Oracle® WebLogic Server SIP Container

Developer's Guide

11*g* Release 1 (11.1.1)

E15461-01

October 2009

ORACLE®

Oracle WebLogic Server SIP Container Developer's Guide, 11*g* Release 1 (11.1.1)

E15461-01

Copyright © 2006, 2009, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

Contents

Preface

This preface contains the following sections:

Audience

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

Documentation Accessibility

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

Accessibility of Code Examples in Documentation

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

Accessibility of Links to External Web Sites in Documentation

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

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request

process. Information about TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html, and a list of phone numbers is available at http://www.fcc.gov/cgb/dro/trsphonebk.html.

Related Documents

For more information, see the following documents in the Oracle WebLogic Server SIP Container set:

- *Oracle WebLogic Server SIP Container Administrator's Guide*
- *Oracle WebLogic Server SIP Container Installation Guide*

Conventions

The following text conventions are used in this document:

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

Part I

Introduction

This part contains introductory information.

Part I contains the following chapter:

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

1

Overview of SIP Servlet Application Development

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

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

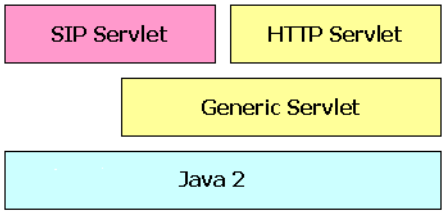
1.1 What is a SIP Servlet?

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

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

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

Figure 1–1 Servlet API and SIP Servlet API



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

Example 1–1 SimpleSIPServlet.java

```
package oracle.example.simple;
import java.io.IOException;
```

```
import javax.servlet.*;
import javax.servlet.sip.*;

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

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

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

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

1.2 SIP Servlets Versus HTTP Servlets

This section describes differences between SIP Servlets and HTTP Servlets.

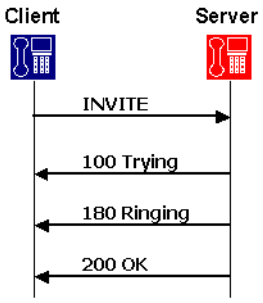
1.2.1 Multiple Responses

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

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

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

Figure 1–2 Example of Request and Response in SIP



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

Currently, SIP Servlet defines the following doXxx methods:

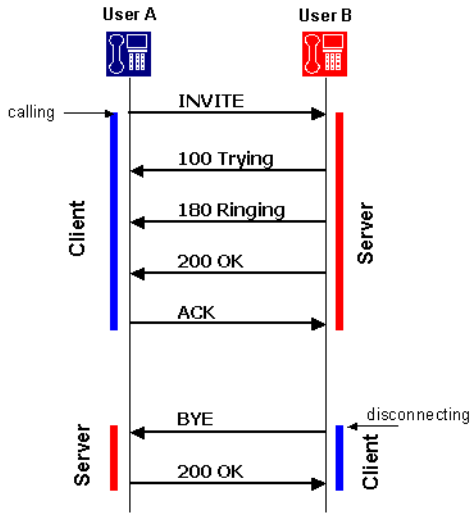
```
protected void doInvite(SipServletRequest req);
protected void doAck(SipServletRequest req);
protected void doOptions(SipServletRequest req);
protected void doBye(SipServletRequest req);
protected void doCancel(SipServletRequest req);
protected void doSubscribe(SipServletRequest req);
protected void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

1.2.2 Receiving Responses

One of the major features of SIP is that roles of a client and server are not fixed. In HTTP, web browsers always send HTTP requests and receive HTTP responses; they never receive HTTP requests and send HTTP responses. In SIP, however, each terminal must have functions of both a client and server.

In Figure 1–3, both SIP phones must call to one another and disconnect the call.

Figure 1–3 Relationship between Client and Server in SIP



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

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

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

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

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

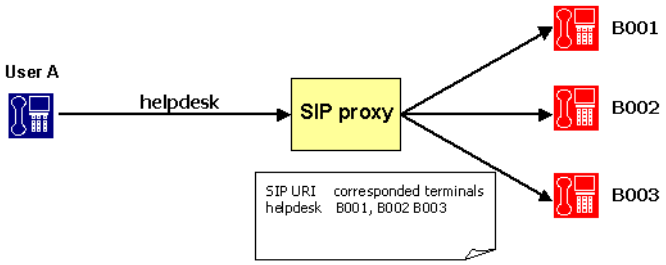
Also, SIP Servlet allows applications to explicitly create requests. Using these functions, SIP servlets can not only wait for requests as a server (UAS), but also send requests as a client (UAC).

1.2.3 Proxy Functions

Another function that is different from the HTTP protocol is "forking." Forking is a process of proxying one request to multiple servers simultaneously (or sequentially)

and used when multiple terminals (operators) are associated with one telephone number (such as in a call center).

Figure 1–4 Proxy Forking

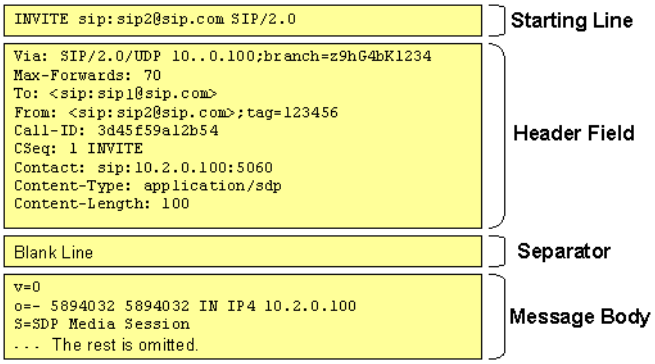


SIP Servlet provides a utility to proxy SIP requests for applications that have proxy functions.

1.2.4 Message Body

As shown in Figure 1–5, the structure of SIP messages is the same as HTTP.

Figure 1–5 SIP Message Example



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

1.2.4.1 Servlet Request

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

1.2.4.2 Servlet Response

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

```
void resetBuffer()
void flushBuffer()
```

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

In SIP, contents stored in the body include:

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

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

1.2.4.3 SipServletMessage

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

1.2.5 Role of a Servlet Container

The following sections describe major functions provided by Oracle WebLogic Server SIP Container as a SIP servlet container:

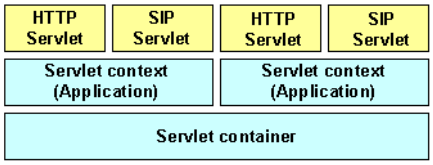
- Application Management—Describes functions such as application management by servlet context, lifecycle management of servlets, application initialization by deployment descriptors.
- SIP Messaging—Describes functions of parsing incoming SIP messages and delivering to appropriate SIP servlets, sending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

1.2.5.1 Application Management

Like HTTP servlet containers, SIP servlet containers manage applications by servlet context (see Figure 1–6). Servlet contexts (applications) are normally archived in a WAR format and deployed in each application server.

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

Figure 1–6 Servlet Container and Servlet Context



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

Oracle WebLogic Server SIP Container can deploy applications using the same method as the application server you use as the platform. However, if you deploy applications including SIP servlets, you need a SIP specific deployment descriptor (sip.xml) defined by SIP servlets. Table 1–1 shows the file structure of a general converged SIP and Web application.

Table 1–1 File Structure Example of Application

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

Information specified in the sip.xml file is similar to that in the web.xml except <servlet-mapping> setting that is different from HTTP servlets. In HTTP you specify a servlet associated with the file name portion of URL. But SIP has no concept of the file name. You set filter conditions using URI or the header field of a SIP request. Example 1–2 shows that a SIP servlet called "register" is assigned all REGISTER methods.

Example 1–2 Filter Condition Example of sip.xml

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

Once deployed, lifecycle of the servlet context is maintained by the servlet container. Although the servlet context is normally started and shutdown when the server is

started and shut down, the system administrator can explicitly start, stop, and reload the servlet context.

