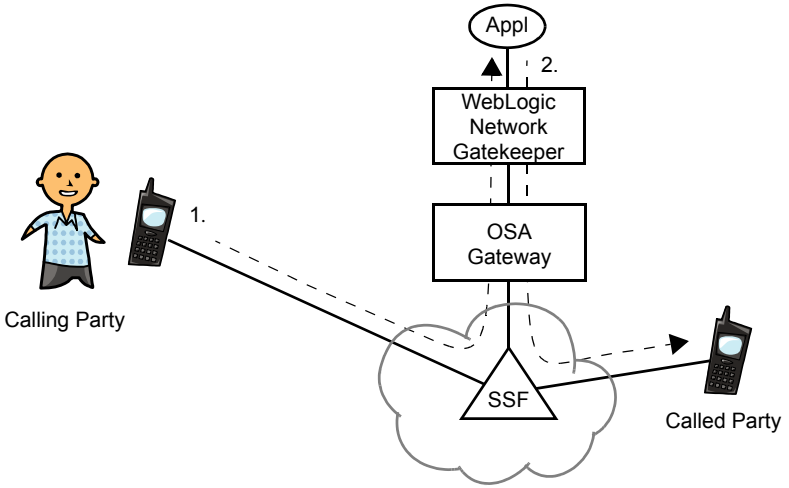


Figure 3-3 Example of a network triggered call

The service contains functions that makes it possible for an application to provide:

- advice of charge
- call specific charging

- call re-routing
- user interaction through announcements and voice prompts

Application initiated calls

Typical usage for application initiated calls are voice chat applications and different types of click-to-call functionality in web and office applications. Figure 3-4, “Example of an application initiated call,” on page 3-12 shows an example where a call is set up using a web based address book application.

In addition, a call between two or more persons can be set up through an application interface. During the call it is possible to add and remove participants through the interface. Also, notifications when individual participants answers and hangs up can be presented.

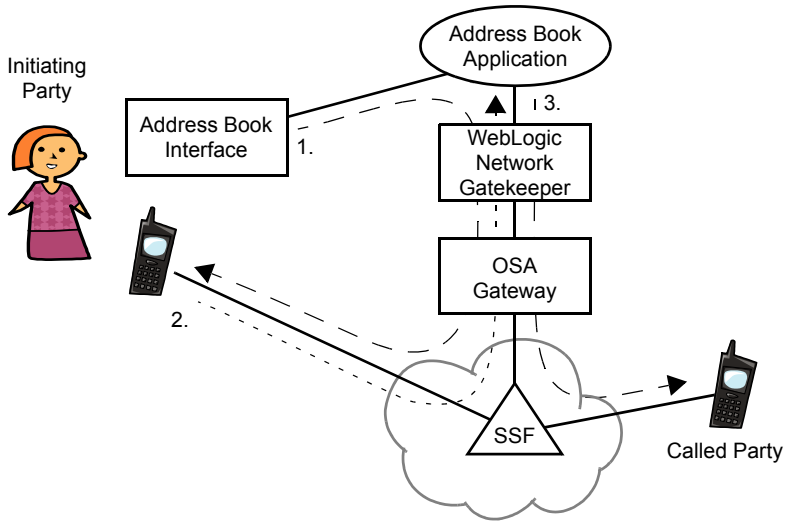


Figure 3-4 Example of an application initiated call

Subscriber profile

The subscriber profile service makes it possible for an application to obtain and manage application subscriber profiles. A subscriber profile consists of data related to a subscriber and

the subscriber's telephony terminal, see Table 3-2 on page 13. The data marked as *read-only* in the table can only be updated by the operator.

Table 3-2 Subscriber Profile Data

Data	Description/Example
Name	Subscriber name
Alias	Alias to ensure the subscribers anonymity towards other users.
Address	Complete postal address
Home phone	-
Office phone	-
Private mail	Home e-mail
Office mail	Office e-mail
Terminal ID	IMSI number
Terminal vendor	For example Nokia or Ericsson
Terminal model	For example 6610 or T630
Screen size	Character rows x columns
Colour terminal	Yes/No
MMS terminal	Yes/No
Fax number	-
Group identity	For example family, office location or work group
Gender	Male/Female
Birth date	In format YYYY-MM-DD
Nationality	-
Mother tongue	-

Currency	-
Miscellaneous	Any type of additional information
Last updated	Date and time the account was last updated (read-only)
Updated by	The user that updated the account (read-only)
Subscription type	Type of subscription, Prepaid, Postpaid, Time Limited, or Free (read-only)
Payment method	Payment method: Credit card or Invoice (read-only)
Balance	Account balance (read-only)
Application subscriptions	List of subscribed applications (combinations of service provider and application IDs) (read-only)

User interaction

The user interaction service makes it possible for an application to interact with call participant(s) during a call or with messaging users during a messaging session. The application communicates with call participant(s) through announcements or messages and with messaging users through text messages.

Call user interaction

The call participant(s) communicate through speech or tone sending. That is, both speech recognition and DTMF (using the terminal's key set (0-9, *, #)) can be used. Announcements can be purely informative or they can prompt a participant to reply through speech or sending DTMF tones back to the application.

Message based user interaction

With message based user interaction, the application and the end users communicate through text messages (SMSes or USSD messages).

SMS based user interaction provides application initiated SMSes with a transaction ID to connect requesting/prompting SMSes with end user's replies.

USSD messages from an application can be purely informative or they can prompt the end user to reply. USSD messages can also be used by the end user to initiate service sessions with applications. When initiating service sessions or replying to an application generated USSD message, the end user can only use the terminal's key set (0-9, *, #). The application can use any type of character supported by the end user's terminal.

User location

The user location service capability makes it possible for an application to obtain the geographical location of telephony terminals. The service supports:

- single location requests
- periodic location request
- triggered location requests

Both single and periodic requests supports multiple destination addresses in one request.

The location can be specified as a base point (longitude, latitude) or as a descriptive (abstracted) position. An abstracted position describes the user's location in terms of:

- Street address
- Zip code
- City
- State
- Area (operator defined)
- Country
- Network (operator defined)

Use of abstracted location information requires interaction with a geographic information system.

Using longitude and latitude, the location is specified as a base point (longitude, latitude) and a geometrical area in which the telephony terminal is located. The geometrical area is referred to as an uncertainty shape related to the base point. The uncertainty shapes are divided in to circles and ellipses.

When supported in the network, extended location, the altitude, terminal type, and a time stamp are also provided.

Also, the service supports provision of geographical information for a terminal, such as city or street address, to applications.

Exactly what is available to the requesting applications is dependant on the underlying networks.

Circle uncertainty shapes

The circle uncertainty shapes are:

- Circle
- Circle sector
- Circle arc

The circle sector is an extended case of the circle, and the circle arc is a extended case of the circle sector, see Figure 3-5, “Circle uncertainty shape definitions,” on page 3-16.

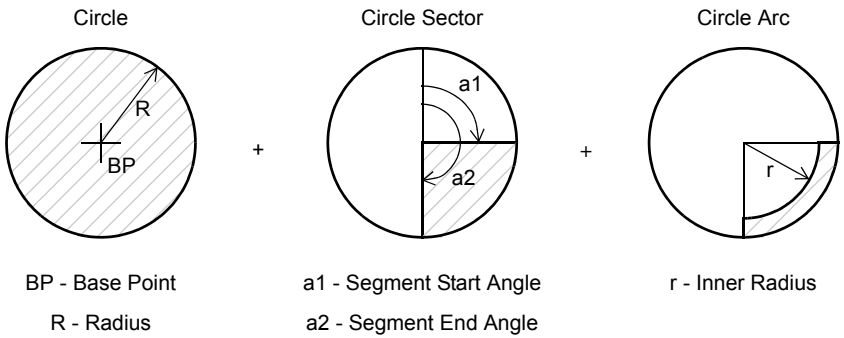


Figure 3-5 Circle uncertainty shape definitions

Ellipse uncertainty shapes

The ellipse uncertainty shapes are:

- Ellipse
- Ellipse sector
- Ellipse arc

The ellipse sector is an extended case of the ellipse, and the ellipse arc is a extended case of the ellipse sector, see Figure 3-6, “Ellipse uncertainty shape definitions,” on page 3-17.

