Oracle® Communications Converged Application Server

Developer's Guide

Release 7.1

F18460-01

May 2019

ORACLE®

Oracle Communications Converged Application Server Developer's Guide, Release 7.1

F18460-01

Contents

4 Best Practices for SIP Applications

5 Composing SIP Applications

10 Forking SIP Requests, Dialog Termination, and Session Keep Alive

11 Using Compact and Long Header Formats for SIP Messages

12 Developing Custom Profile Service Providers

13 Using Content Indirection in SIP Servlets

14 Securing SIP Servlet Resources

15 Enabling Message Logging

16 Generating SNMP Traps from Application Code

17 Using the Location Service RESTful Interface

Preface

This document provides an overview of Session Initiation Protocol (SIP) Servlets and developing SIP applications for Oracle Communications Converged Application Server.

Audience

This document is intended for developers who build SIP applications for use with Converged Application Server. Expertise with Java Enterprise Edition concepts as well as SIP is required.

Documentation Accessibility

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

Access to Oracle Support

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

1

About Developing Applications for the Converged Application Server

This chapter introduces application development for the Oracle Communications Converged Application Server.

About Converged Application Server APIs

Converged Application Server supports a set of different APIs:

- Session Initiation Protocol (SIP) Servlet API
- Media server API
- Diameter API

Communications-oriented applications can also use the Java Platform, Enterprise Edition (JEE) APIs exposed by Oracle WebLogic Server, and thus making them converged applications.

The SIP Servlet container exposes a Java Specification Requests (JSR) 359 compliant API for developing SIP applications.

The media server application program interface (API) is a JSR 309 compliant and provides an object model for controlling media server resources and the topology of media streams independently of the underlying media server control protocols. Media server specifics are handled by a JSR 309 Driver, similar to how Java Database Connectivity (JDBC) are abstracting away database specifics. This allows an application to interact with different media servers regardless of vendor.

The Diameter API provides programmatic access to Diameter nodes. This books does not cover this topic, see *Converged Application Server Diameter Application Development Guide* for further information.

2

Overview of SIP Servlet Application Development

This chapter describes the Session Initiation Protocol (SIP) protocol, and provides a background on SIP application development using the Java programming language.

About the SIP Protocol

SIP is a simple network signalling protocol for creating and terminating sessions with one or more participant. The SIP protocol is designed to be independent of the underlying transport protocol, so SIP applications can run on Transport Control Protocol (TCP), User Datagram Protocol (UDP), or other lower-layer networking protocols.

Typically, the SIP protocol is used for internet telephony and multimedia distribution between two or more endpoints. For example, one person can initiate a telephone call to another person using SIP, or someone may create a conference call with many participants.

The SIP protocol was designed to be very simple, with a limited set of commands. It is also text-based, so humans can read the SIP messages passed between endpoints in a SIP session.

SIP Requests

The SIP protocol defines the following common request types:

Table 2–1 SIP Request Types

SIP Request	Description
INVITE	Initiates a session between two participants.
ACK	The client acknowledges receiving the final message from an INVITE request.
BYE	Terminates a connection.
CANCEL	Cancels any pending actions, but does not terminate any accepted connections.
OPTIONS	Queries the server for a list of capabilities.
REGISTER	Registers the address in the To header with the server.
INFO	Sends a mid-session information that does not modify session state.
UPDATE	Modifies session state without changing the dialog state.

Table 2–1 (Cont.) SIP Request Types

SIP Request	Description
SUBSCRIBE	Subscribes to event notifications from a notifier.
NOTIFY	Notifies a subscriber of a new event.
REFER	As a recipient to issue a SIP call transfer request.
PRACK	A provisional acknowledgement.
PUBLISH	Publishes an event to the server.
MESSAGE	Simple instant message transport.

SIP requests are codes used to indicate the various stages in a connection between SIP-enabled entities.

SIP Responses

The SIP Protocol uses response codes similar to the HTTP protocol. Some of the common response codes are:

- 100 (Trying)
- 200 (OK)
- 404 (Not found)
- 500 (Server error)
- 600 (Global failure)

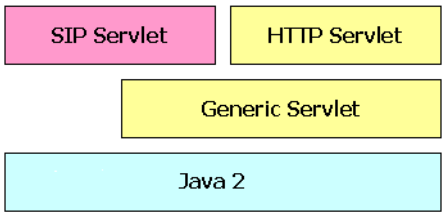
What are SIP Servlets?

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. A Servlet is a Java class in Java EE that conforms to the Java Servlet application program interface (API), a protocol by which a Java class may respond to requests. They are not tied to a specific client-server protocol, but are most often used with the HTTP protocol. Therefore, the word “Servlet” is often used in the meaning of “HTTP Servlet”.

A SIP servlet is a Java programming language server-side component that performs SIP signalling. SIP servlets are managed by a SIP servlet container, which typically are part of a SIP-enabled application server. SIP servlets interact with clients by responding to incoming SIP requests and returning corresponding SIP responses.

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.

Figure 2–1 Servlet API and SIP Servlet API



SIP Servlets are similar to HTTP Servlets, and HTTP servlet developers can easily adapt to the programming model. The service level defined by both HTTP and SIP Servlets is very similar, allowing for the design of applications that support both HTTP and SIP.

Developing SIP Servlets

This section describes general techniques for developing SIP servlets using Java Plain Old Java Objects (POJOs) and Java Enterprise Edition (EE) annotations as well as using legacy version 1.x techniques.

Developing SIP Servlets Using POJOs and Annotations

JSR-359 defines a standard method for creating SIP servlets using Java EE annotations in conjunction with POJOs, significantly reducing the amount of code required compared to the earlier 1.x API techniques. In addition, servlet descriptors such as web.xml and sip.xml are optional.

Example 2–1 illustrates a basic annotated SIP servlet POJO that handles a variety of SIP response and request methods.

Example 2–1 SIP Servlet POJO

```
import javax.inject.Inject;
import javax.servlet.sip.SipFactory;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServletResponse;
import javax.servlet.sip.annotation.Ack;
import javax.servlet.sip.annotation.AnyMethod;
import javax.servlet.sip.annotation.Bye;
import javax.servlet.sip.annotation.ErrorResponse;
import javax.servlet.sip.annotation.Invite;
import javax.servlet.sip.annotation.SipApplication;
import javax.servlet.sip.annotation.SipServlet;
import javax.servlet.sip.annotation.SuccessResponse;
import java.io.IOException;

@SipServlet(loadOnStartup = 1)
public class CallHandler {

  @Inject SipFactory sipFactory;

  @Invite
  public void handleInvite(SipServletRequest request) throws IOException {
    // Handle a SIP invite on an incoming Request...
  }

  @Ack
```

```
  public void handleAck(SipServletRequest request) throws IOException {
    // Handle a SIP ACK on an incoming Request...
  }

  @AnyMethod
  public void handleAllRequests(SipServletRequest request) throws IOException {
    // Handle any generic method on an incoming Request...
  }

  @AnyMethod
  public void handleAllResponses(SipServletResponse resp) throws IOException {
    // Handle any other SIP response for any other method...
  }

  @SuccessResponse
  @ErrorResponse
  @Bye
  public void handleByeResponse(SipServletResponse resp) throws IOException {
    // Handle SIP responses in the case of success, error, or BYE...
  }

}
```

In Example 2–1, the class CallHandler imports the necessary SIP annotation libraries, and is itself annotated with @SipServlet indicating that it is a SIP Servlet. CallHandler then exposes the following public methods:

- **handleInvite**(*request*): Annotated with @Invite, handles any incoming SIP INVITE requests.
- **handleAck**(*request*): Annotated with @Ack, handles any incoming SIP ACK request.
- **handleAllRequests**(*request*): Annotated with @AnyMethod, handles any incoming SIP request method that is not an ACK or an INVITE. The methods annotated with @Invite and @Ack are more specific, and thus have higher precedence than @AnyMethod.
- **handleAllResponses**(*response*): Annotated with @AnyMethod, handles any incoming SIP response. Handles any responses other than success, error, or BYE which are handled by the more specifically annotated **handleByeResponse**().
- **handleByeResponse**(*response*): Annotated with @SuccessResponse, @ErrorResponse, and @Bye, and therefore handles any SIP success, error responses, or BYE. Any other SIP responses are handled by general **handleAllResponses**().

For more information on developing SIP servlet POJOs, see Chapter 3, "SIP Servlet POJOs".

Developing Legacy SIP Servlets

Example 2–2 shows an example of a simple SIP servlet using the legacy 1.x SIP Java API.

Example 2–2 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();
    }
}
```

In Example 2–2 the 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 Example 2–2, the MESSAGE method). Application developers override and implement necessary methods.
3. The life cycle and management methods (init, destroy) of SIP Servlet are exactly the same as HTTP Servlet. Manipulation of sessions and attributes is also the same.

Overview of the Differences Between HTTP Servlets and SIP Servlets

SIP servlets differ from typical HTTP servlets used in web applications in the following ways:

- HTTP servlets have a particular context (called the context-root) in which they run, while SIP servlets have no context.
- HTTP servlets typically return HTML pages to the requesting client, while SIP servlets typically connect SIP-enabled clients to enable telecommunications between the client and server.
- SIP is a peer-to-peer protocol, unlike HTTP, and SIP servlets can originate SIP requests, unlike HTTP servlets which only send responses to the originating client.
- SIP servlets often act as proxies to other SIP endpoints, while HTTP servlets are typically the final endpoint for incoming HTTP requests.
- SIP servlets can generate multiple responses for a particular request.
- SIP servlets can communicate asynchronously, and are not obligated to respond to incoming requests.
- SIP servlets often work in concert with other SIP servlets to respond to particular SIP requests, unlike HTTP servlets which typically are solely responsible for responding to HTTP requests.

Detailed Differences from HTTP Servlets

This section describes detailed differences between SIP Servlets and HTTP Servlets.

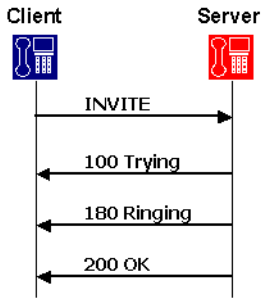
Multiple Responses

You might notice in Example 2–2 that the doMessage method has only one argument. In HTTP, a transaction consists of a pair of request and response messages, 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 2–2 Example of Request and Response in SIP



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

The legacy 1.x SIP Servlet API 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 doRegister(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);
protected void doUpdate(SipServletRequest req);
protected void doRefer(SipServletRequest req);
protected void doPublish(SipServletRequest req);
```

Likewise, the 2.0 SIP Servlet API defines the following annotations:

```
@Invite
@Ack
@Options
@Bye
@Cancel
@Register
@Prack
@Subscribe
@Notify
```

```
@Message
@Info
@Update
@Refer
@Publish
```

Receiving Responses

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

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

Figure 2–3 Relationship between Client and Server in SIP

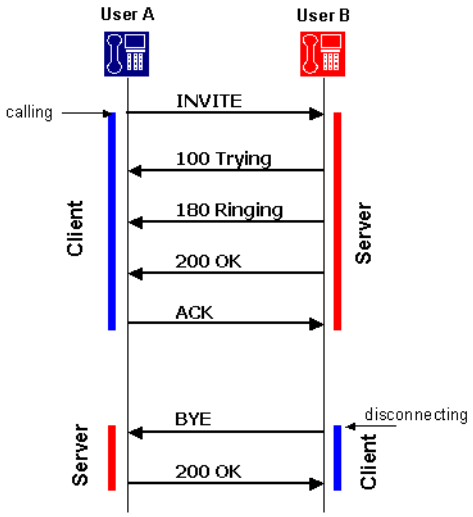


Figure 2–3 indicates that a calling or disconnecting terminal acts as a client. In SIP, roles of a client and server can be changed in one dialog. This client function is called UAC (User Agent Client) and server function is called UAS (User Agent Server), and the terminal is called UA (User Agent). The legacy 1.x 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).

Likewise the 2.x SIP Servlet API defines the following annotations which behave identically to their 1.x counterparts:

```
@ProvisionalResponse
@SuccessResponse
@RedirectResponse
@ErrorResponse
```

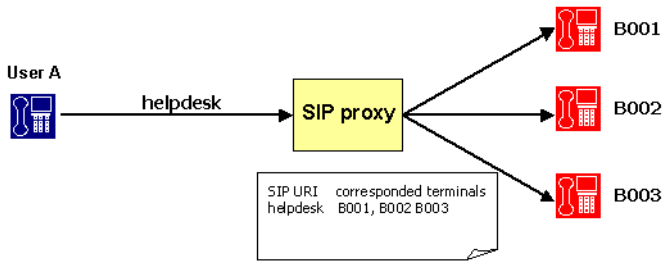
The use of methods to receive responses indicates that the SIP Servlet requests and responses are independently transmitted by the application using different threads. Applications must explicitly manage the association of SIP messages. The use of independent requests and responses makes the process more complicated, but enables you to write more flexible processes.