1.2.5.2 SIP Messaging

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

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

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

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

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

  Note: Filtering should be done carefully. In Oracle WebLogic Server SIP Container, when the received SIP message matches multiple SIP servlets, it is delivered only to one SIP servlet.

  The use of additional criteria such as request parameters can be used to direct a request to a servlet.

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

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

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

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

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

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

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

Table 1–2 Session-Related Terminology

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

Oracle WebLogic Server SIP Container automatically executes the following response and retransmission processes:

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

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

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

In SIP, however, since important information about message delivery exists in some fields, these headers cannot be changed by applications. Headers that can not be

changed by SIP servlets are called "system headers." Table 1–3 below lists system headers:

Table 1–3 System Headers

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

1.2.5.3 Utility Functions

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

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

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

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

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

Table 1–4 Objects Generated with SipFactory

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

SipFactory is located in the servlet context attribute under the default name, as shown in the following code:

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

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

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

Part II

Developing and Programming SIP Applications

This part describes programming guidelines and procedures for SIP applications.

Part II contains the following chapters:

2 Developing Converged Applications

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

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

2.1 Overview of Converged Applications

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

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

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

2.2 Assembling and Packaging a Converged Application

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

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

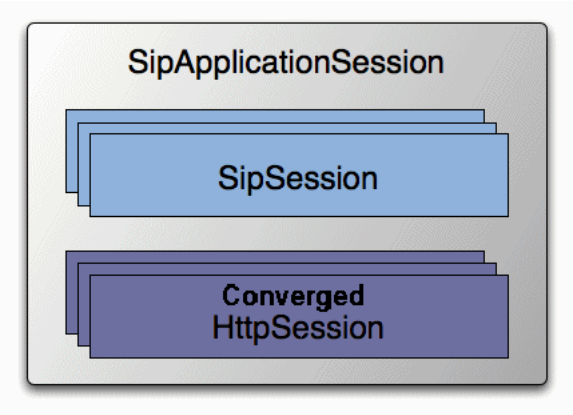
required. A `weblogic.xml` deployment descriptor may also be included to configure Servlet functionality in the WebLogic Server container.

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

2.3 Working with SIP and HTTP Sessions

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

Figure 2–1 Sessions in a Converged Application



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

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

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

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

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

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

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

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

2.3.1 Modifying the SipApplicationSession

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

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

To help application developers manage concurrent access to the application session object, WebLogic Server extends the standard `SipApplicationSession` object with `com.bea.wcp.sip.WlssSipApplicationSession`, and adds two interfaces, `com.bea.wcp.sip.WlssAction` and `com.bea.wcp.sip.WlssAsynchronous Action`, to encapsulate tasks performed to modify the session. When these APIs are

used, the SIP container ensures that all business logic contained within the `WlssAction` and WlssAsynchronousAction object is executed on a locked copy of the associated `SipApplicationSession` instance. The sections that follow describe each interface.

2.3.1.1 Synchronous Access

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

Example 2–1 Example Code using WlssAction API

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

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

2.3.1.2 Asynchronous Access

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

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

Example 2–2 Example Code using WlssAsynchronousAction API

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

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

2.4 Using the Converged Application Example

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

3 SIP Protocol Programming

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

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

3.1 Using Compact and Long Header Formats for SIP Messages

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

3.1.1 Overview of Header Format APIs and Configuration

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

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

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

3.1.2 Summary of Compact Headers

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

Table 3–1 Compact Header Abbreviations

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

Table 3–1 (Cont.) Compact Header Abbreviations

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

3.1.3 Assigning Header Formats with WlssSipServletMessage

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

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

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

3.1.4 Summary of API and Configuration Behavior

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

Table 3–2 API Behavior when Adding Headers

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

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

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

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

Table 3–3 API Behavior for System Headers

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

3.2 Using Content Indirection in SIP Servlets

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

3.2.1 Overview of Content Indirection

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

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

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

3.2.2 Using the Content Indirection API

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

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

3.2.3 Additional Information

Complete details about content indirection are available in RFC 4483.

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

3.3 Generating SNMP Traps from Application Code

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

3.3.1 Overview

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

- Info
- Notice

- Warning
- Error
- Critical
- Alert
- Emergency

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

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

3.3.2 Requirement for Accessing SipServletSnmpTrapRuntimeMBean

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

Example 3–1 Sample Role Requirement in sip.xml

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

3.3.3 Obtaining a Reference to SipServletSnmpTrapRuntimeMBean

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

Example 3–2 Sample Method for Accessing SipServletSnmpTrapRuntimeMBean

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

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

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

3.3.4 Generating an SNMP Trap

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

Example 3–3 Generating a SNMP Trap

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

  public MyServlet () {
  }

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

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

4

Requirements and Best Practices for SIP Applications

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

4.1 Overview of Developing Distributed Applications

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

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

4.2 Applications Must Not Create Threads

WebLogic Server is a multi-threaded application server that carefully manages resource allocation, concurrency, and thread synchronization for the modules it hosts.

To obtain the greatest advantage from the WebLogic Server architecture, construct your application modules according to the SIP Servlet and Java EE API specifications.

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

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

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

4.3 Servlets Must Be Non-Blocking

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

4.4 Store all Application Data in the Session

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

4.5 All Session Data Must Be Serializable

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 4–1 Session Access Resulting in a Deadlock

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

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

4.7 send() Calls Are Buffered

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

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

4.8 Mark SIP Servlets as Distributable

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

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

4.9 Use SipApplicationSessionActivationListener Sparingly

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

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

4.10 Session Expiration Best Practices

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

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

4.11 Observe Best Practices for Java EE Applications

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

5 Composing SIP Applications

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

- Section 5.1, "Application Composition Model"
- Section 5.2, "Using the Default Application Router"
- Section 5.3, "Configuring a Custom Application Router"
- Section 5.4, "Session Key-Based Request Targeting"

> **Note:** The SIP Servlet v1.1 specification (`http://jcp.org/en/jsr/detail?id=289`) describes a formal application selection and composition process, which is fully implemented in WebLogic Server. Use the SIP Servlet v1.1 techniques, as described in this document, for all new development. Application composition techniques described in earlier versions of WebLogic Server are now deprecated.
>
> WebLogic Server provides backwards compatibility for applications using version 1.0 composition techniques, provided that:
>
> - you *do not* configure a custom Application Router, and
> - you *do not* configure Default Application Router properties.

5.1 Application Composition Model

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

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

In contrast to the Application Router, which requires knowledge of which SIP applications are available for processing a message, individual SIP applications remain independent from one another. An individual application performs a very specific

service for a SIP request, without requiring any knowledge of other applications deployed on the system. (The Application Router does require knowledge of deployed applications, and the `SipApplicationRouter` interface provides for automatic notification of application deployment and undeployment.)

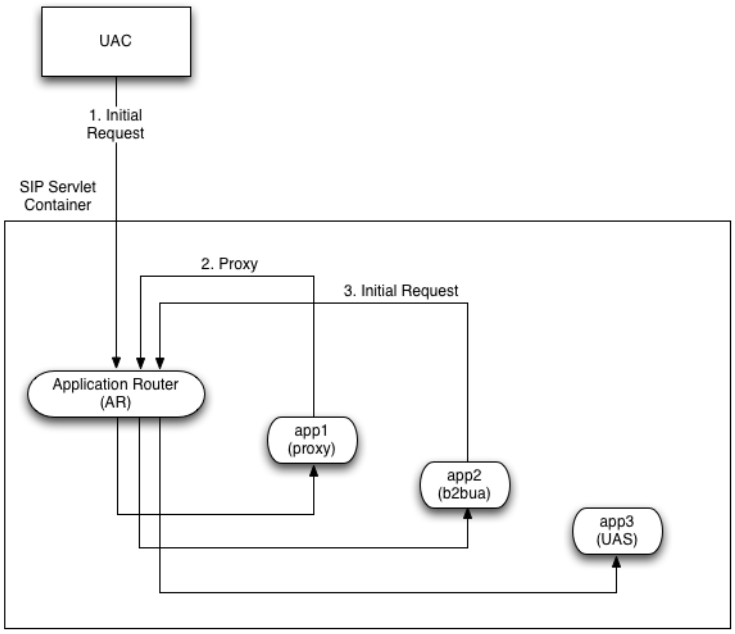
Individual SIP applications may complete their processing of an initial request by proxying or relaying the request, or by terminating the request as a User Agent Server (UAS). If an initial request is proxied or relayed, the SIP container again forwards the request to the Application Router, which selects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

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

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

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

Figure 5–1 Composed Application Model



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

5.2 Using the Default Application Router

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

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

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

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

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

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

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

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

5.3 Configuring a Custom Application Router

By default WebLogic Server uses its DAR implementation.