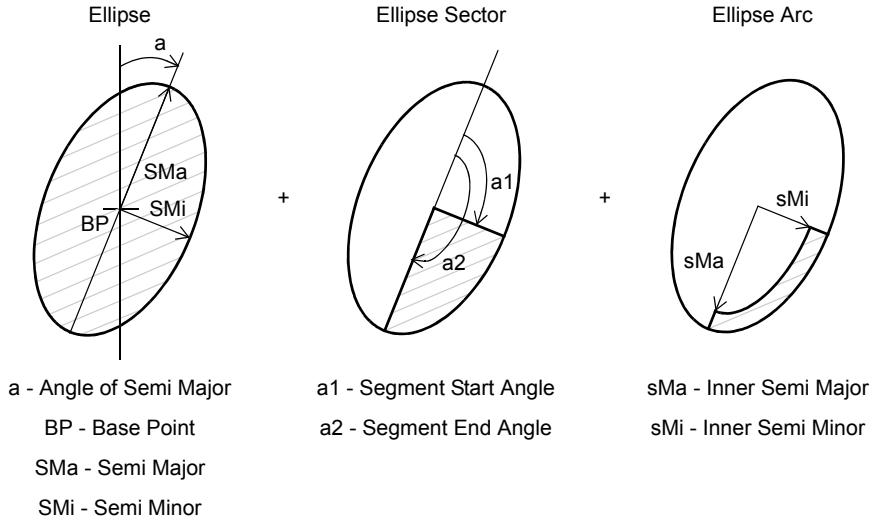


Figure 3-6 Ellipse uncertainty shape definitions

Terminal altitude

If the terminal’s altitude is provided, the actual terminal altitude is somewhere within a span defined by the provided altitude value and two times the altitude uncertainty, see Figure 3-7, “Terminal altitude definition,” on page 3-18.

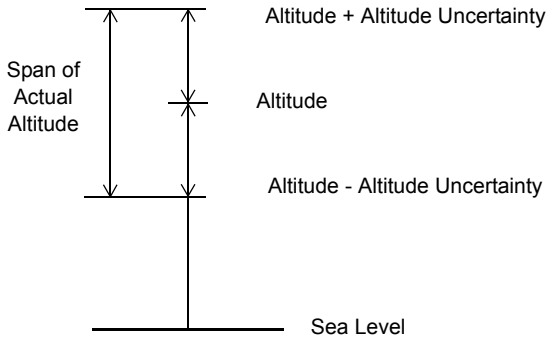


Figure 3-7 Terminal altitude definition

A positive altitude value means above sea level, whereas a negative value means below sea level.

User status

The user status service capability makes it possible for an application to obtain the status of fixed, mobile and IP-based telephony terminals. Possible values are:

- Reachable
- Busy
- Not reachable

The service supports single, periodic and triggered status request and as well as multiple destination addresses in one request.

Exception handling

Currently, there are three different types of exceptions, as described below.

Service-specific exceptions

Service-specific exceptions, one or more for each service.

Name	May occur during
CallException	Call session.
CallSetupException	Call setup.
CallUIException	User interaction session.
ContentBasedChargingException	Invocation of all methods in Content based charging.
UserStatusException	Retrieval of a user status report.
InformationException	Invocation of all methods in Information.
UserLocationException	Retrieval of a user status report
MessagingException	Invocation of all methods in Messaging.
SubscriberProfileException	Invocation of all methods in Subscriber profile.

AccessException

Access exceptions, which are thrown when something is wrong with the data related to an operation. For instance, a user tries to retrieve a manager for a service that he or she is not allowed to use.

CommunicationException

Communication exceptions, which are thrown when a disturbance occurs in the network between the application and WebLogic Network Gatekeeper, or WebLogic Network Gatekeeper has lost some objects related the session.

CHAPTER 4

Extended Web Services Examples

The following sections describe the Extended Web Services examples:

About the examples

Below are a set of examples given that illustrates how to use the Extended Web Services interfaces using AXIS and Java.

Send SMS

Get hold of the Messaging Web Service.

Listing 4-1 Get hold of the Messaging Service

```
MessagingService messagingService = new MessagingServiceLocator();
java.net.URL endpoint = new java.net.URL(messagingWsdlUrl);
messaging = messagingService.getMessaging(endpoint);
```

The security header is created as outlined in "Define the security header" on page 3-7 and the header is added to the port.

Listing 4-2 Add the security header

```
header.setMustUnderstand(true);
((org.apache.axis.client.Stub)messaging).setHeader(header);
```

The mailbox is opened and an identifier ticket for the mailbox is returned.

Listing 4-3 Open the mailbox

```
String mailboxTicket;
mailboxTicket = messaging.openMailbox(myMailbox,
                                      myMailboxPwd,
                                         spID+appID+appInstGroupID);
```

The messaging properties are defined and the method for sending SMSes is invoked.

The origAddress is a combination of the mailbox ID as given by the service provider, the corresponding password and the originator address. The format is "tel:<mailboxID>", for example "tel:50001".

serviceCode is operator-specific, it is used for charging purposes. The message is an ordinary String. The last parameter is operator-specific.

One or more send results are returned, one for each destination address.

Listing 4-4 Define message properties and send the SMS

```
arrayofMessagingProperties[0] = new MessagingProperty();

arrayofMessagingProperties[0].setMessagingPropertyName
                                (MessagingPropertyName.MESSAGE_SENT_TO);

arrayofMessagingProperties[0].setValue(destAddress);

arrayofMessagingProperties[1] = new MessagingProperty();

arrayofMessagingProperties[1].setMessagingPropertyName
                                (MessagingPropertyName.MESSAGE_SENT_FROM);

arrayofMessagingProperties[1].setValue(origAddress);

arrayofMessageSendResult = messaging.sendSMS(mailboxTicket,
                                             myMessage,
                                             arrayofMessagingProperties,
                                             serviceCode,
                                                spID+appID+appInstGroupID);
```

The previously opened mailbox is closed. This destroys the mailbox ticket.

Listing 4-5 Close the mailbox

```
messaging.closeMailBox(mailboxTicket);
```

Below is outlined how the delivery status can be retrieved from the send result array.

Listing 4-6 Get delivery status

```
for (int i = 0; i < arrayofMessageSendResult.length; i++) {
   System.out.println("Message ID: " +
                        arrayofMessageSendResult[i].getMessageID());
   System.out.println("Destination address: " +
                        arrayofMessageSendResult[i].getAddress());
   System.out.println("Status: " +
                arrayofMessageSendResult[i].getSendStatus().getValue());
```

Message Notifications

Message notifications, notification on new messages, are sent asynchronously from WebLogic Network Gatekeeper. This means that the application must implement a Web Service. The initial thing is to start the Web Service server and deploy the implementation of the Web service into the server. The deployment is made using a deployment descriptor that is automatically generated when the Web Service java skeletons are generated. The deployment descriptor (deploy.wsdd) is modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 4-8, “Implementation of the MessageNotificationHandler Web Service,” on page 4-5. The class is based on the auto-generated class MessagingListenerSoapBindingImpl.

.

Listing 4-7 Start SimpleAxis server

```
 // start SimpleAxisServer
org.apache.axis.transport.http.SimpleAxisServerserver =
            new org.apache.axis.transport.http.SimpleAxisServer();
System.out.println("Opening server on port: "+ port);
```

```
ServerSocket ss = new ServerSocket(port);
server.setServerSocket(ss);
server.start();
System.out.println("Starting server...");
// Read the deployment description of the service
InputStream is = new FileInputStrem(deploymenDescriptorFileName);
// Now deploy our  web service
org.apache.axis.client.AdminClient adminClient;
adminClient = new org.apache.axis.client.AdminClient();
System.out.println("Deploying receiver server web service...");
 adminClient.process(new org.apache.axis.utils.Options
                           (new String[] {"-ddd","-tlocal"}),
                              deploymentDescriptorStream);
System.out.println("Server started. Waiting for connections on: " + port);
```

The listener class implements the MessagingListener interface. A set of methods must be defined, in the example there is only code in the newMessageAvailable method in order to outline how to get handle the parameters. messageDescr contains information about the message itself. The message ID is used to fetch the content of the SMS.