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

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 2–4 Proxy Forking



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

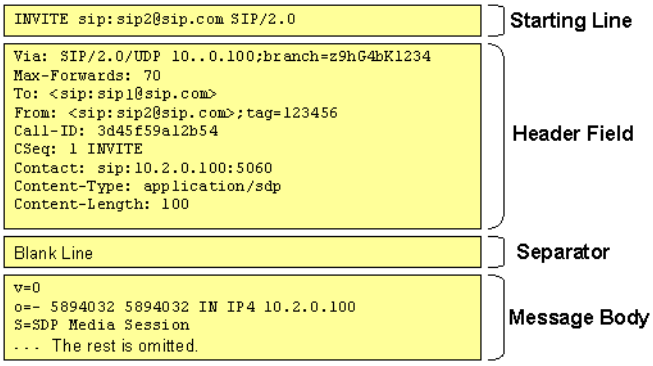
For more information on forking and SIP servlets, see "Forking SIP Requests".

Message Body

As Figure 2–5 illustrates, the contents of SIP messages is the same as the contents of HTTP messages. Both SIP and HTTP messages include:

- Starting line: Identifies the message as a request or a response. The starting line is also referred to as the *initial request line* or the *initial response line*.
- Header field: Provides information about the request or response.
- Separator: A blank line separating the header field from the message body.
- Message body: A message may have a body of data sent after the header lines. In a response, this is where the requested resource is returned to the client (the most common use of the message body).

Figure 2–5 SIP Message Example



HTTP is a protocol that transfers HTML files, images, and multimedia data. Contents to be transferred are stored in the message body. HTTP Servlet defines a stream manipulation-based API that enables the sending and receiving of these large-file content types.

Servlet Request

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

Servlet Response

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

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

In SIP, contents stored in the body include:

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

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

SipServletMessage

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

Role of a Servlet Container

The following sections describe major functions provided by Converged Application Server as a SIP servlet container:

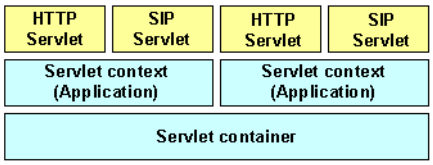
- Application Management: Describes functions such as application management by servlet context, life cycle management of servlets, application initialization by deployment descriptors.
- SIP Messaging: Describes functions of parsing incoming SIP messages and delivering to appropriate SIP servlets, sending 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.

Application Management

Like HTTP servlet containers, SIP servlet containers manage applications by servlet context (see Figure 2–6). Servlet contexts (applications) are normally archived in a WAR format and deployed in each application server.

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

Figure 2–6 Servlet Container and Servlet Context



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

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

Table 2–2 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. Optional when using SIP servlet POJOs.
WEB-INF/sip.xml	The SIP Servlet-defined configuration files for the SIP application. Optional if a SIP servlet POJO is using the programmatic deployment methods of the **SipServletContext** interface, **addServletPojo**(). See the *Converged Application Server Java API Reference* for more information.

Table 2–2 (Cont.) File Structure Example of Application

File	Description
WEB-INF/classes/	Store compiled class files in the directory. You can store both HTTP and SIP servlets in this directory.
WEB-INF/lib/	Store class files archived as Jar files in the directory. You can store both HTTP and SIP servlets in this directory.
***.jsp**, ***.jpg**	Files comprising the Web application (for example JSP) can be deployed in the same way as Java EE.

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

Example 2–3 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, life cycle of the servlet context is maintained by the servlet container. Although the servlet context is normally started and shutdown when the server is started and shutdown, the system administrator can explicitly start, stop, and reload the servlet context.

Note: The **web.xml** file is optional for servlets developed as annotated POJOs.

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 within that SIP session received after that point are considered as subsequent requests.

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

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

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

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

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

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

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

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

Table 2–3 Session-Related Terminology

Session Name	Description
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 because while dialogs and SIP sessions are similar in scope, their exact lifecycles are different.

Table 2–3 (Cont.) Session-Related Terminology

Session Name	Description
Application Session	A means for applications using multiple protocols and dialogs to associate multiple HTTP sessions and SIP sessions. Also called "APP session."

Converged Application Server automatically execute the following response and retransmission processes:

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

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

Applications typically do not need to explicitly set and see header fields in HTTP Servlet, as HTTP Servlet containers automatically manage fields such as Content-Length and Content-Type. SIP Servlet provides the same header management functions.

In SIP, however, since important information about message delivery exists in some fields, these headers are not allowed to change by applications. Headers that can not be changed by SIP Servlets are called system headers. Table 2–4 below lists system headers:

Table 2–4 System Headers

Header Name	Description
Call-ID	Contains ID information to associate multiple SIP messages as Call.
Address	Can be modified except for the host/port and scheme part in the Address URI. This restriction does not apply to the To/From headers. Protected parameters and their values cannot be modified.
From, To	Contains Information on the sender and receiver of the SIP request (SIP, URI, etc.). Modifiable except for the tag parameters.
CSeq	Contains sequence numbers and method names.

Table 2–4 (Cont.) System Headers

Header Name	Description
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. Can only be modified by adding or removing non-protected parameters.
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. Only the following messages can be modified or set by an application: ■ REGISTER requests and responses ■ 3xx responses ■ 485 responses ■ 200/OPTIONS responses

Utility Functions

JSR-359 defines the following utilities, which are available to SIP servlets:

1. SIP Session, Application Session
2. SIP Factory
3. Proxy
4. SipSessionsUtil
5. DnsResolver
6. TimerService
7. SipSecurity

SIP Session, Application Session

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

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

For SIP servlet POJOs, this is handled by the @SipServlet and @SipApplication annotations.

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 2–5 Objects Generated with SipFactory

Class Name	Description
URI, SipURI, Address	Can generate address information including SIP URI from String.

Table 2–5 (Cont.) Objects Generated with SipFactory

Class Name	Description
SipApplicationSession	Creates a new application session. It is invoked when a SIP servlet starts a new SIP signal process.
SipServletRequest	Used when a SIP servlet acts as UAC to create a request. Such requests can not be sent with Proxy.proxyTo. They must be sent with SipServletRequest.send.

SipFactory is located in the servlet context attribute under the default name. You can retrieve it with the following code:

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

For SIP servlet POJOs, this functionality is handled by the @SipFactory annotation. The @SipFactory annotation can be used in place of a ServletContext lookup for the SipFactory from within a Servlet above. The injected SipFactory appears as **sip**/*appname*/**SipFactory** in the application-scoped JNDI tree, where the *appname* is the name of the application.

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 (recursive): 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 JSR-359.
- Supervising mode: In the event of the state change of proxying (response receipts), an application reports this.

For more information, see "Forking SIP Requests".

SipSessionsUtil

A utility class providing additional support for converged HTTP/SIP applications and converged Java EE/SIP applications. This class can be accessed through the ServletContext parameter named javax.servlet.sip.SipSessionsUtil or it can be injected using the @Resource annotation. The injected SipSessionsUtil appears as **sip**/*appname*/**SipSessionsUtil** in the application-scoped JNDI tree, where the *appname* is the name of the application.

For more information, see the Java SIP Servlet API 2.0.

DnsResolver

For information on the DnsResolver utility, see "Annotation for DnsResolver Injection".

TimerService

The @Resource annotation also can be used to inject an instance of the TimerService for scheduling timers. This annotation can replace the following ServletContext based lookup of the TimerService:

```
TimerService t = (TimerService) getServletContext().getAttribute(TIMER_SERVICE);
```

The injected TimerService appears as **sip**/*appname*/**TimerService** in the application-scoped JNDI tree, where the *appname* is the name of the application.

SipSecurity

This annotation is used on a Servlet implementation class to specify security constraints to be enforced by the container on SIP protocol messages. The SIP servlet container will enforce these constraints on the annotated SIP servlet. The @SipSecurity annotation provides an alternative mechanism for defining access control constraints equivalent to those that could otherwise have been expressed declaratively via security-constraint elements in the portable deployment descriptor.

Note: If both security-constraint of the deployment descriptor and the @SipSecurity annotation is present for the same servlet, then the configuration in the deployment descriptor will take precedence.

For information on using the @SipSecurity annotation see section 22.3.10.1 in JSR-359, https://jcp.org/en/jsr/detail?id=359.

Internetworking with Third Party Protocols

A Session Initiation Protocol (SIP) application instance may consist of multiple protocol interactions. Each of these protocol interactions may be satisfied by a protocol library, that exposes its own Java API. As with SIP servlets or SIP Plain Old Java Objects (POJOs) acting as protocol listeners for SIP, these other protocol libraries can expose their own protocol listeners. For example, a Diameter library may expose a POJO that listens in on Diameter messages. Those messages may belong to a protocol session maintained by the Diameter protocol library. A SIP application archive, thus, may contain these protocol listeners as well. Furthermore, those protocol listeners can send SIP messages, while SIP POJOs might send other protocol messages as well.

For a detailed example of integrating with the Diameter protocol, see "Working with Diameter Applications Using CDI and POJOs" in *Converged Application Server Diameter Application Development Guide*.

SIP Servlet Concurrency

Converged Application Server includes concurrency utilities to help you create portable, reliable, thread safe applications. For information on handling SIP servlet concurrency, see Chapter 7, "SIP Servlet Concurrency".

Resolving Telephone Numbers to SipURI

ENUM (E.164 Number Mapping as specified RFC 6116, https://tools.ietf.org/rfc/rfc6116.txt) is a system that uses the Domain Name Service (DNS) to translate telephone numbers, like '+12025552600', into URIs. A TelURL or SipURI with a *user=phone* parameter constitutes a telephone number. Often, SIP servlet applications need to resolve such telephone numbers using ENUM as

specified in RFC 3824, https://tools.ietf.org/html/rfc3284. The SIP servlet API provides a DnsResolver interface to resolve telephone numbers in a TelURL or SipURI that has a *user=phone* parameter. DnsResolver is implemented by Converged Application Server, and is made available to applications as a ServletContext parameter with the name **javax.servlet.sip.DnsResolver**. It can also be accessed using resource injection as described in "Annotation for DnsResolver Injection". After a SIP servlet obtains a DnsResolver, it can use the following methods to resolve the telephone numbers to a SipURI:

- `SipURI resolveToSipURI(URI uri)`
- `List<SipURI> resolveToSipURIs(URI uri)`
- `List<String> resolveToStrings(URI uri, String enumService)`

DnsResolver also contains utility methods that help applications while resolving the URIs. The **toEnum**(URI *uri*) method helps applications to get the representation of a URI in ENUM format. The **resolvesInternally**(SipURI *uri*) method helps applications decide whether the SipURI resolves to the internal container.

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Annotation for DnsResolver Injection

The @Resource annotation defined in *Common Annotations for the Java Platform* (JSR 250, https://jcp.org/en/jsr/detail?id=250) is used to inject an instance of the DnsResolver utility class for DNS Enum lookup of telephone numbers.

This annotation can be used in place of the ServletContext based lookup for the DnsResolver.

The ServletContext lookup in Example 2–4,

Example 2–4 ServletContext Based DnsResolver Lookup

```
DnsResolver s = (DnsResolver)
  getServletContext().getAttribute("javax.servlet.sip.DnsResolver");
```

is equivalent to the annotation based implementation in Example 2–5.

Example 2–5 Annotation Based DnsResolver Lookup

```
@Resource
DnsResolver resolver;
```

3

SIP Servlet POJOs

This chapter describes creating Session Initiation Protocol (SIP) servlets using Java Enterprise Edition (EE) annotations in conjunction with Plain Old Java Objects (POJOs).

About SIP Servlet POJOs

SIP Servlet POJOs are SIP Servlets that do not extend from generic Servlets defined by the GenericServlet interface. They are simple POJOs annotated with an @SipServlet annotation. These POJOs contain annotated methods invoked by Converged Application Server when a SIP message arrives.