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

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

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

5.4 Session Key-Based Request Targeting

The SIP Servlet v1.1 specification also provides a mechanism for associating an initial request with an existing SipApplicationSession object. This mechanism is called session key-based targeting. Session key-based targeting is used to direct initial requests having a particular subscriber (request URI) or region, or other feature to an already-existing SipApplicationSession, rather than generating a new session. To use this targeting mechanism with an application, you create a method that generates

a unique key and annotate that method with `@SipApplicationKey`. When the SIP container selects that application (for example, as a result of the AR choosing it for an initial request), it obtains a key using the annotated method, and uses the key and application name to determine if the `SipApplicationSession` exists. If one exists, the container associates the new request with the existing session, rather than generating a new session.

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

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

6

Securing SIP Servlet Resources

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

6.1 Overview of SIP Servlet Security

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

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

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

Chapter 14 in the SIP Servlet API specification provides more details about the types of security available to SIP Servlets. SIP Servlet security features are similar to security

features available with HTTP Servlets; you can find additional information about HTTP Servlet security by referring to these sections in the Oracle WebLogic Server SIP Container documentation:

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

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

6.2 Triggering SIP Response Codes

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

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

6.3 Specifying the Security Realm

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

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

6.4 Role Mapping Features

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

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

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

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

The sections that follow describe WebLogic Server role assignment in more detail.

6.5 Using Implicit Role Assignment

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

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

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

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

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

6.6 Assigning Roles Using security-role-assignment

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

6.6.1 Important Requirements

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

> **Note:** If you specify a security-role-assignment in weblogic.xml but there is no corresponding security-role element in web.xml, WebLogic Server generates the error message:
>
> `The security-role-assignment references an invalid security-role: rolename`
>
> The server then implicitly maps the security-role defined in sip.xml to a role of the same name, as described in "Using Implicit Role Assignment".

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

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

Example 6–1 Declarative Security Constraints in sip.xml

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

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

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

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

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

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

```
  </security-role>
</web-app>
```

6.6.2 Assigning Roles at Deployment Time

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

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

6.6.3 Dynamically Assigning Roles Using the Administrative Console

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

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

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

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

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

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

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

6.7 Assigning run-as Roles

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

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

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

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

6.8 Role Assignment Precedence for SIP Servlet Roles

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

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

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

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

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

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

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

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

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

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

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

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

6.9 Debugging Security Features

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

6.10 weblogic.xml Deployment Descriptor Reference

The `weblogic.xml` DTD contains detailed information about each of the role mapping elements discussed in this section.

7

Enabling Message Logging

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

7.1 Overview

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

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

7.2 Enabling Message Logging

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

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

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

7.2.1 Specifying a Predefined Logging Level

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

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

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

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

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

7.2.2 Customizing Log Records

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

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

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

Table 7–1 Nested format Elements

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

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

Table 7–2 Available Tokens for Message Log Records

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

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

7.3 Specifying Content Types for Unencrypted Logging

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

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

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

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

Example 7–2 Logging String Content for Additional Content Types

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

7.4 Example Message Log Configuration and Output

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

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

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

Example 7–4 Sample Message Log Output

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

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

7.5 Configuring Log File Rotation

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

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

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

Table 7–3 XML Elements for Configuring Log Rotation

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

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

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

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

Example 7–5 Sample Log Rotation Configuration

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

Part III

Using Diameter

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

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

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

Part VI contains the following chapters:

8

Using the Diameter Base Protocol API

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

- Section 8.1, "Diameter Protocol Packages"
- Section 8.2, "Overview of the Diameter API"
- Section 8.3, "Working with Diameter Nodes"
- Section 8.4, "Implementing a Diameter Application"
- Section 8.5, "Working with Diameter Sessions"
- Section 8.6, "Working with Diameter Messages"
- Section 8.7, "Working with AVPs"
- Section 8.8, "Creating Converged Diameter and SIP Applications"

8.1 Diameter Protocol Packages

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

- Chapter 9, "Using the Profile Service API" describes how to access and manage subscriber profile data using the Diameter Sh application.
- Chapter 11, "Using the Diameter Rf Interface API for Offline Charging" describes how to issue offline charging requests using the Diameter Rf application.
- Chapter 12, "Using the Diameter Ro Interface API for Online Charging" describes how to perform online charging using the Diameter Ro application.

8.2 Overview of the Diameter API

All classes in the Diameter base protocol API reside in the root `com.bea.wcp.diameter` package. Table 8–1 describes the key classes, interfaces, and exceptions in this package.

Table 8–1 Key Elements of the Diameter Base Protocol API

Category	Element	Description
Diameter Node	Node	A class that represents a Diameter node implementation. A diameter node can represent a client- or server-based Diameter application, as well as a Diameter relay agent.
Diameter Applications	Application, ClientApplication	A class that represents a basic Diameter application. ClientApplication extends Application for client-specific features such as specifying destination hosts and realms. All Diameter applications must extend one of these classes to return an application identifier. The classes can also be used directly to create new Diameter sessions.
	ApplicationId	A class that represents the Diameter application ID. This ID is used by the Diameter protocol for routing messages to the appropriate application. The `ApplicationId` corresponds to one of the Auth-Application-Id, Acct-Application-Id, or Vendor-Specific-Application-Id AVPs contained in a Diameter message.
	Session	A class that represents a Diameter session. Applications that perform session-based handling must extend this class to provide application-specific behavior for managing requests and answering messages.
Message Processing	Message, Request, Answer	The Message class is a base class used to represent request and answer message types. Request and Answer extend the base class.
	Command	A class that represents a Diameter command code.
	RAR, RAA	These classes extend the `Request` and `Answer` classes to represent re-authorization messages.
	ResultCode	A class that represents a Diameter result code, and provides constant values for the base Diameter protocol result codes.
AVP Handling	Attribute	A class that provides Diameter attribute information.
	Avp, AvpList	Classes that represent one or more attribute-value pairs in a message. AvpList is also used to represent AVPs contained in a grouped AVP.
	Type	A class that defines the supported AVP datatypes.
Error Handling	DiameterException	The base exception class for Diameter exceptions.
	MessageException	An exception that is raised when an invalid Diameter message is discovered.
	AvpException	An exception that is raised when an invalid AVP is discovered.
Supporting Interfaces	Enumerated	An enum value that implements this interface can be used as the value of an AVP of type INTEGER32, INTEGER64, or ENUMERATED.
	SessionListener	An interface that applications can implement to subscribe to messages delivered to a Diameter session.
	MessageFactory	An interface that allows applications to override the default message decoder for received messages, and create new types of `Request` and `Answer` objects. The default decoding process begins by decoding the message header from the message bytes using an instance of `MessageFactory`. This is done so that an early error message can be generated if the message header is invalid. The actual message AVPs are decoded in a separate step by calling `decodeAvps`. AVP values are fully decoded and validated by calling `validate`, which in turn calls `validateAvp` for each partially-decoded AVP in the message.

In addition to these base Diameter classes, accounting-related classes are stored in the `com.bea.wcp.diameter.accounting` package, and credit-control-related classes

are stored in com.bea.wcp.diameter.cc. See Chapter 11, "Using the Diameter Rf Interface API for Offline Charging", and Chapter 12, "Using the Diameter Ro Interface API for Online Charging" for more information about classes in these packages.