Listing 4-8 Implementation of the MessageNotificationHandler Web Service

```
public class MessageNotificationHandler implements MessagingListener{
   public void deactivate(java.lang.String notificationTicket)
               throws java.rmi.RemoteException {
   }

   public void newMessageAvailable(String notificationTicket,
                                   String mailbox,
                                   MailboxFolder folder,
```

```
                                    MessageDescription messageDescr)
                throws java.rmi.RemoteException {
      System.out.println("->New Message arrived");
      System.out.println("Mailbox " + mailbox );
      System.out.println("Folder " + folder );
      System.out.println("Message Description");
      System.out.println("ID " + messageDescr.getMessageId() );
      System.out.println("Format " + messageDescr.getFormat() );
   }

   public void messageDeliveryAck(String notificationTicket,
                                  String messageId,
                                  MessageStatusType messageStatus)
                throws java.rmi.RemoteException {
      System.out.println("->Message Delivery Ack");
   }
```

When the listener is instantiated, the application enables notifications on certain criteria. In this case the criteria is that a notification is sent to MessageNotificationHandler when a new message arrives to a certain mailbox. The URL of the listener web service is supplied in order to inform WebLogic Network Gatekeeper where the service resides.

Listing 4-9 Open a mailbox and enable notifications

```
String mailboxTicket;
mailboxTicket = messaging.openMailbox(myMailbox, myMailboxPwd,
                                      spID+appID+appInstGroupID);
System.out.println("Mailbox Ticket retrieved");
notificationTicket = messaging.enableMessagingNotification
                           (receiveMessageWsdlUrl,
```

```
                                myMailbox,
                                myMailboxPwd,
                                notificationCriteria.NC_NEW_MESSAGE_ARRIVED,
                                serviceCode,
                                  spID+appID+appInstGroupID);
```

Send MMS

First, a handle to the Messaging service is retrieved, the security header is added to the call object, and the mailbox is opened as described in “Send SMS” on page 4-2. The

The contents of the MMS are sent as SOAP attachment in MIME format, consisting of several attachment parts. The method defineattAchmentPart described in Listing 4-17, “Define an attachment part,” on page 4-14. Each attachment part is added to the header of the object representing the call.

Listing 4-10 Creating two attachment parts.

```
int index = 1;

AttachmentPart ap = new AttachmentPart();

ap = defineAttachmentPart("file:../img/afile.jpg",
                          "image/jpeg",
                          "afile",
                            index++);

((org.apache.axis.client.Stub)sendMms).addAttachment(ap);

ap = defineAttachmentPart("file:../img/anotherfile.jpg",
                          "image/jpeg",
                          "anotherfile",
                            index++);

((org.apache.axis.client.Stub)messaging).addAttachment(ap);
```

The messaging properties are defined in the same manner as when sending an SMS, see "Send SMS" on page 4-2, and the additional message property MESSAGE_FORMAT, with the value MESSAGE_FORMAT_MM is defined for MMS Messages.

Listing 4-11

```
arrayofMessagingProperties[0] = new MessagingProperty();
arrayofMessagingProperties[0].setMessagingPropertyName
                                (MessagingPropertyName.MESSAGE_SENT_TO);
arrayofMessagingProperties[0].setValue(destAddress);
arrayofMessagingProperties[1] = new MessagingProperty();
 arrayofMessagingProperties[1].setMessagingPropertyName
                                (MessagingPropertyName.MESSAGE_SENT_FROM);
 arrayofMessagingProperties[1].setValue(origAddress);
 arrayofMessagingProperties[2] = new MessagingProperty();
 arrayofMessagingProperties[2].setMessagingPropertyName
                                 (MessagingPropertyName.MESSAGE_FORMAT);
 arrayofMessagingProperties[2].setValue
                                 (MessageFormatType.MESSAGE_FORMAT_MM);
```

When the attachment parts have been defined, added to the call object and the message properties have been defined, the MMS is sent. This method is very similar to the sendSMS method as described in "Define message properties and send the SMS" on page 4-3. It is also possible to retrieve the delivery status in the same way as described in "Get delivery status" on page 4-4. FInally, the mailbox should be closed in the same manner as described in "Close the mailbox" on page 4-3

Poll for new messages

An application can poll for new messages. A list of references to the unread messages are returned. The messages are retrieved using these references

The normal procedure initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the messaging interface, definition of the security header, and opening of a mailbox.

The message descriptions are returned in a MesageDescription[]. In the example, each message description is traversed. If a message is an SMS, the actual text is fetched via the messageID in the message description. If it is an MMS, the format is MESSAGE_FORMAT_TEXT. An MMS message requires a bit more processing to fetch the contents. The user-defined class GetMMsApphandler implements this functionality in the method getMms.

Listing 4-12 Get message descriptions of new messages in a mailbox

```
arrayofMessageDescription = messaging.listNewMessages
                                  (mailboxTicket,
                                   MailboxFolder.MESSAGE_MAILBOX_INBOX,
                                     spID+appID+appInstGroupID);

for (int i = 0; i < arrayofMessageDescription.length; i++) {

   System.out.println("<> ");

   System.out.println("Message ID    " +
                        arrayofMessageDescription[i].getMessageId());

   System.out.println("Message format " +
                        arrayofMessageDescription[i].getFormat());

   if (arrayofMessageDescription[i].getFormat().toString() ==
                                                    "MESSAGE_FORMAT_TEXT"){

      String messageText = messaging.getSMS
                              (mailboxTicket,
                               MailboxFolder.MESSAGE_MAILBOX_INBOX,
                               arrayofMessageDescription[i].getMessageId(),
                                 spID+appID+appInstGroupID );

      System.out.println("Message text " + messageText );

   }

   if (arrayofMessageDescription[i].getFormat().toString() ==
                                                    "MESSAGE_FORMAT_MM"){

      GetMmsAppHandler mmsAppHandler = new GetMmsAppHandler();
```

```
      mmsAppHandler.getMms( messagingService,
                            messaging,
                              mailboxTicket,
                            MailboxFolder.MESSAGE_MAILBOX_INBOX,
                            arrayofMessageDescription[i].getMessageId(),
                              spID+appID+appInstGroupID );
   }
}
```

Listing 4-13 Definition of method getMms

```
public void getMms( MessagingServiceLocator messagingService,
                    Messaging messaging, String mailboxTicket,
                    MailboxFolder mailboxFolder, String messageId,
                     String requester) {
```

When the getMms method is invoked, the method getMMS is invoked and the SOAP attachment is retrieved as described in "Get MMS and retrieve the SOAP attachment" on page 4-10.

Listing 4-14 Get MMS and retrieve the SOAP attachment

```
try {
   messaging.getMMS(mailboxTicket, mailboxFolder, messageId, requester);
} catch (Throwable e) {
   // Do some error processing
   System.out.println("Caught exception (get MMS): " + e.getMessage());
   e.printStackTrace();
}
try {
   // Get the context of the SOAP message
```

```
   MessageContext context =
                   messagingService.getCall().getMessageContext();
   // Get the last response message.
   org.apache.axis.Message reqMsg = context.getResponseMessage();
   // Get the SOAP attachmnents
   m_attachments = reqMsg.getAttachmentsImpl();
   System.out.println("Number of attachments: " +
                         m_attachments.getAttachmentCount());
} catch (Throwable e) {
   // Do some error processing
   System.out.println("Caught exception (getAttachments): " +
                         e.getMessage());
}
```

The content is retrieved from the attachment as described in "Retrieve the content from a SOAP attachment" on page 4-11. In this case, each attachment is saved under a unique filename.