Any Java class that is annotated with @SipServlet, but does not extend from javax.servlet.sip.Servlet, is a SIP Servlet POJO. SIP Servlet POJOs support all elements of the @SipServlet annotation. The SIP Container treats POJOs similar to a component class listed in Table EE.5-1 of Java EE specification. These classes act as a Common Dependency Injection (CDI) managed bean and hence support all CDI capabilities explained in the Java EE CDI specification.

The SIP Servlet POJO Life Cycle

SIP Servlet POJOs are instantiated by the procedure to create a non-contextual instance that is not managed by the CDI container. The exact procedure is explained at the end of section titled “Support for Dependency Injection” in the Java EE specification.

The POJO life-cycle closely follows those of other component classes in terms of instantiation and destruction. Thus, shortly after resource and CDI injection completes successfully, the @PostConstruct callback is invoked. Similarly, the @PreDestroy annotation may be applied to one method that is called when the class is taken out of service and is longer be used by the container. Otherwise, the load-on-startup behavior of SIP Servlets applies to SIP Servlet POJOs as well. The behavior can be specified either in the @SipServlet annotation or in the deployment descriptor.

SIP Meta Annotations

A SIP Servlet POJO uses annotated methods to handle SIP Messages. Any Java method annotated with one or more annotations that carry SIP meta-annotations described in this section is used by the container to deliver messages to the application. SIP servlet containers do not depend on individual annotations that use these meta annotations enabling further extensibility as explained in "Extensibility Using SIP Meta-Annotations".

@SipMethod

The @SipMethod annotation associates the name of a SIP method with an annotation. A Java method annotated with a runtime annotation that is itself annotated with SipMethod handles SIP requests or responses with the indicated SIP method. The value of the annotation specifies the name of the SIP method (For example, "INVITE"). If the annotation is specified on a method whose first parameter is not a *SipServletRequest* or *SipServletResponse*, a deployment error occurs.

@SipResponseCode

The @SipResponseCode annotation associates a response code with an annotation. A Java method annotated with a runtime annotation that is itself annotated with @SipResponseCode handles SIP responses with the specified code. The value of the annotation specifies the response code. If an annotation is specified on a method whose first parameter is not *SipServletResponse*, a deployment error occurs.

@SipResponseRange

The @SipResponseRange annotation associates a response filter with an annotation. A Java method annotated with a runtime annotation that is itself annotated with @SipResponseRange handles SIP responses satisfying the filter. If an annotation is specified on a method whose first parameter is not *SipServletResponse*, a deployment error occurs. The specified range includes both beginning and end values. The element *begin* specifies the beginning of the response range, and the element *end* specifies the end of the response range.

@SipPredicate

The @SipPredicate annotation applies a predicate with a Java method in a SIP Servlet POJO. When a Java method is annotated with runtime annotations that are annotated with @SipPredicate, then those predicates are evaluated by Converged Application Server before executing the method. Thus, while other meta-annotations allow applications to define their own annotations, @SipPredicate enables applications to provide further filtering of messages based on specific logic. For example, an application can filter decisions based on the value of the SIP header or the state of the SIP Session.

The value of the annotation is a type of implementation of javax.servlet.sip.Predicate. Converged Application Server instantiates the class and invokes the **Predicate.apply**() method to determine whether or not to invoke the method.

Method Specific Annotations

Annotations are defined for each SIP method, such as INVITE, ACK, and REGISTER. Each runtime annotation is annotated with the SIP meta annotation @SipMethod.

A method annotated with @Invite is invoked by the SIP Container for SIP requests or SIP responses based on the type of the first parameter of the method. When the parameter is of type SipServletRequest.class, it is invoked for all SIP requests with the method INVITE. When the parameter is of type SipServletResponse.class, it is invoked for all SIP responses for the method INVITE.

Example 3–1 shows a SIP Servlet POJO that uses the @Invite annotation.

Example 3–1 @Invite Annotation

```
@SipServlet
public class ExamplePOJO {
  @Invite
  public void handleInviteRequest(SipServletRequest request) {
    //...
  }
  @Invite
  public void handleInviteResponse(SipServletResponse response) {
    //...
  }
}
```

The following method specific annotations are implemented:

- @Invite
- @Ack
- @Options
- @Bye
- @Cancel
- @Register
- @Prack
- @Subscribe
- @Notify
- @Message
- @Info
- @Update
- @Refer
- @Publish

For descriptions of the associated SIP request methods, see "SIP Requests".

@AnyMethod Annotation

The @AnyMethod annotation can handle SIP messages corresponding to any SIP method. Applications may use @AnyMethod annotation to receive all SIP requests and SIP responses without checking the method. If the annotation is specified on a method whose first parameter is not a *SipServletRequest* or *SipServletResponse*, a deployment error occurs.

Response Filtering

JSR-359 defines annotations for filtering responses. The defined annotations are @ProvisionalResponse, @SuccessResponse, @RedirectResponse, and @ErrorResponse. These annotations use the SIP meta-annotation @SipResponseRange to specify the response code range.

Table 3–1 lists the Response annotations and their associated response ranges.

Table 3–1 Response Range Annotations

Annotation	Response Range Begin	Response Range End
@ProvisionalResponse	101	199
@SuccessResponse	200	299
@RedirectResponse	300	399
@ErrorResponse	400	699

Example 3–2 shows the usage of a response range annotation.

Example 3–2 Response Range Annotation

```
@SuccessResponse
public void handleSuccessResponse(SipServletResponse response) {
  //...
}
```

To further filter responses, a POJO may combine both a method specific annotation and a Response Filter annotation. Example 3–3 shows how to handle success responses for INVITE messages.

Example 3–3 Combining Annotations

```
@Invite @SuccessResponse
public void handleInviteSuccessResponse(SipServletResponse response) {
  //...
}
```

It is also possible to specify multiple response filter annotations to a Java method, allowing application developers to handle multiple ranges using the same annotation. Example 3–4 shows how to handle multiple response ranges.

Example 3–4 Multiple Response Range Annotations

```
@Invite @ProvisionalResponse @SuccessResponse
public void handleInviteSuccessResponse(SipServletResponse response) {
  //...
}
```

@BranchResponse Annotation

An application can use the built-in SipPredicate, @BranchResponse, to associate an intermediate final response that arrives on a ProxyBranch with a Java method in a SIP Servlet POJO.

Example 3–5 shows the @BranchResponse definition.

Example 3–5 @BranchResponse Definition

```
@Retention(RUNTIME)
@Target({METHOD})
@SipPredicate(BranchResponse.Predicate.class)
public @interface BranchResponse {
  class Predicate implements
  javax.servlet.sip.Predicate<SipServletResponse> {
    @Override
    public boolean apply(final SipServletResponse response) {
      return response.isBranchResponse();
```

```
    }
  }
```

Extensibility Using SIP Meta-Annotations

Meta-annotations provide built-in extensibility. SIP meta-annotations allow application developers to define and use their own annotations. For example, an application can define an annotation called foo.example.18xResponses for handling only 18x responses. A single annotation can also contain more than one SIP meta-annotation.

Example 3–6 shows a user defined annotation.

Example 3–6 User Defined Annotation

```
@Retention(RUNTIME)
@Target({METHOD})
@SipResponseRange(begin = 200, end = 299)
@SipMethod("INVITE")
public @interface MySucessfulInviteResponse {
}
```

Example 3–7 shows an annotation that combines a @SipPredicate annotation with other meta annotations

Example 3–7 Combined Annotations

```
@Retention(RUNTIME)
@Target({METHOD})
@SipPredicate(MyInitialInvite.Predicate.class)
@SipMethod("INVITE")
public @interface MyInitialInvite {
  class Predicate implements
  javax.servlet.sip.Predicate<SipServletRequest> {
    @Override
    public boolean apply(final SipServletRequest request) {
      return request.isInitial();
    }
  }
}
```

The application can then use the @MyInitialInvite annotation to select a method for handling an initial invite as shown in Example 3–8.

Example 3–8 Using a Custom Annotation

```
@MyInitialInvite
public void handleInviteRequest(SipServletRequest request) {
  //...
}
```

Method Selection Precedence

Converged Application Server ensures that only one java method is invoked for a particular SIP message. When more than one Java method has matching annotations, Converged Application Server selects the Java method with most specific annotations.

For example, given a success response to an INVITE that could go to either of two methods, one of which matches any response to an INVITE, and the other which

matches only success responses to an INVITE, Converged Application Server chooses the second method. Those rules are applied in a particular order. For example, assume that two methods each take a *SipServletResponse*. If one method only matches the INVITE method, and another method only matches 200 OK responses, a 200 OK response to an INVITE goes to the INVITE-matching method because the SIP method matching has higher priority than response-code matching.

The following sections outline the procedure for finding the most specific Java method.

Precedence Rules Equation

The following shorthand is used to indicate specific annotations:

- **Jm**: A Java method annotated by annotations with SIP meta annotations.
- **Sm**: A @SipMethod annotation.
- **Sr**: A @SipResponseRange annotation.
- **Sc**: A @SipResponseCode annotation.
- **Sp**: A @SipPredicate annotation.
- **ESm**: (Method of the SipServletMessage) eq (value of Sm)
- **ESr**: (Status code of the SipServletResponse) in (range of Sr)
- **Esc**: (Status code of the SipServletResponse) eq (value of Sc)
- **ESp**: Result of evaluation of @SipPredicate.

Converged Application Server determine that a **Jm** meets the selection criteria if the annotations evaluate the following equation to *true*. If an annotation is not present in a sub-expression, the equation evaluates to *true*. Example 3–9 shows the complete precedence rules equation.

Example 3–9 Precedence Rules Equation

```
(ESm-1|| ESm-2 ||..||ESm-n) && ((ESc-1 || ESc-2 ||..||ESc-n) ||(ESr-1 || ESr-2 ||
..||ESr-n)) && (ESp-1 || ESp-2||..||ESp-n)
```

Conflict Resolution

If more than one **Jm** meets the selection criteria, Converged Application Server selects the **Jm** with the smallest count of **Sm**. If the number of **Sm**s is equal, the container selects the **Jm** with the smallest count of **Sc**. If the number of **Sc**s are equal, the container selects the one with the shortest span of **Sr**s.

Request Precedence Rules

The following rules apply to requests:

- **Sc** (SipResponseCode) and **Sr** (SipResponseRange) are not applicable to Requests.
- If none of the methods satisfy the basic rule, the container selects a Java method annotated with @AnyMethod.
- If none of the above is true, it results in the default behavior as specified by the section 2.3 of JSR-359.

Response Precedence Rule

The following rule applies to responses:

- If none of the methods satisfy the basic rule, the container selects a Java method annotated with @AnyMethod.

SipPredicate and Method Selection

If an application includes one or more @SipPredicate annotations, the application must ensure that the predicates are not written so as to cause conflict during selection of methods.

Deployment

During deployment, Converged Application Server scans the annotations present. If more than one Java method has the same SIP meta-annotation specificity using @SipMethod, @SipResponseCode, or @SipResponseRange, the container fails the deployment.

Conflict Resolution Examples

The following examples explain which method is selected when there is more than one Java method that match the selection criteria.

In Example 3–10, for a 200/INVITE response, Converged Application Server selects **handleResponse01**(), since that is the more specific of the two methods for that message. For a message with another SIP method, the container will select **handleResponse02**().

Example 3–10 Conflict Resolution Scenario One

```
@Invite
@SuccessResponse
public void handleResponse01(SipServletResponse response) {
  //...
}

@SuccessResponse
public void handleResponse02(SipServletResponse response) {
  //...
}
```

In Example 3–11, for a 200/INVITE response, the container will select **handleResponse02**(), since that is the more specific of the two methods for that message. For 201/INVITE message, the container will select **handleResponse01**().

Example 3–11 Conflict Resolution Scenario Two

```
@Invite
@SuccessResponse
public void handleResponse01(SipServletResponse response) {
  //...
}

@InviteOkResponse
public void handleResponse02(SipServletResponse response) {
  //...
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@SipMethod("INVITE")
@interface InviteOkResponse { }
```

In Example 3–12, for a 200 response **handleResponse03**() is selected since that is the only method that matches the response codes, and, therefore, there is no conflict. For a 201 response, both **handleResponse01**() and **handleResponse02**() match the criteria. However since **handleResponse01**() has a shorter response range than **handleResponse02**(), Converged Application Server chooses **handleResponse01**(). For a 302 response, Converged Application Server selects **handleResponse02**() since that is the only method that match the criteria.