8.2.1 File Required for Compiling Application Using the Diameter API

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

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

8.3 Working with Diameter Nodes

A diameter node is represented by the com.bea.wcp.diameter.Node class. A Diameter node may host one or more Diameter applications, as configured in the diameter.xml file. In order to access a Diameter application, a deployed application (such as a SIP Servlet) must obtain the diameter Node instance and request the application. Example 8–1 shows the sample code used to access the Rf application.

Example 8–1 Accessing a Diameter Node and Application

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

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

1. Set the environment for your domain:

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

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

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

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

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

8.4 Implementing a Diameter Application

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

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

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

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

Example 8–2 shows a simple Diameter client application that does not use sessions.

Example 8–2 Simple Diameter Application

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

8.5 Working with Diameter Sessions

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

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

Example 8–3 shows a simple Diameter session implementation.

Example 8–3 Simple Diameter Session

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

To use the sample session class, the `TestApplication` in Example 8–2 must add a factory method:

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

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

```
TestSession session = testApp.createSession();
```

```
Request req = session.creatRequest();
req.sent();
```

The answer is delivered directly to the Session object.

8.6 Working with Diameter Messages

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

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

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

8.6.1 Sending Request Messages

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

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

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

8.6.2 Sending Answer Messages

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

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

8.6.3 Creating New Command Codes

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

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

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

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

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

8.7 Working with AVPs

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

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

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

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

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

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

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

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

8.7.1 Creating New Attributes

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

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

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

Table 8–1 lists the available attribute types and describes how they are mapped to Java types.

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

Table 8–2 Attribute Types

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

8.8 Creating Converged Diameter and SIP Applications

The Diameter API enables you to create converged applications that utilize both SIP and Diameter functionality. A SIP Servlet can access an available Diameter application through the Diameter Node, as shown in Example 8–4.

Example 8–4 Accessing the Rf Application from a SIP Servlet

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

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

Converged applications can use the Diameter `SessionListener` interface to receive notification when a Diameter message is received by the session. The `SessionListener` interface defines a single method, `rcvMessage()`. Example 8–5 shows an example of how to implement the method.

Example 8–5 Implementing SessionListener

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

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

9

Using the Profile Service API

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

9.1 Overview of Profile Service API and Sh Interface Support

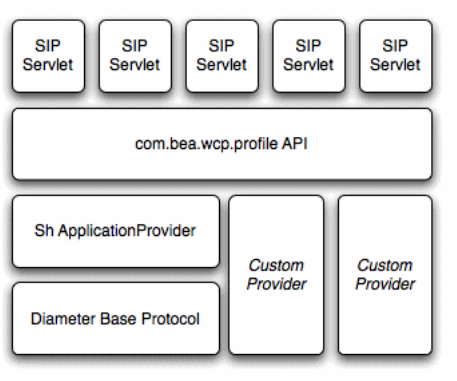
The IMS specification defines the Sh profile service as the method of communication between the Application Server (AS) function and the Home Subscriber Server (HSS), or between multiple IMS Application Servers. The AS uses the Sh profile service in two basic ways:

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

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

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

Figure 9–1 Profile Service API and Sh Provider Implementation



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

9.2 Enabling the Sh Interface Provider

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

9.3 Overview of the Profile Service API

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

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

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

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

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

Table 9–1 summarizes the possible URI values that can be supplied depending on the Sh data reference you are requesting. 3GPP TS 29.328 describes the possible data references and associated reference types in more detail.

Table 9–1 Possible URI Values for Sh Data References

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

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

Table 9–1 summarizes the required document selector key elements for each type of Sh data reference request.

Table 9–2 Summary of Document Selector Elements for Sh Data Reference Requests

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

Table 9–2 (Cont.) Summary of Document Selector Elements for Sh Data Reference Requests

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

9.5 Using a Constructed Document Key to Manage Profile Data

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

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

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

Example 9–1 shows a sample SIP Servlet that obtains and modifies profile data.

Example 9–1 Sample Servlet Using ProfileService to Retrieve and Write User Profile Data

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

9.6 Monitoring Profile Data with ProfileListener

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

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

9.6.1 Prerequisites for Listener Implementations

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

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

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

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

9.6.2 Implementing ProfileListener

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

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

Example 9–2 shows sample code for a Servlet that implements the `ProfileListener` interface.

Example 9–2 Sample Servlet Implementing ProfileListener Interface

```
package demo;
    import com.bea.wcp.profile.*;
    import javax.servlet.sip.SipServletRequest;
```

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

10

Developing Custom Profile Service Providers

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

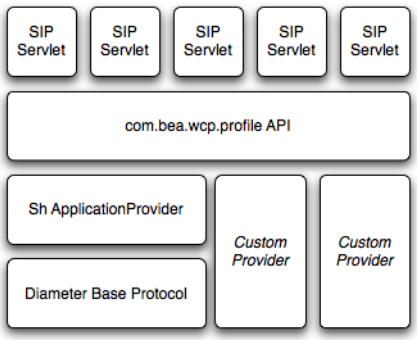
- Section 10.1, "Overview of the Profile Service API"
- Section 10.2, "Implementing Profile Service API Methods"
- Section 10.3, "Configuring and Packaging Profile Providers"
- Section 10.4, "Configuring Profile Providers Using the Administration Console"

10.1 Overview of the Profile Service API

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

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

Figure 10–1 Profile Service API and Provider Implementation



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

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

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

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

10.2 Implementing Profile Service API Methods

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

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

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

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

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

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

10.3 Configuring and Packaging Profile Providers

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

For most profile providers, you can simply package the implementation classes in a JAR file. Then register the library with WebLogic Server.

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

Example 10–1 shows a sample configuration for a provider that accesses data using XCAP.

Example 10–1 Provider Mapping in profile.xml

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

10.3.1 Mapping Profile Requests to Profile Providers

When an application makes a request using the Profile Service API, WebLogic Server must find a corresponding provider to process the request. By default, WebLogic Server maps the prefix of the requested URL to a provider `name` element defined in `profile.xml`. For example, with the basic configuration shown in Example 10–1, WebLogic Server would map Profile Service API requests beginning with `xcap://` to the provider class com.mycompany.profile.XcapProfileProvider.

Alternately, you can define a `mapping` entry in `profile.xml` that lists the prefixes corresponding to each named provider. Example 10–2 shows a mapping with two alternate prefixes.

Example 10–2 Mapping a Provider to Multiple Prefixes

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

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

If the explicit mapping capabilities of `profile.xml` are insufficient, you can create a custom mapping class that implements the com.bea.wcp.profile.ProfileRouter interface, and then identify that class in the `map-by-router` element. Example 10–3 shows an example configuration.

Example 10–3 Using a Custom Mapping Class

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

10.4 Configuring Profile Providers Using the Administration Console

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

Example 10–4 Enabling the Profile Service Resource in config.xml

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

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

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

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

11

Using the Diameter Rf Interface API for Offline Charging

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

11.1 Overview of Rf Interface Support

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

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

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

11.2 Understanding Offline Charging Events

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

The reporting of offline charging events to the CDF is managed through the Diameter Accounting Request (ACR) message. Rf supports the ACR event types described in Table 11–1.

Table 11–1 Rf ACR Event Types

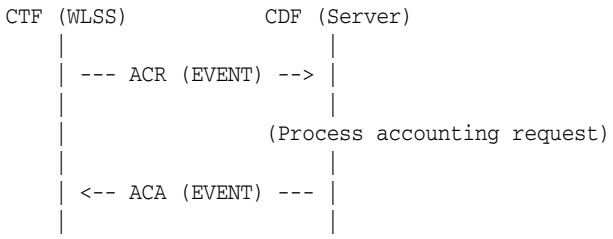
Request	Description
START	Starts an accounting session.
INTERIM	Updates an accounting session.
STOP	Stops an accounting session
EVENT	Indicates a one-time accounting event.