Listing 4-15 Retrieve the content from a SOAP attachment

```
java.util.Collection c = m_attachments.getAttachments();
Iterator it = c.iterator();
// For each attachment
while( it.hasNext()){
   org.apache.axis.attachments.AttachmentPart p =
                   (org.apache.axis.attachments.AttachmentPart)it.next();
   javax.activation.DataHandler dh= p.getDataHandler();
   BufferedInputStream bis = new BufferedInputStream(dh.getInputStream());
   ByteArrayOutputStream bos = new ByteArrayOutputStream();
   while (bis.available() > 0) {
```

```
        bos.write(bis.read());
    }
    byte[] pmsg = bos.toByteArray();
    System.out.println("Message Length: "+pmsg.length);
    System.out.println("Content Type: "+p.getContentType());
    System.out.println("Content ID: "+p.getContentId());
    // Convert mime identifier to file extension
    String type = p.getContentType().substring
                                1+p.getContentType().lastIndexOf("/",
                                   p.getContentType().length()));
    // Save attachment as file
    FileOutputStream fos = new FileOutputStream("Message_" + messageId +
                                                " ID_" +
                                                 p.getContentId()+
                                                     "."+ type);
    fos.write(pmsg);
    fos.close();
}
```

Handling SOAP Attachments

When sending and receiving multimedia messages, the content is handled as attachments in MIME or DIME using SwA, SOAP with Attachments. This technique combines SOAP with MIME, allowing any arbitrary data to be included in a SOAP message.

An SwA message is identical with a normal SOAP message, but the header of the HTTP request contains a Content-Type tag of type “multipart/related”, and the attachment block(s) after the termination tag of the SOAP envelope.

Axis and Java Mail classes can be used to construct and deconstruct MIME/DIME SwA messages.

Encoding a multipart SOAP attachment

Listing 4-16, "Create an attachment," on page 4-13 gives an example on how to create an attachment and to add it to the SOAP header. Two attachment parts are created.

Listing 4-16 Create an attachment

```
SendMessageServiceLocator sendMmsService = new SendMessageServiceLocator();

java.net.URL endpoint = new java.net.URL(sendMmsWsdlUrl);

SendMessagePort sendMms = sendMmsService.getSendMessagePort(endpoint);

AttachmentPart ap = new AttachmentPart();

ap = defineAttachmentPart("file:../img/img1.jpg",
                          "image/jpeg",
                          "img1",
                            index++);

((org.apache.axis.client.Stub)sendMms).addAttachment(ap);

ap = defineAttachmentPart("file:../img/img2.jpg",
                          "image/jpeg",
                          "img2",
                            index++);

((org.apache.axis.client.Stub)sendMms).addAttachment(ap);
```

The method `defineAttachmentPart` is illustrated Listing 4-17, "Define an attachment part," on page 4-14. The method creates an attachment part. The method is invoked with the following parameters:

- `String mmsInfo`, the full URL to the attachment.
- `String contentType`, the mime type.
- `String contentId`, ID of attachment part, unique within the attachment.
- `int index`, ID of attachment part, unique within the attachment.

Listing 4-17 Define an attachment part

```
private AttachmentPart defineAttachmentPart(String mmsInfo,
                                            String contentType,
                                            String contentId,
                                               int index){
   AttachmentPart apPart = new AttachmentPart();
   try {
      URL fileurl = new URL(mmsInfo);
      BufferedInputStream bis =
                           new BufferedInputStream(fileurl.openStream());
       apPart.setContent( bis, contentType);
       apPart.setMimeHeader("Ordinal", String.valueOf(index));
      //reference the attachment by contentId.
       apPart.setContentId(contentId);
   } catch (Exception ex) {
      ex.printStackTrace();
   }
   return apPart;
}
```

Retrieving and Decoding a multipart SOAP attachment

In order to get a SOAP attachment, the response message is necessary since the SOAP attachment is returned in as an attachment in the SOAP header of the HTTP response. In Listing 4-18, “Get a response message,” on page 4-14, the response message is retrieved.

Listing 4-18 Get a response message

```
// Get the context of the SOAP message
```

```
MessageContext context = receiveMmsService.getCall().getMessageContext();
// Get the last response message.
org.apache.axis.Message reqMsg = context.getResponseMessage();
```

When a handle to the response message is retrieved, the SOAP attachments can be fetched.

Listing 4-19 Get the SOAP attachments

```
Attachments attachments = reqMsg.getAttachmentsImpl();
java.util.Collection c = attachments.getAttachments();
```

Each attachment, and each attachment part, is traversed and decoded. In the example the attachments are saved to file.

Listing 4-20 Extract the attachments

```
java.util.Collection c = attachments.getAttachments();
Iterator it = c.iterator();
// For each attachment
while( it.hasNext()){
   org.apache.axis.attachments.AttachmentPart p =
                    (org.apache.axis.attachments.AttachmentPart)it.next();
   javax.activation.DataHandler dh= p.getDataHandler();
   BufferedInputStream bis = new BufferedInputStream(dh.getInputStream());
   ByteArrayOutputStream bos = new ByteArrayOutputStream();
   while (bis.available() > 0) {
      bos.write(bis.read());
}
```

```
byte[] pmsg = bos.toByteArray();

System.out.println("Message Length: "+pmsg.length);

System.out.println("Content Type: "+p.getContentType());

System.out.println("Content ID: "+p.getContentId());

// Convert mime identifier to file extension

String type = p.getContentType().substring(
                                1+p.getContentType().lastIndexOf("/",
                                  p.getContentType().length()));

// Save attachment as file

FileOutputStream fos = new FileOutputStream("ContentID_"
                                            +p.getContentId()+ "."+
                                               type);

fos.write(pmsg);

fos.close();
```

Get the location of a mobile terminal

The position of a mobile terminal can be retrieved both synchronously and asynchronously. The returned position can be requested, and returned, as a geographical position, extend geographical position, and as geo-coded information. See "User location" on page 3-15.

In this example, the location is requested using a synchronous request.

The normal procedure initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the user location service retrieved, as shown in Listing 4-21, "Retrieve the user location interface," on page 4-16.

Listing 4-21 Retrieve the user location interface

```
userLocationService = new UserLocationServiceLocator();

java.net.URL endpoint = new java.net.URL(userLocationWsdlUrl);

userLocation = userLocationService.getUserLocation(endpoint);
```

The security header is added to the userLocation object, see below.

Listing 4-22 Add the security header to the userlocation object

```
((org.apache.axis.client.Stub)userLocation).setHeader(header);
```

The parameters are defined and the method for getting the position is invoked.

Listing 4-23 Get the location

```
String targetAddress = "tel:1234567";
String serviceCode = "cp_free";
String[] myAddresses = {targetAddress};
LocationResponseTime myRequestedResponseTime = new LocationResponseTime();
int mywaitTimeoutSeconds = 20;
myRequestedResponseTime.setResponseTimeIndicator(
                              LocationResponseTimeIndicator.LOW_DELAY);
LocationResult[] locationResultArray =
               userLocation.getLocationWait(myAddresses,
                                            mywaitTimeoutSeconds,
                                            serviceCode,
                                               spID+appID+appInstGroupID);
```

The result is returned as an array, with each element corresponding to an entry in myAddresses. Each element is traversed and the location information is fetched.The type of information retrieved, and how it is returned depends on the uncertainty shape given. The example illustrates three shapes; circle, sector, and circle arc stripe. The uncertainty shapes based on ellipses are handled in a similar manner.