Example 3–12 Conflict Resolution Scenario Three

```
@SuccessResponse
public void handleResponse01(SipServletResponse resp ) {
  //...
}

@NonFailureResponses
public void handleResponse02(SipServletResponse resp) {
  //...
}

@OkResponse
public void handleResponse03(SipServletResponse resp ) {
  //...
}
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseRange(begin = 100, end = 399)
@interface NonFailureResponses { }
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@interface OkResponse { }
```

Container Deployment Failures

The following examples show situations in which containers fail deployment since method precedence cannot be determined.

In Example 3–13, the container fails the deployment, since both the methods contain the same number of requests, and, for an INVITE request, the container will find two methods with the same specificity.

Example 3–13 Container Deployment Failure Scenario One

```
@Invite
@Message
public void handleRequest01(SipServletRequest req) {
  //...
}

@Invite
@Register
public void handleRequest02(SipServletRequest req) {
  //...
```

```
}
```

In Example 3–14, the container fails deployment because the **handleResponse02**() supports both SUBSCRIBE and INFO methods and 200 and 204 responses. The **handleResponse01**() method supports SUBSCRIBE and OPTIONS methods and 200 and 201 responses. Hence, for 200/SUBSCRIBE responses, the specificity of the SIP meta annotations is the same.

Example 3–14 Container Deployment Failure Scenario Two

```
@OptionsOkResponse
@SubscribeAccept
public void handleResponse01(SipServletResponse resp) {
  //...
}

@InfoOkResponse
@SubscribeNoNotification
public void handleResponse02(SipServletResponse resp) {
  //...
}
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@SipMethod("INFO")
@interface InfoOkResponse { }
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(200)
@SipMethod("OPTIONS")
@interface OptionsOkResponse { }
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(204)
@SipMethod("SUBSCRIBE")
@interface SubscribeNoNotification { }
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseCode(201)
@SipMethod("SUBSCRIBE")
@interface SubscribeAccept { }
```

In Example 3–15, the span of the response ranges are exactly the same and all other SIP meta annotations are the same in both the methods. For overlapping responses, SIP meta annotations in the two methods have the same specificity, and the container fails deployment.

Example 3–15 Container Deployment Failure Scenario Three

```
@InviteNonFailureResponses
public void handleResponse01(SipServletResponse resp) {
  //...
}
@Invite
@NonFailureFinalResponses
public void handleResponse02(SipServletResponse resp) {
  //...
}
@Retention(RUNTIME)
@Target(METHOD)
```

```
@SipResponseRange(begin = 100, end = 299)
@SipMethod("INVITE")
@interface InviteNonFailureResponses { }
@Retention(RUNTIME)
@Target(METHOD)
@SipResponseRange(begin = 200, end = 399)
@interface NonFailureFinalResponses { }
```

In Example 3–16, since all the SIP meta annotations of both methods are the same and the @SipPredicate annotation uses the same class, the container fails deployment.

Example 3–16 Container Deployment Failure Scenario Four

```
@Invite
@MySpecialResponses
public void handleResponse01(SipServletResponse resp) {
  //...
}
@InviteSpecialResponses
public void handleResponse02(SipServletResponse resp) {
  //...
}
@Retention(RUNTIME)
@Target(METHOD)
@SipMethod("INVITE")
@SipPredicate(MyPredicate.class)
@interface InviteSpecialResponses { }
@Retention(RUNTIME)
@Target(METHOD)
@SipPredicate(MyPredicate.class)
@interface MySpecialResponses { }
class MyPredicate implements Predicate<SipServletResponse> {
  @Override public boolean apply(SipServletResponse response) {
    //...
  }
}
```

4

Best Practices for SIP Applications

This chapter describes requirements and best practices for developing applications for deployment to Oracle Communications Converged Application Server.

Overview of Developing Distributed Applications for Converged Application Server

In a typical production environment, Session Initiation Protocol (SIP) applications are deployed to a Coherence cluster of Converged Application Servers. 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.

Use the SIP Concurrency Utilities

Converged Application 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 Converged Application Server architecture, make use of the Converged Application Server concurrency utilities as described in "SIP Servlet Concurrency".

If an application needs to access the Future object of a task that belongs to a SIP application session that is not the current SIP application session, create a non-scheduled managed task to access it. For more information about submitting a task with a different application session as its context, see "Specifying Application Session Programmatically".

For scheduled managed tasks, applications should always access future objects from the SIPApplicationSession interface so that they get consistent access to a task's state across the cluster. This approach is highly recommended if a task needs to be cancelled so as to avoid a locally-stored reference. Cancellation of a running task, using the Future or ScheduledFuture object, in interrupted mode does not always guarantee that the task is aborted. See the description of the **cancel** method in:

`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`

Customers should use appropriate concurrency modes based on their application needs so that messages and/or timers belonging to the same Sip application session do not lead to concurrency access issues, such as race conditions or deadlocks. Converged Application Server employs its own locking framework based on the chosen concurrency mode. Discrete multiple operations associated with the same Sip Application session may lead to multiple lock-and-unlock routines. Such a situation

should be avoided by combining these operations into a single non-scheduled managed task for that SIP application session.

Avoid nested lock situations. However, when applications employ nested SIP Application Session locks, the application logic associated with managing these locks must be designed with great care. Sometimes and in special circumstances, a nested SIP Application Session lock may lead to a deadlock. In such a situation, enabling the **wlss.concurrent** debug flag in the Converged Application Server Administration Console can assist you in your attempts to troubleshoot the issue. For information about setting the debug flag in Converged Application Server, see *Converged Application Server Administrator's Guide*.

Treat MessageListener Implementations as Read-Only

MessageListener implementations provided by Converged Application Server Java API SDK are solely meant for read-only purposes, such as logging. They must not be used for any type of SIP manipulation such as adding, removing, or modifying a session attribute or any other manipulation of a container-managed object.

The SIP container does not handle any sort of exception scenario; so it is not advisable for an application to rely on implementing the **MessageListener** interfaces for any type of SIP manipulation.

Local Data Structures Must Not Store Container-Managed Objects

Applications must not store container-managed objects for example, future objects, tasks, SIP request, and SIP response objects in local data structures.

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, Servlet method must not actively wait for data to be retrieved or written before returning control to the SIP Servlet container.

Servlets should also avoid sleep or wait operations in the body of a SIP method as such operations block the container thread and the callstate lock is held for the duration of each operation. Methods need to be written in such a way that control is returned back to the container and the flow is not compromised.

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, Converged Application Server cannot replicate the session state of the non-serializable objects.

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 engines. If you omit the `distributable` element, Converged Application Server does not deploy the SIP Servlet in a clustered environment.

Converged Application Server ignores the `distributable` element in non-clustered environments.

Use SipApplicationSessionActivationListener Sparingly

The `SipApplicationSessionActivationListener` interface can provide callbacks to an application when SIP Sessions are not active or activated. Keep in mind that callbacks occur only in a replicated Converged Application Server deployment.

Keep in mind that in a replicated deployment, Converged Application Server activates and passivates a SIP Session many times before and after the SIP messages are processed for the session. (This occurs normally in any 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.

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. For more information, see the discussion about "Clustering Best Practices" in the *Fusion Middleware Using Clusters for Oracle WebLogic Server*.

Optimizing Memory Utilization and Performance with Serialization

By default, Converged Application Server serializes and de-serializes call states, which optimizes your standalone domains for memory utilization. However, you can disable serialization in Converged Application Server to optimize your standalone domains for performance. You configure whether Converged Application Server uses serialization by using the `wlss.local.serialization` system property. This system property must be provided to the Java Virtual Machine (JVM) that starts Converged Application Server.

When you set the `wlss.local.serialization` system property to `true`, Converged Application Server optimizes a standalone domain for efficient memory utilization. Maintain this setting if memory utilization is of concern in your environment, especially in scenarios where the application session time-out values are large. Converged Application Server serializes the call state after a dialog is established and de-serializes that call state as it becomes necessary to do so. Note that performance may be impacted by the serialization and de-serialization of call states.

When you set the `wlss.local.serialization` system property to `false`, Converged Application Server optimizes a standalone domain for performance. Set this property to `false` when performance is critical in your environment and calls have fewer hold times or lower application session time-out values. Converged Application Server does not serialize or de-serialize call states. Note that, since the call state are held in memory for the life of each call, an increase in the hold times for the calls has an impact on memory utilization.

To enable or disable serialization:

1. Go to the *Domain_Home*/**bin** directory, where *Domain_Home* is the domain's home directory.
2. Open the **startWebLogic.sh** script in a text editor.

3. Change the `wlss.local.serialization` system property to `true` or `false` as required.
4. Save and close the file.

5

Composing SIP Applications

This chapter describes how to use Oracle Communications Converged Application Server application composition features.

Note: The Session Initiation Protocol (SIP) Servlet v2.0 specification (https://jcp.org/en/jsr/detail?id=359) describes a formal application selection and composition process, which is fully implemented in Converged Application Server. Use the SIP Servlet v2.0 techniques, as described in this document, for all new development. Application composition techniques described in earlier versions of Converged Application Server are now deprecated.

Converged Application Server provides backwards compatibility for applications using version 1.0 composition techniques, provided that:

- You *do not* configure a custom Application Router.
- You *do not* configure the Default Application Router properties.

Using the Application Router

Application composition is the process of "chaining" multiple SIP applications into a logical path to apply services to a SIP request. The SIP Servlet v2.0 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 Converged Application Server, all initial requests are first delivered to the AR, which determines the application used to process the request.

Converged Application Server supports the Default Application Router, which can be configured using a text file. Custom Application Routers are also supported. You create a Custom Application Router by implementing the `SipApplicationRouter` interface. A Custom Application Router can use complex processing to make routing decisions.

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

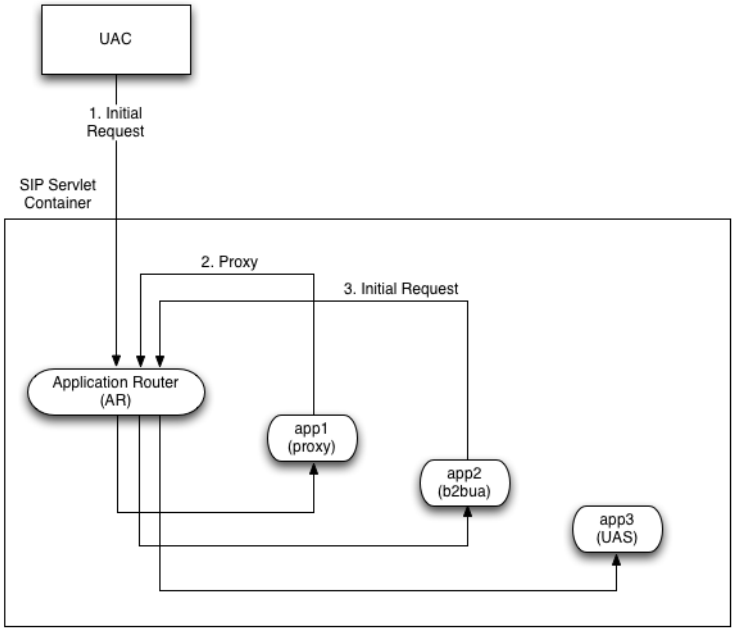
(UAS). If an initial request is proxied or relayed, the SIP container again forwards the request to the Application Router, which selects the next SIP application to provide a service for the request. In this way, the AR can chain multiple SIP applications as needed to process a request. The chaining 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 Converged Application Server container.

Using the Default Application Router

Converged Application Server includes a Default Application Router (DAR), which provides the basic functionality described in the SIP Servlet Specification v2.0 (http://jcp.org/en/jsr/detail?id=359), Appendix C: Default Application Router.

The DAR JSON Configuration File

Converged Application Server lets you use a simple configuration text file that follows JavaScript Object Notation (JSON)-based application composition rules. The configuration file consists of an array of chains, each containing a criteria followed by an array of applications that form the application chain in JSON form. The criteria is expressed using a JSON encoding of rule language specified in Appending D SIP Request Object Model of Java Specification Requests (JSR)-359.

In Example 5–1, the DAR invokes two applications on an INVITE request with the host part of the From header as “example.com”. The applications are identified by their names as defined in the application deployment descriptors. The subscriber identity returned is the URI from the From and To header respectively for the two applications. The DAR does not return any route to Converged Application Server and maintains the invocation state in the stateInfo as the index of the last application in the list.