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

11.2.1 Event-Based Charging

Event-based charging events are reported through the ACR EVENT message. Example 11–1 shows the basic message flow.

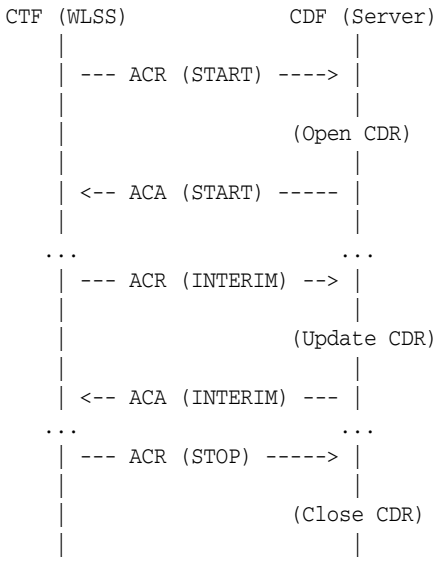
Example 11–1 Message Flow for Event-Based Charging



11.2.2 Session-Based Charging

Session-based charging uses the ACR START, INTERIM, and STOP requests to report usage to the CDF. During a session, the CTF may report multiple ACR INTERIM requests depending on the session lifecycle. Example 11–2 shows the basic message flow

Example 11–2 Message Flow for Session-Based Charging



```
                    | <-- ACA (STOP) ------ |
                    |                       |
```

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

11.3 Configuring the Rf Application

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

Example 11–3 shows a sample excerpt from `diameter.xml` that enables Rf with a CDF realm of "oracle.com" and host "cdf.oracle.com:"

Example 11–3 Sample Rf Application Configuration (diameter.xml)

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

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

11.4 Using the Offline Charging API

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

The classes in package `com.bea.wcp.diameter.accounting` provide general support for Diameter accounting messages and sessions. Table 11–2 summarizes the classes.

Table 11–2 Diameter Accounting Classes

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

In addition, classes in package `com.bea.wcp.diameter.charging` support the Rf application specifically. Table 11–3 summarizes the classes.

Table 11–3 Diameter Rf Application Support Classes

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

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

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

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

11.4.1 Accessing the Rf Application

If the Rf application is deployed, then applications deployed on WebLogic Server can obtain an instance of the application from the Diameter node (`com.bea.wcp.diameter.Node` class). Example 11–4 shows the sample Servlet code used to obtain the Diameter `Node` and access the Rf application.

Example 11–4 Accessing the Rf Application

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

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

11.4.2 Implementing Session-Based Charging

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

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

Example 11–5 shows the sample code used to start a new session-based accounting session.

Example 11–5 Starting a Session-Based Account Session

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

In Example 11–5, the `RfSession` is stored as a SIP application session attribute so that it can be used to send additional accounting requests as the call progresses. Example 11–6 shows how to send an INTERIM request.

Example 11–6 Sending an INTERIM request

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

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

11.4.2.1 Specifying the Session Expiration

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

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

11.4.2.2 Sending Asynchronous Events

Applications generally use the synchronous `sendAndWait()` method. However, if latency is critical, an asynchronous API is provided wherein the application Servlet is asynchronously notified when an answer message is received from the CDF. To use the asynchronous API, an application first registers an instance of `SessionListener` in order to asynchronously receive messages delivered to the session, as shown in Example 11–7.

Example 11–7 Registering a SessionListener

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

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

When a Diameter request or answer message is received from the CDF, the application Servlet is notified by calling the `rcvMessage(Message msg)` method. The associated SIP application session can then be retrieved from the `RfSession` if it was stored as a session attribute, as shown in Example 11–8.

Example 11–8 Retrieving the RfSession after a Notification

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

11.4.3 Implementing Event-Based Charging

For an event-based charging request, the charging request is a one-time event and the session is automatically terminated upon receipt of the corresponding EVENT ACA message. The `sendAndWait(long timeout)` method can be used to synchronously send the EVENT request and block the thread until a response has been received from the CDF. Example 11–9 shows an example that uses an `RfSession` for sending an event-based charging request.

Example 11–9 Event-Based Charging Using RfSession

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

For convenience, it is also possible send event-based charging requests using the `RfApplication` directly, as shown in Example 11–10.

Example 11–10 Event-Based Charging Using RfApplication

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

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

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

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

11.4.4 Using the Accounting Session State

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

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

12

Using the Diameter Ro Interface API for Online Charging

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

12.1 Overview of Ro Interface Support

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

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

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

12.2 Understanding Credit Authorization Models

RFC 4006 defines two basic types of credit authorization models:

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

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

The sections that follow describe each model in more detail.

12.2.1 Credit Authorization with Unit Determination

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

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

12.2.2 Credit Authorization with Direct Debiting

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

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

12.2.3 Determining Units and Rating

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

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

12.3 Configuring the Ro Application

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

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

Example 12–1 shows a sample excerpt from `diameter.xml` that enables Ro with an OCF host name of "myocs.oracle.com."

Example 12–1 Sample Ro Application Configuration (diameter.xml)

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

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

12.4 Overview of the Online Charging API

WebLogic Server provides an online charging API to enable any deployed application to act as a CTF and issue online charging events to an OCS through the Ro protocol. All online charging requests use the Diameter Credit-Control-Request (CCR) message. The CC-Request-Type AVP is used to indicate the type of charging used. In the charging API, the CC-Request-Type is represented by the `RequestType` class in package `com.bea.wcp.diameter.cc`. Table 12–1 shows the request types associated with different credit authorization models.

Table 12–1 Credit Control Request Types

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

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

The base diameter package, `com.bea.wcp.diameter`, contains classes to support the re-authorization requests used in Ro. The `com.bea.wcp.diameter.cc` package contains classes to support credit-control applications, including Ro applications. `com.bea.wcp.diameter.charging` directly supports the Ro credit-control application. Table 12–2 summarizes the classes of interest to Ro credit-control.

Table 12–2 Summary of Ro Classes

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

12.5 Accessing the Ro Application

If the Ro application is deployed, then applications deployed on WebLogic Server can obtain an instance of the application from the Diameter node (`com.bea.wcp.diameter.Node` class). Example 12–2 shows the sample Servlet code used to obtain the Diameter `Node` and access the Ro application.

Example 12–2 Accessing the Ro Application

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

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

12.6 Implementing Session-Based Charging

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

The example in Example 12–3 creates a new `RoSession` for event-based charging, and sends a CCR request to start the first interrogation. The `RoSession` instance is saved so that it can be terminated later, after the service has finished.

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

Example 12–3 Creating and Using a RoSession

```
RoSession session = roApp.createSession();
CCR ccr = session.createCCR(RequestType.INITIAL);
```

```
CCA cca = ccr.sendAndWait();
sipAppSession.setAttribute("RoSession", session);
...
```

12.6.1 Handling Re-Auth-Request Messages

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

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

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

Example 12–4 shows sample `rcvMessage()` code for processing a Re-Auth-Request.

Example 12–4 Managing a Re-Auth-Request

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

In Example 12–4, upon receiving the Re-Auth-Request the application sends an RAA with the result code DIAMETER_LIMITED_SUCCESS to indicate to the OCS that an additional CCR request is required in order to complete the procedure. The CCR is then sent to initiate credit re-authorization.

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

12.7 Sending Credit-Control-Request Messages

The CCR class represents a Diameter Credit-Control-Request message, and can be used to send credit control requests to the OCF. For both ECUR (Event-Based Charging

with Unit Reservation) and SCUR (Session-Based Charging with Unit Reservation), an instance of `RoSession` is used to create new CCR requests. You can also use `RoApplication` directly to create CCR messages for IEC (Immediate Event Charging). Example 12–5 shows an example of how to create and send a CCR.