Listing 4-24 Get the coordinates

```
for (int i = 0; i < locationResultArray.length; i++) {

   System.out.println("Terminal " +locationResultArray[i].getAddress());

   System.out.println("Latidute " +
                locationResultArray[i].getTheLocation().getLatitude());

   System.out.println("Longitude " +
                locationResultArray[i].getTheLocation().getLongitude());

    LocationUncertaintyShapeTypes locationUncertaintyShapeType =
                           locationResultArray[i].getTheLocation().
                             getShape().getLocationUncertaintyShapeType();

   // Handle different types of positioning info differently

   if (locationUncertaintyShapeType ==
        LocationUncertaintyShapeTypes.CIRCLE ) {

      LocationUncertaintyShapeCircle locationUncertaintyShapeCircle =
                   (LocationUncertaintyShapeCircle)locationResultArray[i].
                     getTheLocation().getShape().getValue();

      System.out.println("Uncertainty Shape Circle: radius " +
                            locationUncertaintyShapeCircle.getRadius());

   }

   else if (locationUncertaintyShapeType ==
             LocationUncertaintyShapeTypes.CIRCLE_SECTOR ) {

      LocationUncertaintyShapeCircleSector
      locationUncertaintyShapeCircleSector =
      (LocationUncertaintyShapeCircleSector)locationResultArray[i].
       getTheLocation().getShape().getValue();

       System.out.println("Uncertainty Shape Circle Sector: start angle:" +
           locationUncertaintyShapeCircleSector.getSegmentStartAngle());

       System.out.println("segment end angle:" +
           locationUncertaintyShapeCircleSector.getSegmentEndAngle());

       System.out.println("radius " +
           locationUncertaintyShapeCircleSector.getCircle().getRadius());
```

```
    }
    else if (locationUncertaintyShapeType ==
              LocationUncertaintyShapeTypes.CIRCLE_ARC_STRIPE ){

        LocationUncertaintyShapeCircleArcStripe
        locationUncertaintyShapeCircleArcStripe =
        (LocationUncertaintyShapeCircleArcStripe)locationResultArray[i].
        getTheLocation().getShape().getValue();

       LocationUncertaintyShapeCircleSector
       locationUncertaintyShapeCircleSector =
       (LocationUncertaintyShapeCircleSector)
       locationUncertaintyShapeCircleArcStripe.getCircleSector();

       System.out.println("Uncertainty Shape Circle Arc Stripe: " +
           "segment start angle:" +
            locationUncertaintyShapeCircleSector.getSegmentStartAngle());

       System.out.println("segment end angle:" +
          locationUncertaintyShapeCircleSector.getSegmentEndAngle());

       System.out.println("radius " +
            locationUncertaintyShapeCircleSector.getCircle().getRadius());

      System.out.println("inner radius " +
            locationUncertaintyShapeCircleArcStripe.getInnerRadius());

    }
    else{

       System.out.println("Uncertainty shape other than cirular.");

    }
}
```

Application-initiated messaging user interaction

In this example, a message is sent using the messaging user interaction interface and a reply of the message is taken care of by the application.

In the example, the reply is requested using a synchronous request.

The standard initial procedure as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. The messaging service capability is not used, instead the user interaction service is retrieved, as shown in Listing 4-25, "Retrieve the user interaction interface," on page 4-20.

Listing 4-25 Retrieve the user interaction interface

```
UserInteractionService userInteractionService =
   new UserInteractionServiceLocator();
java.net.URL endpoint = new java.net.URL(messagingUserInteractionWsdlUrl);
userInteraction = userInteractionService.
   getMessagingUserInteraction(endpoint);
```

The security header is added to the userInformation object.

Listing 4-26 Add the security header to the userInteraction object

```
((org.apache.axis.client.Stub)userInteraction).setHeader(header);
```

A user interaction session is created. The returned identifier (uiTicket) for the session is used in each subsequent call within the session. The parameter destAddress defines to which address a subsequent message shall be distributed. The format of the address must be in URI-format (tel:<address>).

Listing 4-27 Create a user interaction session

```
uiTicket = userInteraction.createUI(destAddress);
```

The type of information and the information to send to the telephony terminal is defined. In Listing 4-28, "Define the data to send to the terminal," on page 4-21, the type is UI_INFO_DATA, and the information is a string. No data encoding scheme is defined.

Listing 4-28 Define the data to send to the terminal

```
UserInformation info = new UserInformation();
info.setUserInformationType(UserInformationType.UI_INFO_DATA);
UserInformationData userInformationData = new UserInformationData();
userInformationData.setInfoData("Do you wish to proceed? Y or N");
userInformationData.setInfoDataEncodingScheme("");
info.setValue(userInformationData);
```

Finally, the information is sent to the terminal as described in Listing 4-29, "Send the data and wait for a reply," on page 4-21. In this case a synchronous method call is used. The ticket identifying the session is supplied, together with data defined in Listing 4-28, "Define the data to send to the terminal," on page 4-21.

The parameters minLength, maxLength, endSequence, startTimeOut, and language are not used when using the message based user interaction service. The parameter waitTimeOut defines, in seconds, for how long the synchronous request shall wait before an answer arrives from the terminal the information was sent to. If this time is exceeded, a UIException is thrown.

The parameters serviceCode and requesterID are defined by the operator.

The answer collected is returned in the string collectedInfo.

Listing 4-29 Send the data and wait for a reply

```
int minLength = 0;
int maxLength = 0;
String endSequence = "";
int startTimeOut = 0;
```

```
int interCharTimeout = 0;
String language = "";
int waitTimeOut = 180;
String serviceCode = "A service code";
String requesterID = "A requester ID";
String collectedInfo = userInteraction.sendInfoAndCollectWait(
                                            uiTicket,
                                            info,
                                            minLength,
                                            maxLength,
                                            endSequence,
                                            startTimeOut,
                                            interCharTimeout,
                                            language,
                                            waitTimeOut,
                                            serviceCode,
                                                requesterID);
```

When the user interaction session is over, it is closed and the application logs out.

Listing 4-30 Close user interaction session and logout

```
userInteraction.closeUI(uiTicket);
myLoginSession.logout(sessionId);
```

Network-initiated messaging user interaction

In this example, a message is sent from a terminal. The incoming message arrives to the application via the network-initiated messaging user interaction listener interface. When the message arrives, the application sends a response back to the terminal via the messaging user interaction Web Service.

Notifications on network-initiated user interaction sessions are sent asynchronously from WebLogic Network Gatekeeper. This means that the application must implement a Web Service. The initial thing is to start the Web Service server and deploy the implementation of the Web service into the server. The deployment is performed using a deployment descriptor that is automatically generated when the Web Service java skeletons are generated. The deployment descriptor (deploy.wsdd) is modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 4-33, "Declaration of the class implementing the listener interface," on page 4-24 and Listing 4-34, "Implementation of processUINotification," on page 4-25. The class is based on the auto-generated class UserInteractionNetworkListenerSoapBindingImpl.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. The messaging service capability is not used, instead is the User Interaction service retrieved, as shown in Listing 4-25, "Retrieve the user interaction interface," on page 4-20 and the security header is added as described in Listing 4-26, "Add the security header to the userInteraction object," on page 4-20.

First, the Simple Axis server is started and the WSDD file describing the Web service is deployed as outlined in Listing 4-7, "Start SimpleAxis server," on page 4-4.

When the Web Service is deployed, its endpoint (URL) must be registered in WebLogic Network Gatekeeper as outlined in Listing 4-31, "Registering the listener for network initiated user interaction sessions," on page 4-24. The listener is registered on the object representing the user interaction Web Service.

The URL is registered together with notification criteria. All criteria must be fulfilled in order to distribute a notification from WebLogic Network Gatekeeper to the application. The criteria is expressed in the parameters aPartyAddressExpression, bPartyAddressExpression, and userInteractionCode.

The address expressions allows for wildcards (* and ?). The format of the addresses must be in URI-format (tel:<address>). The parameter userInteractionCode is defined by the operator.

The parameters serviceCode and requesterID are defined by the operator.

An ID for the notification listener is returned. This ID is supplied in every notification to the listener interface to correlate the listener with a notification. It is also used when the notification listener is removed.

Listing 4-31 Registering the listener for network initiated user interaction sessions

```
String listenerID = userInteraction.addNetworkUIListener(
                            notificationWsdlUrl,
                            aPartyAddressExpression,
                            bPartyAddressExpression,
                            userInteractionCode,
                            serviceCode,
                              requesterID);
```

When the application is not interested in receiving notifications, it de-registers the notification listener as described in Listing 4-32, “Removing the notification listener,” on page 4-24.

Listing 4-32 Removing the notification listener

```
userInteraction.removeNetworkUIListener(listenerID, requesterID);
```

The class implementing the network-initiated user interaction interface is declared as below.

Listing 4-33 Declaration of the class implementing the listener interface

```
public class MessagingUINwInitListener implements
   UserInteractionNetworkListener {
```

The method processUINotification, as described in Listing 4-34, “Implementation of processUINotification,” on page 4-25, is invoked when a user interaction session has been initiated from the network via WebLogic Network Gatekeeper.