Example 5–1 DAR JSON Configuration File

```
{
  "chains" : [
  {
    "description" : "Example Application Chain",
    "criteria" : {
      "and" :{
        "equal" : {
          "request.method" : "INVITE"
        },
        "contains" : {
          "ignore-case :"true"
          "request.from.uri.host" : "example.com"
          }
          }
          },
          "applications" : [
          { "name" : "OriginatingCallWaiting",
          "subscriber": "request.from",
          "region": "ORIGINATING",
          "route-modifier": "NO_ROUTE"},
          {"name" : "CallForwarding",
          "subscriber":"request.to",
          "region":"TERMINATING",
          "route-modifier":"NO_ROUTE"}
          ]
          } ]
          }
```

Example 5–2 shows a criteria element excerpted from a DAR JSON configuration file.

Example 5–2 DAR Criteria Element

```
"criteria" : {
  "and" :[
  {"equal" : {
  "request.method" : "INVITE" }},
  {"equal" : {
  "request.to.uri.port" : 5060 }},
  {"contains" : {
    "ignore-case :"true",
    "request.from.uri.host" : "example.com"
    }}
    ]
    }
```

The relevant JSON objects in a DAR configuration file are:

- **chains**: An array of application chains. Each element in the array represents one application chain.
- **description**: A description of the application chain.

- **criteria**: The criteria contains JSON encoded rules that are based on the object model specified in Appendix D SIP Request Object Model of JSR-359. When this predicate evaluates to *true*, the chain is chosen for handling the request.
- **applications**: An array of applications in this chain. This element contains the data for populating the SipApplicationRouterInfo object. Each application contains the following elements:
  - **name**: Name of the application as known to Converged Application Server.
  - **subscriber**: The SIP header which forms the identity of the subscriber that DAR returns. This is specified according to the object model in Appendix D SIP Request Object Model of JSR-359. Alternatively, it can return any string.
  - **region**: The routing region that can be one of the strings "ORIGINATING", "TERMINATING", or "NEUTRAL".
  - **routes**: An array of SIP URIs indicating the routes as returned by the Application Router. It can be an empty string.
  - **route-modifier**: A route modifier that can be one of the strings "ROUTE", "ROUTE_BACK", or "NO_ROUTE".

Legacy DAR Configuration Files

For backwards compatibility, Converged Application Server supports DAR property files as described in this section.

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 Converged Application 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, or it can be included in your server startup script.

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

Note that the Converged Application 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 Converged Application 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.

Configuring a Custom Application Router

In contrast to DAR, which is property-file driven, a Custom Application Router is implemented as a Java class, which allows for complex decision-making processes.

If you develop a custom Application Router, you must store the implementation for the AR in the `/approuter` subdirectory of the domain home directory. Supporting libraries for the AR can be stored in a `/lib` subdirectory within `/approuter`. (If you

have multiple implementations of `SipApplicationRouter`, use the `-Djavax.servlet.sip.ar.spi.SipApplicationRouterProvider` option at startup to specify which one to use.)

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

Converged Application Server provides several configuration parameters to specify the AR class and to pass initialization properties to the AR or AR. To configure these parameters using the Administration Console:

1. Log in to the Administration Console for your domain.
2. Select the **SIP Server** node in the left pane.
3. Click the **Configuration** tab and then select the **Application Router** subtab.
4. Use the options on the Application Router pane to configure the custom AR:
   - **Use Custom Application Router**: Select this option to use a custom AR instead of the Default AR. Note that you must restart the server after selecting or clearing this option, to switch between using the DAR and a custom AR.
   - **Use Json form configuration file**: Select this to load the AR configuration from a JavaScript Object Notation (JSON) file instead of a property file. For more information on the format of this file, see "The DAR JSON Configuration File".
   - **Custom Application Router filename**: Specify only the filename of the custom AR (packaged as a JAR) to use. The custom AR implementation must reside in the *`$DOMAIN_HOME`*`/approuter` subdirectory.
   - **Default application name**: The name of a default application that the container should call when the custom AR cannot find an application to process an initial request.

     If no default application is specified, the container returns a 500 error if the AR cannot select an application.

     > **Note:** You must first deploy an application before specifying its name as the value of Default application name.

   - **Application Router configuration data**: Enter properties to pass to the AR in the `init` method. The options are passed either to the DAR or custom AR, depending on whether the **Use Custom AR** option is selected.

     All configuration properties must conform to the Java Properties format. DAR properties must further adhere to the detailed property format described in Appendix C of the SIP Servlet Specification v1.1 (`http://jcp.org/en/jsr/detail?id=289`). Each property must be listed on a separate, single line without line breaks or spaces, as in:

     ```
     INVITE:("OriginatingCallWaiting","DAR:From","ORIGINATING","","NO_
     ROUTE","0"),("CallForwarding","DAR:To","TERMINATING","","NO_ROUTE","1")
     SUBSCRIBE:("CallForwarding","DAR:To","TERMINATING","","NO_ROUTE","1")
     ```

     You can optionally specify AR initialization properties when starting the Converged Application Server instance by including the

`-Djavax.servlet.sip.ar.dar.configuration` Java option. (To specify a property file, rather than a URI, include the prefix `file:///`) If you specify the Java startup option, the container ignores any configuration properties defined in **AR configuration data** (stored in **sipserver.xml**). You can modify the properties in **AR configuration data** at any time, but the properties are not passed to the AR until the server is restarted with the `-Djavax.servlet.sip.ar.dar.configuration` option omitted.

5. Click **Save**.

See Appendix C in the SIP Servlet Specification v2.0 (`http://jcp.org/en/jsr/detail?id=359`) for more information about the function of the AR. See also the SIP Servlet v2.0 API for information about how to implement a custom AR.

Application Router Behavior

When Converged Application Server receives an initial request from an external entity, or when an application acts as a User Agent Client (UAC) and sends an initial request with a new routing directive, the application selection process is started fresh. In that case, the Application Router is called with the following information:

- The SipServletRequest.
- The new routing directive.

Based on the supplied information, the configuration of which subscriber subscribes to which set of applications, and any other information that it may wish to use, for example, time of day, network condition, or external subscriber profile database, the Application Router returns the following information:

- The name of the selected application.
- The subscriber identity that the selected application is to serve.
- The routing region that the application serves.
- An array of zero or more routes to push (the routes array).
- A route modifier that tells the container how to interpret the route.
- An optional stateInfo serializable object.

The Application Router can return routes to Converged Application Server using its **SipApplicationRouterInfo.getRoutes**() method. The routes can be external or internal. External routes are used by the Application Router to instruct Converged Application Server to send the request externally. An internal route is returned when the Application Router wishes to modify the popped route header as seen by the application code, through the **SipServletRequest.getPoppedRoute**() method. When the request is received by Converged Application Server, the request may have had a Route header belonging to Converged Application Server, which it removes and makes available as described in "Popped Route Header". The route modifier returned by the Application Router tells Converged Application Server how to use the routes returned by it and also how the popped route needs to be presented.

The route modifier can be one of the following enum values:

- **ROUTE** indicates that **SipApplicationRouterInfo.getRoutes**() returns valid routes. Converged Application Server decides if they are external or internal. All of the routes returned must be of the same type, so Converged Application Server can make the determination by examining the first route only.

- **ROUTE_FINAL** indicates that **SipApplicationRouterInfo.getRoutes**() returns valid routes. The returned route can contain internal routes, external routes, or both. Converged Application Server pushes all routes returned by the Application Router.
- **ROUTE_BACK** directs Converged Application Server to push its own route before pushing the routes obtained from **SipApplicationRouterInfo.getRoutes**().
- **NO_ROUTE** indicates that Application Router is not returning any routes and the SipApplicationRouterInfo.getRoutes() value, if any, should be disregarded.

The behavior of the container with respect to the route modifiers is explained in "Procedure for Routing an Initial Request".

The **stateInfo serializable** object is useful for the Application Router to store state information from one invocation to the next. An Application Router implementation may choose to put any information in the **stateInfo** object, and this object is opaque to Converged Application Server and not accessible to the applications. Typically, an Application Router implementation may store information such as subscriber identity, the name of the application last invoked, and a precomputed list of applications that are to be invoked next. Application Router must not change the **stateinfo** after it is returned to Converged Application Server, and each result returned to Converged Application Server must contain a different **stateinfo** object.

If the selected application subsequently proxies or sends a new initial request based on the first one with a CONTINUE or REVERSE routing directive, the Application Router is called again. This time, in addition to the **SipServletRequest** and the routing directive, it is also supplied with the **stateInfo** object that it previously returned. In this way, the Application Router delegates the maintenance of the application selection state to Converged Application Server, and thus it can be stateless with respect to each initial request it processes.

If the Application Router determines that no application is selected to service a request, it returns null as the name.

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Order of Routing Regions

Because proximity of an application to its subscriber confers priority, it is beneficial for the management of feature interaction that originating applications are closest to the caller, and that terminating applications are closest to the callee. This can be satisfied if the following rules are followed:

- The originating region applications should be invoked first followed by terminating region applications.
- The applications that service a subscriber are contiguous (that is, no insertion of applications that service other subscribers in between).

On the other hand, it is entirely possible that the Application Router progresses directly to the terminating region if the caller is not a subscriber, or the caller does not subscribe to any applications. It is also possible that the application server does not serve any originating subscribers or has determined through some means that the originating applications have already been invoked and it should only look for terminating applications.

Inter-Container Application Routing

Converged Application Server supports applications distributed across multiple containers. The Application Router may return external routes in its **SipApplicationRouterInfo.getRoutes**() method that point to other application servers that it wishes the request to be routed to. Converged Application Server then pushes the routes onto the Route header stack of the request and sends the request externally. If this request arrives at another JSR-359 compliant container, it invokes the Application Router residing in that container so that the application selection process may continue. The first Application Router may pass any state information to the second Application Router by embedding it in the Route header. This is in accordance with the cascaded services model (SERL) as the applications can reside on different hosts and still participate in the application composition process.

Note: Some architectures require that the originating and terminating applications be hosted on different servers. The deployer can easily accomplish this by configuring the Application Routers such that one server hosts only originating applications and the other only terminating applications. Either the subscriber data can be partitioned such that the first server only serves originating users and the second serves terminating users. Alternatively, the Application Routers can collaborate by passing some state information in the Route headers, indicating for example that the first server has already completed the invocation of originating services.

Popped Route Header

On receiving an initial request that contains a SIP Route header (preloaded) or receiving a subsequent request with a Route header (converted from a Record-Route header), Converged Application Server determines if the request is intended for itself (based on local policy, for example IP addresses of interfaces or representative DNS entries). If it is, Converged Application Server removes the Route header before passing it to any application or the Application Router.

A side effect of removing a SIP Route message header before presenting the request to applications (and the Application Router) is that applications do not have access to the SIP Route message header and its associated information. Certain architectures utilize the SIP Route header for transporting application and other related information.

The following methods return the Route header popped by the container:

- **Address getPoppedRoute**();
- **Address getInitialPoppedRoute**();

If application composition is being used, the values returned by those methods may differ. The **getPoppedRoute** method returns the route popped before the current application invocation in the composition chain. The **getInitialPoppedRoute** method returns the route popped by Converged Application Server when it first received the request.

If no header is popped by Converged Application Serer on an initial request, both methods return *null*.

Both methods return the Route header as an **Address**, so, parameters added to the Record-Route header using the **Proxy.getRecordRouteURI**() method should be retrieved not from the popped route Address directly, but from the URI of the popped route Address.

Converged Application Server Behavior

Converged Application Server instantiates and initializes the Application Router and provides to it the initial list of deployed applications. When new applications are deployed or when applications are undeployed, Converged Application Server also inform the Application Router.

Converged Application Server receives an initial request from an external entity or from an application, and invokes the Application Router to obtain the name of the application to service the initial request, and then dispatches the request to the main servlet within the application. Converged Application Server also maintains application selection state including:

- The routing directive associated with this request.
- Routing region (originating, terminating, or neutral).
- Actions on the route returned from the Application Router in conjunction with the route modifier.
- Arbitrary, opaque state information returned from the Application Router.

Procedure for Routing an Initial Request

When Converged Application Server receives a new initial request, it first creates and initializes the various pieces of application selection state as follows:

- **Directive**:
  - If a request is received from an external SIP entity, the directive is set to NEW.
  - If a request is received from an application, the directive is set either implicitly or explicitly by the application.
- **Application router stateInfo**:
  - If a request is received from an application and the directive is CONTINUE or REVERSE, stateInfo is set to that of the original request with which this request is associated.
  - Otherwise, stateInfo is not set initially.
- **Subscriber URI**: Not set initially.
- **Routing Region**: Not set initially.