Example 12–5 Creating and Sending a CCR

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

Once a CCR request is created, you can set whatever application- or service-specific AVPs that are required before sending the request using the `addAvp()` method. Because some of the same AVPs need to be included in each new request for the session, it is also possible to set these AVPs on the session itself. Example 12–6 shows a sample that sets:

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

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

Example 12–6 Setting AVPs in the CCR

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

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

12.8 Handling Failures

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

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

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

Example 12–7 terminates the credit control session if timer Tx expires before receiving the CCA. If the CCA is received later by the Diameter subsystem, the message is ignored because the session longer exists.

Example 12–7 Checking for Timer Tx Expiry

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

Part IV

Reference

This part contains the following appendices:

- Appendix A, "Profile Service Provider Configuration Reference (profile.xml)"
- Appendix B, "Developing SIP Servlets Using Eclipse"
- Appendix C, "Porting Existing Applications to Oracle WebLogic Server SIP Container"

A

Profile Service Provider Configuration Reference (profile.xml)

This appendix provides a complete reference to the profile provider configuration file, `profile.xml`, in the following sections:

- Section A.1, "Overview of profile.xml"
- Section A.2, "Graphical Representation"
- Section A.3, "Editing profile.xml"
- Section A.4, "XML Schema"
- Section A.5, "Example profile.xml File"
- Section A.6, "XML Element Description"

A.1 Overview of profile.xml

The `profile.xml` file configures attributes of a profile service provider, such as:

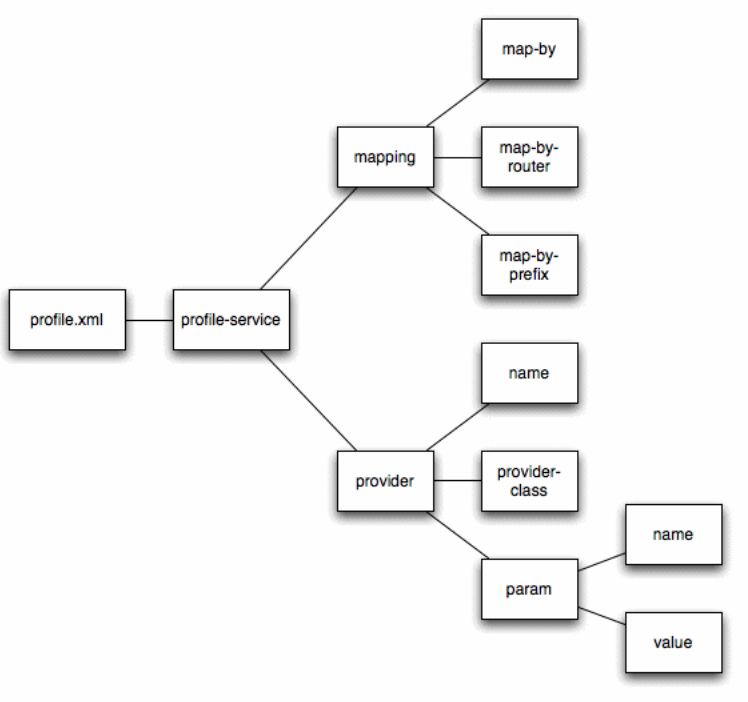
- Name of the provider
- Class name of the provider implementation
- Optional arguments passed to the provider
- Mapping rules for using the provider.

`profile.xml` is stored in the *`DOMAIN_DIR`*`/config/custom` subdirectory where *`DOMAIN_DIR`* is the root directory of the WebLogic Server domain.

A.2 Graphical Representation

Figure A–1 shows the element hierarchy of the `profile.xml` file.

Figure A–1 Element Hierarchy of profile.xml



A.3 Editing profile.xml

Oracle recommends using the Administration Console profile service extension to modify `profile.xml` indirectly, rather than editing the file by hand. Using the Administration Console ensures that the `profile.xml` document always contains valid XML. See Configuring Profile Providers Using the Administration Console in Developing Applications with WebLogic Server.

You may need to manually view or edit `profile.xml` to troubleshoot problem configurations, repair corrupted files, or to roll out custom profile provider configurations to a large number of machines when installing or upgrading WebLogic Server. When you manually edit `profile.xml`, you must reboot servers to apply your changes.

A.3.1 Steps for Editing profile.xml

If you need to modify `profile.xml` on a production system, follow these steps:

1. Use a text editor to open the *`DOMAIN_DIR`*`/config/custom/profile.xml` *`file, where DOMAIN_DIR`* is the root directory of the WebLogic Server domain.
2. Modify the `profile.xml` file as necessary. See "XML Element Description" for a full description of the XML elements.
3. Save your changes and exit the text editor.
4. Reboot or start servers to have your changes take effect.
5. Test the updated system to validate the configuration.

A.4 XML Schema

The full schema for the `profile.xml` file is bundled within the `profile-service-descriptor-binding.jar` library, installed in the *`WLSS_HOME`*`/server/lib/wlss` directory.

A.5 Example profile.xml File

See Developing Custom Profile Providers in *Developing SIP Applications* for sample listings of `profile.xml` configuration files.

A.6 XML Element Description

The following sections describe each XML element in `profile.xml`.

A.6.1 profile-service

The top level `profile-service` element contains the entire profile service configuration.

A.6.2 mapping

Specifies how requests for profile data are mapped to profile provider implementations.

A.6.2.1 map-by

Specifies the technique used for mapping documents to providers:

- `router` uses a custom router class, specified by `map-by-router`, to determine the provider.
- `prefix` uses the specified `map-by-prefix` entry to map documents to a provider.
- `provider-name` uses the specified `name` element in the `provider` entry to map documents to a provider.

A.6.2.2 map-by-prefix

Specifies the prefix used to map documents to profile providers when mapping by prefix.

A.6.2.3 map-by-router

Specifies the router class (implementing `com.bea.wcp.profile.ProfileRouter`) used to map documents to profile providers with router-based mapping.

A.6.3 provider

Configures the profile provider implementation and startup options.

A.6.3.1 name

Specifies a name for the provider configuration. The `name` element is also used for mapping documents to the provider if you specify the `provider-name` mapping technique.

A.6.3.2 provider-class

Specifies the profile provider class (implementing `com.bea.wcp.profile.ProfileServiceSpi`).

A.6.3.3 param

Uses the `name` and `value` elements to specify optional parameters to the provider implementation.

B

Developing SIP Servlets Using Eclipse

This appendix describes how to use Eclipse to develop SIP Servlets for use with WebLogic Server, in the following sections:

- Section B.1, "Overview"
- Section B.2, "Setting Up the Development Environment"
- Section B.3, "Building and Deploying the Project"
- Section B.4, "Debugging SIP Servlets"

B.1 Overview

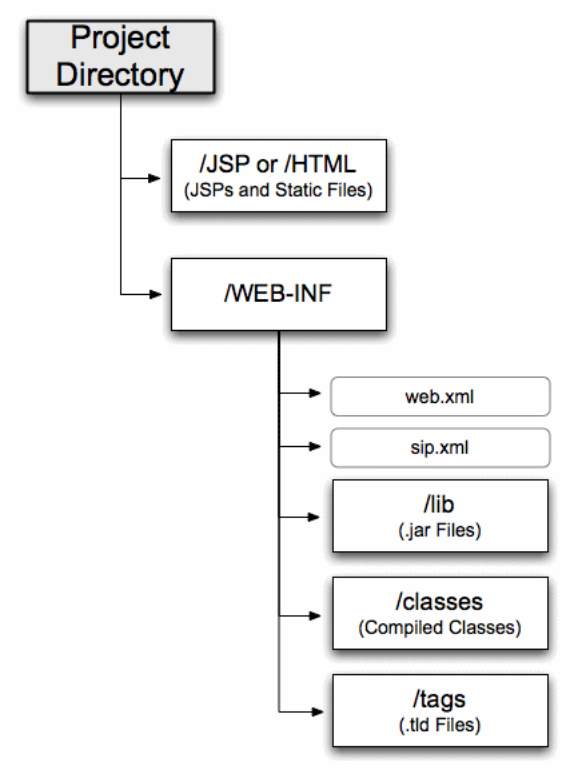
This document provides detailed instructions for using the Eclipse IDE as a tool for developing and deploying SIP Servlets with WebLogic Server. The full development environment requires the following components, which you must obtain and install before proceeding:

- WebLogic Server
- JDK 1.6.05
- Eclipse version 3.4 or Eclipse 3.3 Europe. This includes a CVS client and server (required only for version control).