In the example below, a reply is sent back to the party initiating the user interaction session using the synchronous method sendInfoWait. The ticket identifying the user interaction session (uiTicket) is created in WebLogic Network Gatekeeper, and the reply must be sent using the same ticket. The call to sendInfoWait is performed on the object representing the user interaction Web

Service. Note that the call to sendInfoWait is only used to illustrate that a response should be performed using the object representing the user interaction Web service, and that the uiTicket used is the ticket retrieved as a parameter in processUINotification. In a live production environment, a separate thread should be created in combination with a call to the asynchronous method sendInfo.

Listing 4-34 Implementation of processUINotification

```
public void processUINotification(String notificationTicket,
                                  String uiTicket,
                                  String originator,
                                  String destination,
                                  String userInteractionCode,
                                  UIEventDataTypeCode dataTypeCode,
                                  String dataString)
   throws java.rmi.RemoteException {
   System.out.println("Got a processUINotification ");
   System.out.println(" Notification Ticket " + notificationTicket);
   System.out.println(" UI ticket " + uiTicket);
   System.out.println(" Originator " + originator);
   System.out.println(" Destination " + destination);
   System.out.println(" UI Code " + userInteractionCode);
   System.out.println(" DataTypeCode " + dataTypeCode.getValue());
   System.out.println(" DataTypeCode " + dataString);
   UserInformation info = new UserInformation();
   info.setUserInformationType(UserInformationType.UI_INFO_DATA);
   UserInformationData userInformationData = new UserInformationData();
   userInformationData.setInfoData("A reply");
   info.setValue(userInformationData);
   userInformationData.setInfoDataEncodingScheme("");
```

```
    MessagingUINwInitAppHandler.userInteraction.sendInfoWait(uiTicket,
                                                              info,
                                                              0,
                                                              "",
                                                              15,
                                                             "cp_free,
                                                                  "an ID");
}
```

Setting up an application-initiated call

A call can be set up between two or more parties from an application using the Call control Web Service. The methods relevant for Call control are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

In the example below, a call between two parties is set up from the application.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the Call control service retrieved, as shown in Listing 4-35, "Retrieve the call control interface," on page 4-26.

Listing 4-35 Retrieve the call control interface

```
CallControlService callControlService = new CallControlServiceLocator();
java.net.URL endpoint = new java.net.URL(callControlWsdlUrl);
callControl = callControlService.getCallControl(endpoint);
```

The security header is added to the userLocation object, see below.

Listing 4-36 Add the security header to the callControl object

```
((org.apache.axis.client.Stub)callControl).setHeader(header);
```

First, the data about the call, such as originator is defined, as defined in Listing 4-37, "Defining the originator of the call and set up the first call leg," on page 4-27. Since the synchronous method is used, a timeout value is supplied. A callTicket, representing the call is returned. The method returns when the originator has gone off-hook, and the call processing continues as outlined in Listing 4-38, "Setting up the second call leg," on page 4-28.

Listing 4-37 Defining the originator of the call and set up the first call leg

```
String callTicket ="";
String originator = "tel:1234567";
int timeout = 10;
String requesterID = "An ID";
String serviceCode = "cp_free";
try {
   callTicket = callControl.createCall(originator,
                                       timeout,
                                       serviceCode,
                                          requesterID);
   System.out.println("Originator for call created: " + originator);
}
catch (CallSetupException e) {
   System.out.println("CallSetupException");
   System.out.println("Caught exception: " + e.getMessage());
}
catch (CallException e) {
   System.out.println("CallException");
   System.out.println("Caught exception: " + e.getMessage());
}
```

```
catch (GeneralException e) {
   System.out.println("GeneralException");
   System.out.println("Caught exception: " + e.getMessage());
}
catch (Throwable e) {
   System.out.println("Other exception");
   System.out.println("Caught exception: " + e.getMessage());
}
```

When the originator of the call and data about the call is created, the second call leg is set up and the second participant in the call is added. The callTicket retrieved when the call was created is used.

Listing 4-38 Setting up the second call leg

```
String participant = "tel:2345678";
try {
   callControl.addParticipantWait(callTicket,
                                  participant,
                                     timeout);
   System.out.println("Participant added: " + participant);
}
catch (CallSetupException e) {
   System.out.println("CallSetupException");
   System.out.println("Caught exception: " + e.getMessage());
}
catch (CallException e) {
   System.out.println("CallException");
   System.out.println("Caught exception: " + e.getMessage());
```

```
}
catch (GeneralException e) {
   System.out.println("GeneralException ");
   System.out.println("Caught exception: " + e.getMessage());
}
catch (Throwable e) {
   System.out.println("Other exception");
   System.out.println("Caught exception: " + e.getMessage());
}
```

When the second call leg has been set up, the call is deassigned to the network as outlined in Listing 4-39, "Deassign the call to the network," on page 4-29. From this point the call is no longer controlled by the application, although it may be supervised.

Listing 4-39 Deassign the call to the network

```
try {
   callControl.deassign(callTicket);
   System.out.println("Call deassigned");
}
catch (CallException e) {
   System.out.println("CallException");
   System.out.println("Caught exception: " + e.getMessage());
}
catch (GeneralException e) {
   System.out.println("GeneralException");
   System.out.println("Caught exception: " + e.getMessage());
}
```

```
catch (Throwable e) {
   System.out.println("Other exception");
   System.out.println("Caught exception: " + e.getMessage());
}
```

Network-initiated call control

In this example, a call attempt is made from a terminal. A notification about the call setup attempt arrives to the application via the network-initiated call control listener interface. When the notification arrives, the application sends a response back to the terminal via the Call control Web Service.

Notifications on network-initiated call attempts are sent asynchronously from WebLogic Network Gatekeeper. This means that the application must implement a Web Service. The initial thing is to start the Web Service server and deploy the implementation of the Web service into the server. The deployment is performed using a deployment descriptor that is automatically generated when the Web Service java skeletons are generated. The deployment descriptor (deploy.wsdd) is modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 4-33, "Declaration of the class implementing the listener interface," on page 4-24 and Listing 4-34, "Implementation of processUINotification," on page 4-25. The class is based on the auto-generated class UserInteractionNetworkListenerSoapBindingImpl.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. The messaging service capability is not used, instead is the User Interaction service retrieved, as shown in Listing 4-25, "Retrieve the user interaction interface," on page 4-20 and the security header is added as described in Listing 4-26, "Add the security header to the userInteraction object," on page 4-20.

First, the Simple Axis server is started and the WSDD file describing the Web service is deployed as outlined in Listing 4-7, "Start SimpleAxis server," on page 4-4.

When the Web Service is deployed, its endpoint (URL) must be registered in WebLogic Network Gatekeeper as outlined in Listing 4-31, "Registering the listener for network initiated user interaction sessions," on page 4-24. The listener is registered on the object representing the user interaction Web Service.

The URL is registered together with notification criteria. All criteria must be fulfilled in order to distribute a notification from WebLogic Network Gatekeeper to the application. The criteria is

expressed in the parameters aPartyAddressExpression, bPartyAddressExpression, and userInteractionCode.

The address expressions allows for wildcards (* and ?). The format of the addresses must be in URI-format (tel:<address>). The parameter userInteractionCode is defined by the operator.

The parameters serviceCode and requesterID are defined by the operator.

An ID for the notification listener is returned. This ID is supplied in every notification to the listener interface to correlate the listener with a notification. It is also used when the notification listener is removed.

Listing 4-40 Registering the listener for network initiated call control

```
java.lang.String aPartyAddressExpression = "tel:*";
java.lang.String bPartyAddressExpression = "tel:*";
CallEventCriteria[] eventCriteria = new CallEventCriteria[1];
System.out.println("Created Call event criteria array");
eventCriteria[0] = new CallEventCriteria();
eventCriteria[0].setEvent(NetworkCallEvent.ADDRESS_ANALYSED);
eventCriteria[0].setMonitorMode(CallMonitorMode.INTERRUPT);
String listenerID;
listenerID = callControl.addNetworkCallListener(notificationWsdlUrl,
                                                aPartyAddressExpression,
                                                bPartyAddressExpression,
                                                eventCriteria,
                                                serviceCode,
                                                requesterID);
```

When the application is not interested in receiving notifications, it de-registers the notification listener as described in Listing 4-32, “Removing the notification listener,” on page 4-24.