With the application selection state initialized, the following procedure is executed:

1. The **SipApplicationRouter.getNextApplication**() method of the Application Router object is called. The Application Router returns a **SipApplicationRouterInfo** object, named *result* for the purposes of this discussion.
2. The **result.getRouteModifier**() method is checked.
   - If **result.getRouteModifier**() is **ROUTE**, routes are retrieved using **result.getRoutes**().
     - If the first returned route is external (does not belong to Converged Application Server), all of the routes on the request's Route header stack are pushed, and the request is sent externally. Note that the first returned route becomes the top route header of the request.
     - If the first returned route is internal, Converged Application Server makes it available to the applications using the

SipServletRequest.getPoppedRoute() method and ignores the remaining ones, if any. That allows the Application Router to modify the popped route before passing it to the application.

- If **result.getRouteModifier**() is ROUTE_FINAL, all of the routes, regardless of whether the route is internal or external, are pushed on the request's route header stack and the request is sent.
- If **result.getRouteModifier**() is **ROUTE_BACK**, a route is pushed back to Converged Application Server followed by all of the routes obtained from **result.getRoutes**() and the request is sent externally. When the request eventually returns, Converged Application Server sets the routing directive to **CONTINUE** and also retrieve the Application Router state (*routingRegion*, *stateInfo*) and passes it in the call to **getNextApplication**() to continue processing the application chain.

  To retrieve the Application Router state, the container route in the request originally sent externally includes Application Router state (*routingRegion* and *stateinfo*) encoded as a route parameter.
- If **result.getRouteModifier**() is **NO_ROUTE**, result.getRoutes() is disregarded and processing continues.

3. The **result.getNextApplicationName**() is checked.
   - If **result.getNextApplicationName**() is not *null*:
     - The application selection state on the *SipSession: stateInfo* is set to **result.getStateInfo**(), *region* to **result.getRegion**(), and *URI* to **result.getSubscriberURI**().
     - A servlet is selected from the application.
   - If **result.getNextApplicationName**() is *null*:
     - If the *Request-URI* is not addressed to this container, or if there are one or more Route headers, the request is sent out according to the standard SIP mechanism.
     - If the *Request-URI* is addressed to Converged Application Server and there is no *Route* header, the request is not sent since it will cause a loop. Instead, Converged Application Server rejects the request with a 404 Not Found final response with no *Retry-After* header.

The effect of sending the request externally, for example, as a result of ROUTE_FINAL or ROUTE_BACK routing directive, can cause the message to come back to Converged Application Server and be presented to the Application Router again. Converged Application Server optimizes this case by calling **getNextApplication**() directly instead of sending the message.

Note: As a guideline, it is strongly recommended that applications to not rely on *Via* or *Record-Route* headers for their application logic, since SIP Servlet container compliance may vary. Applications should instead use the SipServletMessage methods **getLocalXXX**, **getRemoteXXX** if they are interested in the upstream entity.

If **SipApplicationRouter.getNextApplication**() throws an exception, Converged Application Server sends a 500 Server Internal Error final response to the initial request.

Application Router Packaging and Deployment

Converged Application Server loads and instantiates Application Router implementations. To be portable across containers, the Application Router implementation must be packaged in accordance with the rules specified by the Java SE Service Provider framework. Specifically, the JAR file containing the Application Router implementation must include the META-INF/services/javax.servlet.sip.ar.spi.SipApplicationRouterProvider file. The contents of that file indicate the name of the concrete public subclass of the javax.servlet.sip.ar.spi.SipApplicationRouterProvider class. The concrete subclass must have a no-arg public constructor.

As specified by the Service Provider framework, the providers may be installed by:

1. Including the provider JAR in the system classpath.
2. Including the provider JAR in the extension class path.
3. Container-specific means.

If the container uses classpath-based deployment, the first Application Router JAR file found in the classpath is installed. To avoid ambiguity when multiple Application Router implementations are present in the classpath, the **javax.servlet.sip.ar.spi.SipApplicationRouterProvider** system property can override loading behavior and force a specific provider implementation to be used.

Using the Legacy Custom Application Router

The Converged Application Server provides a built-in CAR that you can use. To use the CAR implementation, you supply configuration parameters to the CAR in the form of an XML file. The file specifies the applications in the chain, and the rules for targeting them.

The rules can impose conditions on application targeting based on factors such as the user identity or the request URI.

To use the prebuilt CAR, first create the configuration file that controls the behavior of the CAR.

After creating the configuration file, follow the steps listed in "Configuring a Custom Application Router" to apply the built-in CAR. In the configuration fields, provide the following values:

- For the custom AR filename, use **approuter-SDP.jar**
- In the AR configuration data field, specify the name of the configuration file you created as the value of the `configFileName` variable. For example:

  ```
  configFileName=./app.xml
  ```

- As the second line of the AR configuration data, enter the following:

  ```
  byPassIfAppIsNotWorking=true
  ```

The following section provides more information on the prebuilt CAR configuration file format.

Configuring the Legacy Custom Application Router

You control the prebuilt custom application router using an XML-based configuration file. The file lets you specify the application chain and the conditions for invoking the applications.

The configuration file is specified by `configFileName` property in the AR Configuration Data field of the UI.

You place the file in the following location:

Domain_Home/**approuter/lib**, where *Domain_Home* is the domain's home directory.

The schema definition for the configuration file is located in the same location. It is named **app-easydef.xsd**.

Example 5–3 shows a sample configuration for the prebuilt CAR implementation:

Example 5–3 Example Prebuilt CAR Configuration File

```
<app-router-conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oracle.com/sdp/easyapp-def easyapp-def.xsd"
    xmlns="http://www.oracle.com/sdp/easyapp-def" >
  <external-resource>
      <file>
         <file-path>./approuter/lib/user.properties</file-path>
      </file>
  </external-resource>
  <user-identity-header>From</user-identity-header>
  <terminating>
    <app>
       <app-name>basic-call-app</app-name>
       <index>0</index>
       <mapping-rule>
         <protocol>SIP</protocol>
 <pattern>
     <and>
             <equal><var>request.method</var><value>INVITE</value></equal>
<not><contains><var>request.uri</var><value>voicemail</value></contains></not>
           </and>
         </pattern>
         <subscriber-identity>.*</subscriber-identity>
         <request-uri>.*</request-uri>
       </mapping-rule>
    </app>
    <app>
       <app-name>presence-app</app-name>
       <index>1</index>
       <mapping-rule>
         <protocol>SIP</protocol>
 <pattern>
     <and>
             <equal><var>request.method</var><value>SUBSCRIBE</value></equal>
             <equal><var>request.method</var><value>PUBLISH</value></equal>
           </and>
         </pattern>
       <subscriber-identity>.*</subscriber-identity>
         <request-uri>.*</request-uri>
       </mapping-rule>
    </app>
    <app>
       <app-name>RouteToExternalURI</app-name>
       <index>2</index>
       <externalURI>sip:media@voicemail.com</externalURI>
       <mapping-rule>
         <protocol>SIP</protocol>
 <pattern>
     <and>
```

```
            <equal><var>request.method</var><value>INVITE</value></equal>
            <contains><var>request.uri</var><value>voicemail</value></contains>
          </and>
        </pattern>
       <subscriber-identity>.*</subscriber-identity>
        <request-uri>.*</request-uri>
       </mapping-rule>
    </app>
  </terminating>
</app-router-conf>
```

Notice the application named RouteToExternalURI, in the final app-name element. This is a symbolic application name that enables routing to an external URI. The CAR implementation adds the external URI to `SipApplicationRouterInfo`, which directs the container to route the request to the external URI. You can configure more than one special application, each with its own name, pattern, and index.

The pattern element syntax is the same as the pattern syntax used in **sip.xml**

The Application Router must provide the user identity of a subscriber to retrieve subscriber information from external sources. For the IMS environment, the P-Asserted-Identify header identifies the user by default. For non-IMS environments, the From header identifies the user. You can specify which header should be used to extract the user identity using the user-identity-header element.

For example:

```
<app-router-conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oracle.com/sdp/app-def app-def.xsd"
    xmlns="http://www.oracle.com/sdp/app-def" >
  <external-resource>
      <file>
         <file-path>./approuter/lib/user.properties</file-path>
      </file>
  </external-resource>
  <user-identity-header>From</user-identity-header>
```

To access a database, you must specify the JNDI name for the JDBC connection the SQL statement that selects the subscriber information.

For example:

```
  <external-resource>
      <rdbms>
          <jdbc-jndi-name>jdbc/OwlcsLs</jdbc-jndi-name>
          <sql>select appname from userapp where aor=?</sql>
      </rdbms>
  </external-resource>
```

In this example, `appname` identifies the column name in the table that contains the applications for the subscriber. The `aor` variable is the column name that represents the subscriber. The parameter for this SQL will be the P-Asserted-Identity header or From header of the initial request, as defined by the user-identity-header element.

If an HSS system is used as the external source, the diameter channel must be set up for each server, as specified in the *Converged Application Server Administrator's Guide*.

Additional information you need to specify in the configuration file includes:

- file-path: The diameter configuration file path
- service-indication: The Service-Indication AVP value which is defined in the 3GPP 29.328 section 7.4

- app-element: The customer AVP name. Its value is the applications subscribed to by the subscriber. In the context of the HSS, the app-element is the value of the ServiceData AVP.

The configuration file may be like:

```
  <external-resource>
     <hss>
        <file-path>./approuter/lib/hssconfig.xml</file-path>
        <service-indication>ARTest</service-indication>
        <app-element>apps</app-element>
     </hss>
  </external-resource>
```

The diameter configuration file, hssconfig.xml in the example, must comply with the OCCAS diameter.xml format, and be located in the following directory:

Domain_Home/**approuter/lib**, where *Domain_Home* is the domain's home directory.

An example of the hssconfig.xml file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>

<diameter xmlns="http://www.bea.com/ns/wlcp/diameter/300" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
  <configuration>
    <name>hssclient</name>
    <target>engine1</target>
    <target>engine2</target>
    <host>hssclient</host>
    <realm>bea.com</realm>

    <message-debug-enabled>true</message-debug-enabled>

      <application>
       <name>WlssShApplication</name>
        <class-name>com.bea.wcp.diameter.sh.WlssShApplication</class-name>
        <param>
          <name>destination.host</name>
          <value>hss</value>
        </param>
      </application>

      <peer>
        <host>hss</host>
        <address>10.182.101.206</address>
        <port>3900</port>
      </peer>
  </configuration>
</diameter>
```

Properties can also reside externally. This may be useful in testing or evaluation scenarios. In this case, you only need to configure the file path.