B.1.1 SIP Servlet Organization

Building a SIP Servlet produces a Web Archive (WAR file or directory) as an end product. A basic SIP Servlet WAR file contains the subdirectories and contents described in Figure B–1.

Figure B–1 SIP Servlet WAR Contents



B.2 Setting Up the Development Environment

Follow these steps to set up the development environment for a new SIP Servlet project:

1. Create a new WebLogic Server Domain.
2. Create a new Eclipse project.

The sections that follow describe each step in detail.

B.2.1 Creating a WebLogic Server Domain

In order to deploy and test your SIP Servlet, you need access to a WebLogic Server domain that you can reconfigure and restart as necessary. Follow the instructions in *Oracle WebLogic Server SIP Container Installation Guide* to create a new domain using the Configuration Wizard. When generating a new domain:

- Select Development Mode as the startup mode for the new domain.
- Select Sun SDK 1.6.05 as the SDK for the new domain.

B.2.2 Verifying the Default Eclipse JVM

Eclipse 3.4 uses the required version Java 6 (1.6) by default. Follow these steps to verify the configured JVM:

1. Start Eclipse.
2. Select **Window** >**Preferences**.
3. Expand the Java category in the left pane, and select Installed JREs.

4. Verify that Java 6 (1.6) is configured. If it is, proceed to step 10.
5. If not configured correctly, click Add... to add a new JRE.
6. Enter a name to use for the new JRE in the JRE name field.
7. Click the Browse... button next to the JRE home directory field. Then navigate to the `MIDDLEWARE_HOME/jdk160_05` directory and click **OK**.
8. Click **OK** to add the new JRE.
9. Select the check box next to the new JRE to make it the default.
10. Click **OK** to dismiss the preferences dialog.

B.2.3 Creating a New Eclipse Project

Follow these steps to create a new Eclipse project for your SIP Servlet development, adding the WebLogic Server libraries required for building and deploying the application:

1. Start Eclipse.
2. Select **File** > **New** > **Project**...
3. Select Java Project and click **Next**.
4. Enter a name for your project in the Project Name field.
5. In the Location field, select "Create project in workspace" if you have not yet begun writing the SIP Servlet code. If you already have source code available in another location, select "Create project at external location" and specify the directory. Click **Next**.
6. Click the Libraries tab and follow these steps to add required JARs to your project:
   a. Click **Add External JARs**...
   b. Use the JAR selection dialog to add the `MIDDLEWARE_HOME/server/lib/weblogic.jar` file to your project.
   c. Repeat the process to add the `MIDDLEWARE_HOME/server/lib/wlss/sipservlet.jar` and `MIDDLEWARE_HOME/server/lib/wlss/wlssapi.jar` files to your project.
7. Add any additional JAR files that you may require for your project.
8. To enable deploying directly from eclipse, change the build folder from /src/build to /src/WebContent/WEB-INF/classes. This means that you do not have to package the application before deploying it.
9. Click **Finish** to create the new project. Eclipse displays your new project name in the Package Explorer.
10. Right-click on the name of your project and use the **New** >**Folder** command to recreate the directory structure shown in Figure B–1, "SIP Servlet WAR Contents".

B.3 Building and Deploying the Project

The `build.xml` file that you created compiles your code, packages the WAR, and copies the WAR file to the `/applications` subdirectory of your development domain. WebLogic Server automatically deploys valid applications located in the `/applications` subdirectory.

B.4 Debugging SIP Servlets

In order to debug SIP Servlets, you must enable certain debug options when you start WebLogic Server. Follow these steps to add the required debug options to the script used to start WebLogic Server, and to attachthe debugger from within Eclipse:

> **Note:** On Linux, debug is enabled by default if you install in developer mode. However, the port is set to 8453.

1. Use a text editor to open the `StartWebLogic.cmd` script for your development domain.
2. Beneath the line that reads:

```
set JAVA_OPTIONS=

Enter the following line:

set DEBUG_OPTS=-Xdebug -Xrunjdwp:transport=dt_
socket,address=9000,server=y,suspend=n
```

3. In the last line of the file, add the `%DEBUG_OPTS%` variable in the place indicated below:

```
"%JAVA_HOME%\bin\java" %JAVA_VM% %MEM_ARGS% %JAVA_OPTIONS% %DEBUG_OPTS%
-Dweblogic.Name=%SERVER_NAME% -Dweblogic.management.username=%WLS_USER%
-Dweblogic.management.password=%WLS_PW%
-Dweblogic.management.server=%ADMIN_URL%
-Djava.security.policy="%MIDDLEWARE_HOME%\server\lib\weblogic.policy"
weblogic.Server
```

4. Save the file and use the script to restart WebLogic Server.
5. To attach the debugger from within Eclipse select the **Run** > **Open debug** dialog.
6. Create a new "Remote Java Application".
7. Enter the host and port corresponding to the DEBUG_OPTS.

C

Porting Existing Applications to Oracle WebLogic Server SIP Container

This chapter describes guidelines and issues related to porting existing applications based on SIP Servlet v1.0 specification to Oracle WebLogic Server SIP Container and the SIP Servlet v1.1 specification. It contains the following sections:

C.1 Application Router and Legacy Application Composition

The SIP Servlet v1.1 specification describes a formal application selection and composition process, which is fully implemented in WebLogic Server. Use the SIP Servlet v1.1 techniques for all new development. Application composition techniques described in earlier versions are now deprecated.

WebLogic Server provides backwards compatibility for applications using version SIP Servlet 1.0 composition techniques, provided that:

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

C.2 SipSession and SipApplicationSession Not Serializable

The `SipSession` and `SipApplicationSession` interfaces are no longer serializable in the SIP Servlet v1.1 specification. WebLogic Server maintains binary compatibility for the earlier v1.0 specification, to allow any compiled applications that treat these interfaces as serializable objects to work. However, you must modify the

source code of such applications before you can recompile them with WebLogic Server.

Version 1.0 Servlets that stored the `SipSession` as a serializable info object using the `TimerService.createTimer` API can achieve similar functionality by storing the `SipSession` ID as the serializable `info` object. On receiving the timer expiration callback, applications must use the `SipApplicationSession` and the serialized ID object returned by the `ServletTimer` to find the `SipSession` within the `SipApplicationSession` using the retrieved ID. See the SIP Servlet v1.1 API JavaDoc for more information.

C.3 SipServletResponse.setCharacterEncoding() API Change

`SipServletResponse.setCharacterEncoding()` no longer throws `UnsupportedEncodingException`. If you have an application that explicitly catches `UnsupportedEncodingException` with this method, the existing, compiled application can be deployed to WebLogic Server unchanged. However, the source code must be modified to not catch the exception before you can recompile.

C.4 Transactional Restrictions for SipServletRequest and SipServletResponse

SIP Servlet v1.1 acknowledges that `SipServletRequest` and `SipServletResponse` objects always belong to a SIP transaction. The specification further defines the conditions for committing a message, after which no application can modify or re-send the message. See *5.2 Implicit Transaction State* in the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`) for a list of conditions that commit SIP messages.

As a result of this change, any attempt to modify (set, add, or remove a header) or send a committed message now results in an `IllegalStateException`. Ensure that any existing code checks for the committed status of a message using `SipServletMessage.isCommitted()` before modifying or sending a message.