Listing 4-41 Removing the notification listener

```
callControl.removeNetworkCallListener(listenerID)
```

The class implementing the network-initiated call control interface is declared as below.

Listing 4-42 Declaration of the class implementing the listener interface

```
public class CallNwInitListener implements NetworkCallListener {
```

The method processCall, as described in Listing 4-43, “Implementation of processCall,” on page 4-32, is invoked when the monitor mode of the call is INTERRUPTED, that is the call is owned by the application and it can be manipulated. The method processNotification, as described in Listing 4-44, “Implementation of processNotification,” on page 4-33, is invoked when the monitor mode of the call is NOTIFY, that is the call is not owned by the application and it can only be monitored and not be manipulated by the application. The monitor mode is defined when the listener is registered, see Listing 4-40, “Registering the listener for network initiated call control,” on page 4-31.

processCall receives notifications on calls in monitor mode INTERRUPT. The ticket identifying the call (callTicket) is created in WebLogic Network Gatekeeper, and subsequent actions on the call must be performed using the same ticket.

Listing 4-43 Implementation of processCall

```
public void processCall(String listenerTicket,
                        String callTicket,
                        String originator,
                        String participant,
                        NetworkCallEvent event)
   throws java.rmi.RemoteException {

   System.out.println("Got a processCall ");

   System.out.println(" Call Ticket " + callTicket);
```

```
   System.out.println(" Originator " + originator);
   System.out.println(" Participant " + participant);
   System.out.println(" Network Call Event " + event.getValue());
}
```

processNotification receives notifications on calls in monitor mode NOTIFY

Listing 4-44 Implementation of processNotification

```
public void processNotification(String listenerTicket,
                                String originator,
                                String participant,
                                NetworkCallEvent event)
   throws java.rmi.RemoteException {
   System.out.println("Got a processNotification ");
   System.out.println(" Listener Ticket " + listenerTicket);
   System.out.println(" Originator " + originator);
   System.out.println(" Participant " + participant);
   System.out.println(" Network Call Event " + event.getValue());
}
```

The parameter event holds information on the type of event in the network that resulted in the call to the network-initiated call control interface.

When the application is no longer interested in receiving notifications, the listener must be removed as outlined in Listing 4-45, “Removing the listener,” on page 4-33.

Listing 4-45 Removing the listener

```
callControl.removeNetworkCallListener(listenerID);
```

Handling call-based user interaction

Call-based user interaction sessions use resources such as voice-prompt machines (IVRs) in the telecom network. When using call-based user interaction these resources are addressed from the application.

Call-based user interaction is always used together with the Call control Web Service, since the user interaction part takes advantage of existing call legs, created by the Call control Web Service, and routes the call legs to the IVRS.

The methods relevant for Call user interaction are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

In the example below, an existing call is created as described in "Setting up an application-initiated call" on page 4-26.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the Call user interaction service retrieved, as shown in Listing 4-46, "Retrieve the call user interaction interface," on page 4-34.

Listing 4-46 Retrieve the call user interaction interface

```
CallUserInteractionService callUserInteractionService =
   new CallUserInteractionServiceLocator();

java.net.URL endpoint =
   new java.net.URL(callUserInteractionWsdlUrl);

callUserInteraction =
   callUserInteractionService.getCallUserInteraction(endpoint);
```

The security header is added to the callUserInteraction object, see below.

Listing 4-47 Add the security header to the callUserInteraction object

```
((org.apache.axis.client.Stub)callUserInteraction).setHeader(header);
```

Assuming that a call leg is setup to a party, and that the ticket identifying the call (callTicket) is available, a call user interaction session is created as described in.Listing 4-48, “Create the call user interaction session,” on page 4-35. For information on how to retrieve a call ticket, see Listing 4-37, “Defining the originator of the call and set up the first call leg,” on page 4-27. The session is also created with a participant, or originator. Using “tel:*” when creating the sessions means that all call legs will be involved in the session. That is, the party in the existing call that shall interact with the IVR.

Listing 4-48 Create the call user interaction session

```
String originator = "tel:1234567";

callUiTicket = callUserInteraction.createCallUserInteraction(callTicket,
                                                              originator);
```

When the callUITicket identifying the session is available, the application can set up a connection to the IVR. First, data about the IVR, such as type of identifier supplied, and the relevant data for the type of identifier. In Listing 4-49, “Defining data about the IVR,” on page 4-35, the type of identifier for the announcement to be played is a UI_INFO_ID, which indicates that the accompanying data is an ID of the resource to use. The actual ID is also supplied.

Listing 4-49 Defining data about the IVR

```
UserInformation info = new UserInformation();

info.setUserInformationType(UserInformationType.UI_INFO_ID);

Integer informationID = new Integer(132);

info.setValue((java.lang.Object)informationID);
```

The synchronous method sendInfoAndCollectWait is invoked as described in Listing 4-50, “Setting up the user interaction dialogue,” on page 4-36. This method sends information to the IVR on which an announcement to be played, and also instructs the IVR to collect input from the terminal, for example via DTMF. Since the synchronous method is used, a timeout value for the whole dialogue is supplied in the parameter waitTimeoutSeconds. Other time-out parameters are also supplied, together with information on maximum and minimum length of the input, and an optional end sequence if variable length input is used. A typical example of an end sequence is a hash mark (#). A string, representing the input retrieved from the terminal is returned. Which input parameters that shall be used is dependant on the functionality available in the IVR used.

Listing 4-50 Setting up the user interaction dialogue

```
int minimumLength = 0;

int maximumLength = 100;

String endSequence = "#";

int startTimeoutSeconds = 20;

int interCharTimeoutSeconds = 5;

int waitTimeoutSeconds = 20;

String language = "EN";

System.out.println("Calling sendInfoAndCollectWait");

System.out.println("callUiTicket: " + callUiTicket);

System.out.println("info.getUserInformationType(): " +
                    info.getUserInformationType();

System.out.println("info.getValue(): " + info.getValue());

collectedInfo = callUserInteraction.sendInfoAndCollectWait(callUiTicket,
                                                    info,
                                                    minimumLength,
                                                    maximumLength,
                                                    endSequence,
                                                    startTimeoutSeconds,
                                                    interCharTimeoutSeconds,
```

```
                                                    language,
                                                    waitTimeoutSeconds,
                                                    serviceCode,
                                                       requesterID);
System.out.println("InformationCollected from User " + collectedInfo);
```

Finally, the user interaction session between the IVR and the terminal is closed as outlined in Listing 4-51, “Close the user interaction session,” on page 4-37.

Listing 4-51 Close the user interaction session

```
callUserInteraction.close(callUiTicket);
```

Handling subscriber data

The Subscriber profile Web service allows for setting and retrieving data related to subscribers. The methods relevant for Subscriber Profile are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

In the example below, data about a subscriber is added and retrieved.

The normal initial procedure is used as described in “Send SMS” on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the Subscriber profile service retrieved, as shown in Listing 4-52, “Retrieve the subscriber profile service,” on page 4-37.

Listing 4-52 Retrieve the subscriber profile service

```
SubscriberProfileService subscriberProfileService = new
SubscriberProfileServiceLocator();

java.net.URL endpoint = new java.net.URL(subscriberProfileWsdlUrl);

subscriberProfile = subscriberProfileService.
                     getSubscriberProfile(endpoint);
```

The security header is added to the subscriberProfile object, see below.

Listing 4-53 Add the security header to the subscriberProfile object

```
(((org.apache.axis.client.Stub)subscriberProfile).setHeader(header);
```

First, the data to be set is defined. The data is defined as name-value pairs, as outlined in Listing 4-54, “Define and store the data,” on page 4-38. The data to be set in this example are Street address, payment method to use and if the terminal supports MMS. The data is keyed on the parameter address.