```
  <external-resource>
      <file>
         <file-path>./approuter/lib/user.properties</file-path>
      </file>
  </external-resource>
```

The file, **user.properties** in the example, should contain configuration information consisting of name-value pairs, and should be parsable by a Java Properties class. It should be located in the following directory:

Domain_Home/**approuter/lib**, where *Domain_Home* is the domain's home directory.

The format of each line in the user properties file should be subscriber name followed by the applications available to the subscriber.

For example:

```
alice@example.com=proxyregistrar,app2,app1
bob@example.com=proxyregistrar,app1
```

The index element specifies the order of invocation for the applications. The lower number has higher priority. The index must start at 0.

The mapping-rule element is used to determine if the application should be invoked by a special initial request. The value of subscriber-identity and request-uri must be Java regular expression. Only if the initial request matches all conditions, including protocol, pattern, subscriber-identity and request-uri, can the request be targeted to the application.

Session Key-Based Request Targeting

The SIP Servlet v2.0 specification 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.

Accessing SIP Applications Using SIP Application Index Keys

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 18-30 in the SIP Servlet Specification v2.0 (`http://jcp.org/en/jsr/detail?id=359`) for more information about using session key-based targeting.

Application Composition and SIP-HTTP Convergence

The @SipApplicationKey annotation is a useful tool you can employ to process HTTP requests in a converged module. You can set up a JavaServer page to manage the call

states of a SIP application, an HTTP servlet to process HTTP requests and a proxy application to proxy the call request to more than one user agent server (UAS).

Converged Application Server provides an example called *convergence_indexkey* that shows how SIP application session index keys can be used to access SIP applications. It uses application composition whereby more than one application is called to process the same SIP request; and also shows SIP-HTTP session convergence.

The *convergence_indexkey* example employs the JSR 289 API interfaces called javax.servlet.sip.ConvergedHttpSession and javax.servlet.sip.SipSessionsUtil. The example is packaged as an exploded EAR file containing a web application and the Enterprise Java Beans (EJB).

Note: The web application and EJB access do not require the configuration file **sip.xml**.

For more information about accessing *convergence_indexkey*, see "Examples".

Join and Replaces Header Support

Converged Application Server provides support for the use of both the Join and Replaces headers. To learn how to create SIP applications that use the functionality provided by the Join and Replaces headers, refer to the JSR 359 documentation and APIs.

About the Join Header

The Join header, defined in RFC 3911, is for use with SIP Call Control and Multi-Party applications. The Join header logically joins an existing SIP dialog with a new SIP dialog. You can use this to enable features such as Call Forwarding, Message Screening, and Call Center Monitoring.

The Join header contains information an application can use to match an existing SIP dialog to a new dialog. You can use the Join header to add a new dialog or SIP application session to an existing SIP application session in the same way that an encoded URI is used. This is achieved by setting the `call-id`, `to-tag`, and `from-tag` in the Join header of the SIP INVITE to match that of the existing dialog.

About the Replaces Header

The Replaces header, defined in RFC 3891, logically replaces an existing SIP dialog with a new SIP dialog. You can use this functionality to enable features such as Attended Call Transfer and Call Pickup.

The Replaces header contains information used to match and replace an existing SIP dialog (using the `call-id`, `to-tag`, and `from-tag`) to the newly created dialog. The Join header can be used to replace an existing SIP session associated with a SIP application session with a new dialog/session. This is achieved by setting the `call-id`, `to-tag`, and `from-tag` in the Replaces header of the INVITE to match that of an existing dialog.

Note: The SIP application must determine to send a BYE message using the original dialog. Converged Application Server does not automatically send a BYE message to terminate the original dialog.

Enabling Support for Join and Replaces Headers

Support for the Join and Replaces headers is disabled by default. If you have applications that need to use the Join and Replaces headers, you must enable Converged Application Server to handle these types of headers.

> **Note:** Enabling support for the Join and Replaces header may affect the performance of Converged Application Server. When enabling this feature ensure that your deployment of Converged Application Server has enough memory, computing power, and network bandwidth to function properly using Join- and Replaces-enabled applications.

To enable support for Join and Replaces headers, edit the entry for the `-Dwlss.dialog.index.enabled=false` command in the **startWebLogic.sh** script, and set its value to `true`. The **startWebLogic.sh** script is located in the *DOMAIN_HOME***/bin** directory, where *DOMAIN_HOME* is the domain's home directory. When support for Join and Replace headers is enabled, the entry in the **startWebLogic.sh** script appears as shown below:

```
-Dwlss.dialog.index.enabled=true
```

See the *Converged Application Server Administrator's Guide* learn more about the **startWebLogic.sh** script and the start-up options it controls.

6

Developing Converged Applications

This chapter describes how to develop converged Hypertext Transfer Protocol (HTTP) and Session Initiation Protocol (SIP) applications with Oracle Communications Converged Application Server.

Overview of Converged Applications

In a *converged application*, SIP protocol functionality is combined with HTTP or Java Platform, Enterprise Edition (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 Enterprise Archive (EAR) file. EAR is a file format used by Java EE for packaging one or more modules into a single archive so that the deployment of the various modules onto an application server happens simultaneously and coherently. It also contains XML files called deployment descriptors which describe how to deploy the modules. 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.

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.
- A **weblogic.xml** deployment descriptor may be included to configure Servlet functionality in the Converged Application Server container.
- Observe the following restrictions on deployment descriptor elements:

- The `distributable` tag can be present in **sip.xml** or specified using a SipApplication annotation.
- `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.

Examples

Converged Application Server includes sample converged applications when installed using the **Complete Installation** option. All source code, deployment descriptors, and build files for the examples are found in

*`OCCAS_home`***/occas/samples/sipserver/examples**

where, *OCCAS_home* is the directory in which the Converged Application Server software is installed. By default, *occas_home* is a subdirectory of *Oracle_home*; for example, *Oracle_home***/occas**. For more information on these placeholders, see Table1-1 in *Converged Application Server Installation Guide*.

For descriptions of the examples, source code, and build files, see `index.html` under the `src` sub-directory in the examples location.

7

SIP Servlet Concurrency

This chapter describes how to develop Session Initiation Protocol (SIP) Servlet applications that support concurrency in Oracle Communications Converged Application Server. It also explains ways to develop portable applications without concurrency issues.

Multiple Servlets executing simultaneously may have active access to shared resources like the SipSession and SipApplicationSession objects. These resources may also be accessed concurrently from ServletTimer objects, from other Java Platform, Enterprise Edition (Java EE) modules in a converged Java EE application, and by the SIP Servlet Container. Operations, such as updating values of session attributes, carried out by different threads on these objects can create concurrency issues, including deadlock, for some applications.

> **Note:** For details on the application program interface (API) described in this chapter, see the *Java SIP Servlet API 2.0* JavaDocs.

Specifying Concurrency Mode

A SIP Servlet application can specify the required level of concurrency control Converged Application Server should apply while executing applications with the @SipApplication annotation or in the deployment descriptor. Converged Application Server supports two values for the ConcurrencyMode enum: *VENDORSPECIFIC* and *APPLICATIONSESSION*.

The following shows an example of a servlet specifying the concurrency mode.

Example 7–1 Setting the Concurrency Mode

```
package com.example;
import javax.servlet.sip.SipServlet;
@SipApplication (name = "SampleApplication", concurrencyMode =
       ConcurrencyMode.APPLICATIONSESSION)
```

Table 7–1 lists the available concurrency modes.

Table 7–1 Concurrency Modes

Mode	Description
VENDORSPECIFIC	Indicates that Converged Application Server can choose any concurrency level it deems appropriate. The level of guarantee of thread safety is determined by Converged Application Server. This is the default.

Table 7–1 (Cont.) Concurrency Modes

Mode	Description
APPLICATIONSESSION	Indicates that Converged Application Server performs concurrency control at the level of the application session. It ensures that two messages belonging to the same application session are never processed simultaneously. It also ensures that various timer tasks or internal threads managed by Converged Application Server that access the application session are not executed at the same time.

Concurrency Utilities

The Concurrency Utilities for Java EE specification explains how an application can use Concurrency Utilities in an enterprise server environment. Converged Application Server uses those utilities to execute asynchronous tasks and to let you develop thread safe applications. A SIP Servlet application may use any of the Managed Objects specified "Default Managed Objects".

Propagating SipApplicationSession Context

When a SIP Servlet application specifies a concurrency control of **ConcurrencyMode.APPLICATIONSESSION**, Converged Application Server makes sure that appropriate concurrency control is maintained. To effectively maintain the concurrency control at the SipApplicationSession level, the active SIP application session also needs to be passed as context information.

To enable this, Converged Application Server exposes one or more **ManagedExecutorService** and **ManagedScheduledExecutorService** objects that execute submitted tasks in a thread pool managed by the Converged Application Server. The **ManagedExecutorService** object provides methods for submitting tasks for execution. And the **ManagedScheduledExecutorService** object provides methods for submitting delayed or periodic tasks for execution at specific times.

While executing the tasks, Converged Application Server makes sure that concurrency control specified by the application is maintained.

Note: Applications should use the default managed objects as specified in "Default Managed Objects", to propagate a SIP Application Session.

When a servlet (or any other part of the application) submits a task using the default **ManagedExecutorService** or **ManagedScheduledExecutorService**, apart from the context information specified in the Concurrency Utilities for EE specification, the active application session is also passed as the context information. The active application session is responsible for the execution of the thread where the application logic is running. Some examples are the application session of the SipServletMessage that triggered servlet execution, the application session of the executing ServletTimer, the relevant application session of the application listener, and the application session of the ConvergedHttpSession of the HTTP servlet.

If the task is submitted from a thread where no application session is active, then the application can use the mechanism specified in "Specifying Application Session Programmatically" for specifying an application session as the context.

To submit a scheduled task, the servlet container launches a new thread.

Specifying Application Session Programmatically

A thread might want to submit a task with a different application session as its context than the context active when the task was submitted. In that case, the application is expected to create a contextual object proxy using **ContextService** for submitting the task or by using **ManagedTask**. The application can then specify an application session of its choosing as the context by using one of the following execution properties:

- **javax.servlet.sip.ApplicationSessionKey**: Specifies the SIP application key.
- **javax.servlet.sip.ApplicationSessionId**: Specifies the application session ID.
- **javax.servlet.sip.ApplicationSession.create**: Indicates that the container creates a new SipApplicationSession and uses it as the context.

Applications can also access these constants from javax.servlet.sip.SipServlet fields SIP_APPLICATIONSESSION_KEY, SIP_APPLICATIONSESSION_ID, and SIP_APPLICATIONSESSION_CREATE respectively. Table 7–2 lists the actions that Converged Application server takes based on application settings.

Table 7–2 Application Settings and Actions Taken by Converged Application Server

Application Settings	Resulting Converged Application Server Actions
Both javax.servlet.sip.ApplicationSessionKey and javax.servlet.sip.ApplicationSessionId are specified by the application.	javax.servlet.sip.ApplicationSessionId given precedence.
Application session (specified by either javax.servlet.sip.ApplicationSessionKey or javax.servlet.sip.ApplicationSessionId) is invalid or cannot be found.	Container aborts the execution of the task. ManagedTaskListener.taskAborted is called throwing an AbortedException.
javax.servlet.sip.ApplicationSession.create, and javax.servlet.sip.ApplicationSessionId are specified by the application.	Discards javax.servlet.sip.ApplicationSession.create
Both javax.servlet.sip.ApplicationSession.create and javax.servlet.sip.ApplicationSessionKey are specified.	Uses the specified application session key to create the SipApplicationSession. If Converged Application Server finds an existing application session with the specified application session key, that application session will be used as the context.
Only javax.servlet.sip.ApplicationSession.create is specified.	Creates a new SipApplicationSession and uses that as the context.

To avoid concurrency issues, applications are required to submit asynchronous tasks whenever they use an application session different from the active application session. That rule applies for all kinds of application components, such as SIP Servlets, HTTP Servlets, other Java EE components, Servlet Timers, and asynchronous tasks.

Maintaining Thread Safety with Multiple Application Session Contexts

Converged Application Server ensures that tasks are run in a thread safe manner with respect to the SIP application session specified as the context. However, an application should be careful while sharing the objects between tasks with a different application session context. The thread safety of such objects, whether it is an application-specific object or an object received from the container, must be maintained by the application. For example, objects such as *SipURI*, or *Address* should be cloned before they are shared between tasks.

ContextService

As specified in section 3.3 of the Concurrency Utilities for Java EE specification, **javax.enterprise.concurrent.ContextService** allows applications to create contextual objects without using a managed executor. Just like a submitted contextual task, whenever a method on the contextual object is invoked, the method executes with the thread context of the associated application component instance.

If a contextual proxy created using the default SIP context service (see "Default Managed Objects") and if that contextual proxy is executed in a thread known to Converged Application Server (for example, an HTTP Servlet thread or a MDB thread), the application session context is set properly. This application session context is based on the thread that created the contextual proxy. The application may use the execution properties mentioned in "Specifying Application Session Programmatically" to use a different application session as the context.

Note: If the contextual proxy is executed directly on a thread that already has an active SIP application session and the contextual proxy is created with another SIP application session as the context, the container throws an IllegalStateException.

Default Managed Objects

Converged Application Server provides preconfigured, default managed objects shown in Table 7–3.

Table 7–3 Default Managed Objects

Java EE Object	Converged Application Server Object
ManagedExecutorService	java:comp/**ManagedSipExecutorService**
ManagedScheduledExecutorService	java:comp/**ManagedScheduledSipExecutorService**
ContextService	java:comp/**SipContextService**
ManagedThreadFactory	java:comp/**ManagedSipThreadFactory**

The types of contexts propagated by the default objects include naming context, classloader, and security information apart from the application session.

Note: An application must not use these default managed objects if it does not want the application session context to be propagated.

Accessing an Active Application Session

You may want to access the application session that is active on the current thread, whether it is from a thread running a submitted or scheduled task or from a thread executing the servlet.

It is possible to access SipApplicationSession using **SipSessionsUtil**, and the ID, or application key. Converged Application Server provides a more direct access to the active application with the **SipSessionsUtil.getCurrentApplicationSession**() method. This method is especially useful when the tasks are not Common Dependency Injection (CDI) managed beans where the CDI built-in beans for injecting SipApplicationSession cannot be used.

Accessing Tasks Futures

A Future represents the result of an asynchronous computation. The methods of the Future interface enable applications to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

Your application can access the Future (java.util.concurrent.Future) objects of submitted or scheduled tasks that are not completed nor canceled. For

- Submitted tasks:

  Asynchronous tasks are typically submitted to the **ManagedExecutorService** object. The **ManagedExecutorService** object returns an instance of the **Future** interface object. For information about:

  – **ManagedExecutorService**, see

  https://docs.oracle.com/javaee/7/api/javax/enterprise/concurrent/ManagedExecutorService.html

  – **Future**, see

  https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

- Scheduled Tasks

  Asynchronous tasks are typically scheduled with the **ManagedScheduledExecutorService** object. The **ManagedScheduledExecutorService** object returns an instance of the **ScheduledFuture** object. For information about:

  – **ManagedScheduledExecutorService**, see

  https://docs.oracle.com/javaee/7/api/javax/enterprise/concurrent/ManagedScheduledExecutorService.html

  – **ScheduledFuture**, see

  http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledFuture.html

  Important: Converged Application Server does not support both the get methods of the **ScheduledFuture** interface. Applications can use the lifecycle events associated with a **ManagedTaskListener**.

For information on the possible task lifecycle events that can occur when a **ManagedTaskListener** is associated with a task, see:

https://docs.oracle.com/javaee/7/api/javax/enterprise/concurrent/ManagedTaskListener.html

Accessing the Futures of Tasks in a Sip Application Session

Applications can access the Futures of tasks belonging to a **SipApplicationSession** by one of the following methods:

- **SipApplicationSession.getTaskFuture**(String *identityName*)

  This method returns the Future object of the task with the specified identity name, belonging to the application session.

https://sipservlet-spec.java.net/javadoc/2.0/final/javax/servlet/sip/SipApplicationSession.html#getTaskFuture(java.lang.String)

To use this method, applications are required to provide the unique identity name within the SipApplicationSession for **javax.enterprise.concurrent.ManagedTask.IDENTITY_NAME** execution property, when submitting or scheduling the task. For more information, see

https://concurrency-ee-spec.java.net/javadoc/javax/enterprise/concurrent/ManagedTask.html#IDENTITY_NAME

- **SipApplicationSession.getTaskFutures**()

  This method returns the Future objects of all submitted or scheduled task belonging to the application session. For more information on this method, see

  https://sipservlet-spec.java.net/javadoc/2.0/final/javax/servlet/sip/SipApplicationSession.html#getTaskFutures

About Saving Future Objects

Applications should not attempt to save the Future object of a scheduled task in the Sip application session. Doing so will cause the application server to throw the following error:

Example 7–2 Error Seen When Future Objects are Saved in Sip Application Sessions