C.5 Immutable Parameters for New Parameterable Interface

SIP Servlet v1.1 introduces a new `javax.servlet.sip.Parameterable` interface for accessing, creating, and modifying parameters in various SIP headers. Note that the system header parameters described in Table C–1 are immutable and cannot be modified using this new interface.

Table C–1 Immutable System Header Parameters

Header	Immutable Parameters
From	tag
To	tag
Via	branch, received, rport, wlsslport, wlssladdr, maddr, ttl
Record-Route	All parameters are immutable.
Route	For initial requests, the application that pushes the Route header can modify any of the header's parameters. In all other cases, the parameters of the Route header are immutable.

Table C–1 (Cont.) Immutable System Header Parameters

Path	For Register requests, the application that pushes the Path header can modify any of the header's parameters.In all other cases, the parameters of the Path header are immutable.

C.6 Stateless Transaction Proxies Deprecated

For applications in WebLogic Server, the Proxy function is always transactionally stateful, and setting the Proxy object to stateless has no effect.

The `Proxy.setStateful()` and `Proxy.getStateful()` methods are redundant: `Proxy.getStateful()` always returns true, and `Proxy.setStateful()` performs no operation.

C.7 Backward-Compatibility Mode for v1.0 Deployments

WebLogic Server automatically detects precompiled, v1.0 deployments and alters the SIP container behavior to maintain backward compatibility. The sections that follow describe differences in behavior that occur when deploying v1.0 SIP Servlets to WebLogic Server.

C.7.1 Validation Warnings for v1.0 Servlet Deployments

The SIP Servlet v1.1 specification requires more strict validation of Servlet deployments than the previous specification. In the following cases, v1.0 SIP Servlets can be successfully deployed to WebLogic Server, but a warning message is displayed at deployment:

- If a listener is declared in the `listener-class` element of a v1.0 deployment descriptor but the corresponding class does not implement the `EventListener` interface, a warning is displayed during deployment. (Version 1.1 SIP Servlets that declare a listener *must* implement `EventListener`, or the application cannot be deployed).
- If a SIP Servlet is declared in the `servlet-class` element of a v1.0 deployment descriptor, but the corresponding class does extend the `SipServlet` abstract class, a warning is displayed. (Version 1.1 SIP Servlets *must* extend `SipServlet`, or the application cannot be deployed).

C.7.2 Modifying Committed Messages

The SIP Servlet v1.1 specification now recommends that the SIP container throw an `IllegalStateException` if an application attempts to modify a committed message. To maintain backward compatibility, WebLogic Server throws the `IllegalStateException` only when a version 1.1 SIP Servlet deployment modifies a committed message.

C.7.3 Path Header as System Header

The SIP Servlet v1.1 specification now defines the `Path` header as a system header, which cannot be modified by an application. Version 1.0 SIP Servlets can still modify the `Path` header, but a warning message is generated. Version 1.1 SIP Servlets that attempt to modify the `Path` header fail with an `IllegalArgumentException`.

C.7.4 SipServletResponse.createPrack() Exception

In WebLogic Server, `SipServletResponse.createPrack()` can throw `Rel100Exception` only for version 1.1 SIP Servlets. `createPrack()` does not throw the exception for version 1.0 SIP Servlets to maintain backward compatibility.

C.7.5 Proxy.proxyTo() Exceptions

For version 1.1 SIP Servlets, WebLogic Server throws an `IllegalStateException` if a version 1.1 SIP Servlet specifies a duplicate branch URI with `Proxy.proxyTo(uri)` or `Proxy.proxyTo(uris)`. To maintain backward compatibility, WebLogic Server ignores the duplicate URIs (and throws no exception) if a version 1.0 SIP Servlet specifies duplicate URIs with these methods.

C.7.6 Changes to Proxy Branch Timers

SIP Servlet v1.1 makes several protocol changes that effect the behavior of proxy branching for both sequential and parallel proxying.

For sequential proxying, the v1.1 specification requires that WebLogic Server start a branch timer using the maximum of the `sequential-search-timeout` value, which is configured in `sip.xml`, or SIP protocol Timer C (> 3 minutes). Prior versions of WebLogic Server always set sequential branch proxy timeouts using the value of `sequential-search-timeout`; this behavior is maintained for v1.0 deployments.

For parallel proxying, the v1.1 specification provides a new `proxyTimeout` value that controls proxying. The specification requires that WebLogic Server reset a branch timer using the configured `proxyTimeout` value, rather than using the Timer C value as required in the SIP Servlet v1.0 specification. The Timer C value is still used for v1.0 deployments.

C.8 Deprecated APIs

Earlier versions of WebLogic SIP Server provided proprietary APIs to support functionality and RFCs that were not supported in the SIP Servlet v1.0 specification. The SIP Servlet v1.1 specification adds new RFC support and functionality, making the proprietary APIs redundant. Table C–2 shows newly-available SIP Servlet v1.1 methods that must be used in place of now-deprecated WebLogic SIP Server methods. The deprecated methods are still available in this release to provide backward compatibility for v1.0 applications.

Table C–2 Deprecated APIs

Deprecated Methods (WebLogic SIP Server Proprietary)	Replacement Method (SIP Servlet v1.1)

`WlssSipServlet.doRefer(),`
`WlssSipServlet.doUpdate(),`
`WlssSipServlet.doPrack()`
	`SipServlet.doRefer(),`
	`SipServlet.doUpdate(),`
	`SipServlet.doPrack()`

`WlssSipServletResponse.createPrack()`	`SipServletResponse.createPrack()`
`WlssProxy.getAddToPath(),` `WlssProxy.setAddToPath()`	`Proxy.getAddToPath(),Proxy.setAddToPath()`
`WlssSipServletMessage.setHeaderForm(),` `WlssSipServletMessage.getHeaderForm()`	`SipServletMessage.setHeaderForm(),` `SipServletMessage.getHeaderForm()`
`com.bea.wcp.util.Sessions`	See Table 6–1, "Sessions in a Converged Application".

C.9 SNMP MIB Changes

Previous versions of the WebLogic Server SNMP MIB definition did not follow the WebLogic MIB naming convention. Specifically, the MIB table column name label did not begin with the table name. WebLogic Server changes the SNMP MIB definition to prepend labels with `sipServer` in order to comply with the WebLogic naming convention and provide compatibility with WebLogic tools that generate the metadata file.

For example, in version 3.x the `SipServerEntry` MIB definition was:

```
SipServerEntry  ::=  SEQUENCE {
sipServerIndex  DisplayString,
t1TimeoutInterval  INTEGER,
t2TimeoutInterval  INTEGER,
t4TimeoutInterval  INTEGER,
....
}
```

In WebLogic Server, the definition is now:

```
SipServerEntry  ::=  SEQUENCE {
sipServerIndex  DisplayString,
sipServerT1TimeoutInterval  Counter64,
sipServerT2TimeoutInterval  INTEGER,
sipServerT4TimeoutInterval  INTEGER,
.....
}
```

This change in the MIB may cause backwards compatibility issues if an application or script uses the MIB table column name labels directly. All hard-coded labels, such as `iso.org.dod.internet.private.enterprises.bea.wlss.sipServerTable.t1TimeoutInterval` must be changed to prepend the table name (`iso.org.dod.internet.private.enterprises.bea.wlss.sipServerTable.sipServerT1TimeoutInterval`).

Note: Client-side SNMP tools generally load a MIB and issue commands to retrieve values based on the loaded MIB labels. These tools are unaffected by the above change.

The complete WebLogic Server MIB file is installed as $WLSS_HOME/server/lib/wlss/BEA-WLSS-MIB.asn1.

C.10 Renamed Diagnostic Monitors and Actions

The diagnostic monitors and diagnostic actions provided in WebLogic Server are now prefixed with `occas/`. For example, the SIP Server 3.1 `Sip_Servlet_Before_Service` monitor is now named `occas/Sip_Servlet_Before_Service`. You must update any existing diagnostic configuration files or applications that reference the non-prefixed names before they can work with WebLogic Server.

Index

R

S

U