Listing 4-54 Define and store the data

```
Property[] properties = new Property[3];
properties[0] = new Property();
properties[0].setPropertyType(PropertyTypes.ADDRESS);
properties[0].setAddress("Elm Street 32, Dodge City");
properties[1] = new Property();
properties[1].setPropertyType(PropertyTypes.MMS_ENABLED_TERMINAL);
java.lang.Boolean mmsEnabledTerminal = new java.lang.Boolean(true);
properties[1].setMmsEnabledTerminal(mmsEnabledTerminal);
properties[2] = new Property();
properties[2].setPropertyType(PropertyTypes.PAYMENT_METHOD);
PaymentMethod paymentMethod = new PaymentMethod();
paymentMethod.setPaymentType(PaymentType.INVOICE);
Short invoicenumber = new Short((short()1);
paymentMethod.setValue(invoicenumber);
properties[2].setPaymentMethod(paymentMethod);
String address = "tel:12345678";
```

```
int waitTimeoutSeconds = 10;

System.out.println("About to set Subscriber Profile " );

subscriberProfile.setSubscriberPropertyWait(address,
                                            properties,
                                            waitTimeoutSeconds,
                                            serviceCode,
                                               requesterID);
```

When data shall be retrieved using the subscriber profile database, the same type of mechanism applies as when storing data. A set of name-value pairs representing the data to be fetched is defined. The set of name-value pairs are assembled in an array and the array is a parameter in the method call.

The data is retrieved in an array, also as name-value pairs.

Listing 4-55 Retrieve data

```
PropertyTypes[] propertyTypes = new PropertyTypes[2];

propertyTypes[0] = PropertyTypes.ADDRESS;

propertyTypes[1] = PropertyTypes.MMS_ENABLED_TERMINAL;

Property[] someProperties;

someProperties = subscriberProfile.getSubscriberPropertyWait(
                                              address,
                                              propertyTypes,
                                              waitTimeoutSeconds,
                                              serviceCode,
                                                 requesterID);

for (int i = 0; i<someProperties.length; i++ ) {

   if (someProperties[i].getPropertyType() ==
   PropertyTypes.MMS_ENABLED_TERMINAL) {

      System.out.println("MMS Enabled Terminal: "+
      someProperties[i].getMmsEnabledTerminal());
```

```
    }
    if (someProperties[i].getPropertyType() == PropertyTypes.ADDRESS) {
       System.out.println("Address : " + someProperties[i].getAddress());
    }
}
```

Getting the status of a terminal

The User status Web service allows for getting the status of one or more terminals. The methods relevant for User Status are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

In the example below, the status of one single terminal is retrieved.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the User status service retrieved, as shown in Listing 4-56, "Retrieve the user status service," on page 4-40.

Listing 4-56 Retrieve the user status service

```
UserStatusService userStatusService = new UserStatusServiceLocator();
java.net.URL endpoint = new java.net.URL(userStatusWsdlUrl);
userStatus = userStatusService.getUserStatus(endpoint);
```

The security header is added to the userStatus object, see below.

Listing 4-57 Add the security header to the userStatus object

```
(((org.apache.axis.client.Stub)userStatus).setHeader(header);
```

The status of several terminals can be retrieved in one single method invocation. In this case only one status request is performed. Since an synchronous request is used, a timeout value is defined.

Listing 4-58 Retrieve status information

```
String [] addresses;

addresses = new String[1];

String user = "tel:1234567";

addresses[0] = user;

int waitTimeoutSeconds = 15;

StatusResult[] statusResult = userStatus.getStatusWait(
                                                    addresses,
                                                    waitTimeoutSeconds,
                                                    serviceCode,
                                                        requesterID);
```

The status result is returned in an array, with one entry per terminal as outlined in Listing 4-59, “Traverse returned data,” on page 4-41. Not only is the status of the terminal retrieved, but also the outcome of the actual status request and the type of terminal if this information is reported from the network.

Listing 4-59 Traverse returned data

```
for (int i = 0; i<statusResult.length; i++ ) {

   System.out.println("User : "+ statusResult[i].getAddress());

   System.out.println("Status of status request : "+
   statusResult[i].getReqStatus().getValue());

   System.out.println("Status of terminal: "+
   statusResult[i].getUserStatus().getAStatusIndicator());

   System.out.println("Terminal type: "+
      statusResult[i].getUserStatus().getATerminalType().getValue());
```

```
}
```

Charge based on content

The Charging Web service allows for reserving amounts and volumes from an end-users account, and to debit and credit the reservations. Direct debit and credit is also supported. The methods relevant for Charging are described in API Description Extended Web Services for WebLogic Network Gatekeeper.

In the example below, a charging session is created, and a reservation is made. An amount is debited, and the session is queried about the amount left in the reservation.

The normal initial procedure is used as described in "Send SMS" on page 4-2; login, retrieval of the interface, and definition of the security header is used. Naturally, the messaging service capability is not used, instead is the Content based charging service retrieved, as shown in Listing 4-60, "Retrieve the charging service," on page 4-42.

Listing 4-60 Retrieve the charging service

```
ContentBasedChargingService contentBasedChargingService = new
ContentBasedChargingServiceLocator();

java.net.URL endpoint = new java.net.URL(ContentBasedChargingWsdlUrl);

contentBasedCharging = contentBasedChargingService.
                        getContentBasedCharging(endpoint);
```

The security header is added to the contentBasedCharging object, see below.

Listing 4-61 Add the security header to the contentBasedCharging object

```
(((org.apache.axis.client.Stub)contentBasedCharging).setHeader(header);
```

First, a charging session is created as outlined in Listing 4-62, “Create charging session,” on page 4-43. All subsequent charging operations are performed in this session. The session is identified by a session ID, holding data such as charging session ticket. The session identifier is returned when the session is created. The session is created with the address of the party to charge, given in the parameter address. Other parameters are provided by the operator.

Listing 4-62 Create charging session

```
String merchantId ="merchant_id";

String address ="tel:462222222";

CorrelationID corrId = new CorrelationID();

String requesterId = "Requester ID";

int correlation = 1;

int corrType = 1;

corrId.setCorrelation(correlation);

corrId.setCorrType(corrType);

cSessionID = contentBasedCharging.createChargingSession(merchantId,
                                                        accountId,
                                                        address,
                                                        corrId,
                                                        serviceCode,
                                                            requesterId);
```

An amount is reserved from the party to charge’s account as outlined in Listing 4-63, “Reserve amount,” on page 4-44. Meta data, such as currency is also given. A request number is used in all charging operations. All charging operations take a request number as input parameter. Since this reservation is the first, the initial request number is fetched from the charging session identifier. The charging ticket is also fetched from the charging session identifier. The request number to be used in the next operation in the session is returned.

Listing 4-63 Reserve amount

```
float amountReserve = 2;
String currency = "SEK";
String description = "A descriptive text";
nextReqNo = contentBasedCharging.reserveAmountWait(
                                 cSessionID.getChargingTicket(),
                                 amountReserve,
                                 currency,
                                 description,
                                    cSessionID.getInitialRequestNumber()) ;
```

An amount is debited from the reservation as outlined in Listing 4-64, “Debit amount,” on page 4-44. When setting the parameter releaseAmount to True, the reservation is released, and hence is the reserved amount zero after this reservation, although the reservation was on 2 SEK and the debited amount was 1 SEK.

Listing 4-64 Debit amount

```
float amountDebit = 1;
boolean releaseAmount = true;
nextReqNo = contentBasedCharging.debitAmountWait(
                                   cSessionID.getChargingTicket(),
                                               amountDebit,
                                                currency,
                                                description,
                                                nextReqNo,
                                                    releaseAmount);
```

The amount left in the reservation can be checked as outlined in Listing 4-65, “Check amount left in reservation,” on page 4-45.

Listing 4-65 Check amount left in reservation

```
amountLeft = contentBasedCharging.getAmountLeftWait(
                                        cSessionID.getChargingTicket());
```

The charging session is terminated as outlined in Listing 4-66, “Close charging session,” on page 4-45.

Listing 4-66 Close charging session

```
contentBasedCharging.close(cSessionID.getChargingTicket());
```

APPENDIX A

References

API Description Extended Web Services for WebLogic Network Gatekeeper

Apache Axis, http://ws.apache.org/axis/

J2SE SDK, http://java.sun.com

JavaMail, http://java.sun.com