```
java.lang.IllegalArgumentException: Attribute must be serializable for
distributable application
```

Concurrency Examples

Example 7–3 shows submitting a task with an active SipApplication context.

Example 7–3 Task with Active SipApplication Context

```
@SipServlet
public class ExamplePOJO1{
  @Resource(lookup="java:comp/ManagedSipExecutorService")
  ManagedExecutorService mes;
  @Invite
  protected void onInvite(SipServletRequest req) {
    // Create a task instance. MySipTask implements Callable...
    MySipTask sipTask = new MySipTask();
    // Submit the task to the ManagedExecutorService...
    Future sipFuture = mes.submit(sipTask);
  }
}
```

Example 7–4 shows submitting a task with a different SipApplicationSession context.

Example 7–4 Task with a Different SipApplication Context

```
@SipServlet
public class ExamplePOJO2{
  @Resource(lookup="java:comp/ManagedScheduledSipExecutorService")
  ManagedScheduledExecutorService mses;
  @Resource(lookup="java:comp/SipContextService")
  ContextService ctxSvc;
  @Invite
```

```
  protected void onInvite(SipServletRequest req) {
    SipApplicationSession sas = //...lookup using SipSessionsUtil
    // Set any custom context data through execution properties...
    Map<String, String> execProps = new HashMap<>();
    execProps.put("javax.servlet.sip.ApplicationSessionId", sas.getId());
    // Create a task instance. MySipTask implements Callable...
    MySipTask sipTask = new MySipTask();
    Callable task = ctxSvc.createContextualProxy
    (sipTask, execProps, Callable.class);
    // Submit the task to the ManagedScheduledExecutorService...
    Future sipFuture = mses.submit(task);
  }
}
```

Example 7–5 shows a more complete TaskHandler class.

Example 7–5 TaskHandler Class

```
@Dependent
public class TaskHandler {

  @Resource(lookup = "java:comp/SipContextService")
  ContextService sipCS;

  @Resource(lookup = "java:comp/ManagedSipExecutorService")
  ManagedExecutorService sipMES;

  @Inject
  SipSessionsUtil ssu;

  public void doIt(final String sasId, final String appState) {
    Map<String, String> props = new HashMap<>();
    props.put(SipServlet.SIP_APPLICATIONSESSION_ID, sasId);

    final SipSessionsUtil util = ssu;
    Runnable task = () -> {
      final SipApplicationSession session =
        util.getCurrentApplicationSession();
      session.setAttribute("counter", 1);
      session.setAttribute("appState", appState);
    };

    task = sipCS.createContextualProxy(task, props, Runnable.class);
    task.run();
  }

  public void doAsync(final String sasId, final String appState) {
    Map<String, String> props = new HashMap<>();
    props.put(SipServlet.SIP_APPLICATIONSESSION_ID, sasId);

    final SipSessionsUtil util = ssu;

    Runnable task = (Runnable & Serializable) () -> {
      final SipApplicationSession session =
        util.getCurrentApplicationSession();
      int counter = (int) session.getAttribute("counter");
      session.setAttribute("counter", ++counter);
      session.setAttribute("appState", appState);
    };
```

```
    task = (Runnable) sipCS.createContextualProxy
      (task, props, Runnable.class, Serializable.class);
    sipMES.submit(task);
  }
```

8

Managing Client Initiated Connections

This chapter describes how to manage client-initiated connections in Oracle Communications Converged Application Server.

The Session Initiation Protocol (SIP) outbound specification, as defined in RFC-5656, https://tools.ietf.org/html/rfc5656, specifies techniques for keeping connections that a User Application (UA) establishes alive, especially in environments where the UA is behind a Network Address Translation (NAT) device or a firewall. A SIP Servlet application may take different roles with respect to those defined in RFC 5626 for keeping the connections alive. For example, a SIP Servlet application may act as an authoritative proxy, an edge proxy, a registrar, or a UA. Refer to RFC 5626 for more details about the exact procedure for managing client-initiated connections.

Converged Application SIP Servlet containers support sending keep alive messages from a user agent as well as responding to keep alive messages defined in RFC 5626. They also support a FlowBinder interface for managing the connections as described in the following sections.

In RFC 5626, a flow is a transport-layer association between two hosts that is represented by the network address and port number of both ends and by the transport protocol. For Transport Control Protocol (TCP), a flow is equivalent to a TCP connection. For User Datagram Protocol (UDP), a flow is a bidirectional stream of datagrams between a single pair of Internet Protocol (IP) addresses and ports of both peers.

From a SIP Servlet application's perspective, the flow is a transport layer association between the SIP Servlet container and another SIP endpoint on a specific transport.

Table 8–1 describes the Flow interface's methods.

Table 8–1 Flow Interface Methods

Method	Description
getToken()	Retrieves the flow token representing a flow.
getLocalURI()	Retrieves the SipURI representation of the local end of transport association.
getRemoteURI()	Retrieves the SipURI representation of the remote end of transport association.
release()	Indicates to the container that the application will not use the Flow object any more.
isActive()	Indicates whether the flow is active or not. Flow becomes inactive when it is closed.

Applications may get an instance of a Flow that represents this transport layer association by using the **SipSession.getFlow**(), **Proxy.getFlow**(), or **ProxyBranch.getFlow**() methods.

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Retrieving a Flow Object from the Container

A SIP Servlet application can retrieve the Flow object corresponding to a flow token from the container by using the **SipServletContext.getFlow**(String *flowToken*) method.

Maintaining Connections Initiated by SIP User Agents

By default, when a SIP user agent located behind a NAT device or firewall initiates a connection, the SIP container keeps the connections alive. SIP user agents located behind a NAT are able to communicate with SIP nodes located on the other side of NAT.

The associated configuration parameter, **SIP Outbound Support**, is accessible in the Administration Console. The check box entry for **SIP Outbound Support** is selected, by default. You can disable the default support for connections that SIP user agents located behind a NAT initiate with SIP nodes located on the other side of NAT. To do so, the **General** configuration tab for **SIP Server** in the **Domain Structure** for the Administration Console and clear and the **SIP Outbound Support** check box.

If the check box for the **SIP Outbound Support** entry is cleared, all communication handling between user agents located behind a NAT and SIP nodes located on the other side of NAT is disabled. See "About Accessing the Administration Console" in *Converged Application Server Administrator's Guide*.

UAC Sending Keep-Alive

If a SIP Servlet acting as a User Agent Client (UAC) sends a REGISTER request with the headers required for RFC 5626 support, on receiving a 200 response to such a request, if that response contains an outbound option-tag in the Require header field, Converged Application Server starts sending keep-alive messages. These keep-alive messages may be double Carriage Return Line Feeds (CRLFs) for connection-oriented transports, such as TCP and Session Traversal Utilities for NAT (STUN) binding requests as described in RFC 5626.

Example 8–1 shows an implementation of such a UAC:

Example 8–1 UAC Keep-Alive

```
void sendRegister(SipServletRequest register) throws Exception {
  register.setHeader("Supported", "path,outbound");
  // Add the instance ID...
  Parameterable contact = register.getParameterableHeader("Contact");
  contact.setParameter("+sip.instance", <instanceId>);
  // Add the registration ID...
  contact.setParameter("reg-id", <registrationId>);
  register.send();
}

// The container starts sending keep-alive messages...
```

```
@SuccessResponse
public void handle200ok (SipServletResponse resp) {
  Flow flow = resp.getSession().getFlow();
}
```

Handling Flow Failures

An instance of FlowListener may be configured as a SipListener. Whenever the flow fails, the container invokes all configured FlowListeners. Applications may retrieve the flow and complete the necessary business logic, which typically includes sending the REGISTER message again.

Example 8–2 shows the FlowListener and FlowFailedEvent definitions.

Example 8–2 Flow Failure Handling

```
public interface FlowListener extends EventListener {
  void flowFailed(FlowFailedEvent flowFailedEvent);
}
public class FlowFailedEvent extends EventObject {
  public FlowFailedEvent(Flow flow) {
    super(flow);
  }
  public Flow getFlow() {
    return (Flow) super.getSource();
  }
}
```

Reusing a Flow

A UAC or a User Agent Server (UAS) may specify the flow to send messages on by using one of the following methods.

- **SipSession.setFlow**(Flow *flow*).
- **Proxy.setFlow**(Flow *flow*)
- **ProxyBranch.setFlow**(Flow *flow*)

For example, a UAC can specify that the same flow that is used for sending REGISTER messages be used for sending INVITE messages as shown in Example 8–3.

Example 8–3 Reusing a Flow

```
void sendInvite(SipServletRequest invite){
  Flow flow = // Retrieve the flow...
  invite.getSession().setFlow(flow);
  invite.send();
}
```

Implementing Edge Proxies

When an edge proxy is implemented using SIP Servlets, the application inserts the flow token into the request as explained in RFC 5626. The application can retrieve flow tokens from the Flow object for that purpose.

Example 8–4 illustrates how an application might implement this logic.

Example 8–4 Inserting a Flow Token in a Request

```
@Register
public void handleRegister(SipServletRequest request){
  Proxy proxy = request.getProxy();
  Flow flow = proxy.getFlow();
  proxy.setAddToPath(true);
  SipURI pathURI = proxy.getPathURI();
  pathURI.setUser("edge");
  pathURI.setParameter("flow", flow.getToken());
  pathURI.setParameter("ob", "true");
  proxy.proxyTo(request.getRequestURI());
}
```

Similarly, when a different request that needs to be forwarded to a UA arrives at the edge proxy, it can look up the Flow object using the flow token present in the request. After the retrieved flow object is set on the Proxy (or ProxyBranch) by using the **Proxy.setFlow**(Flow flow) or **ProxyBranch.setFlow**(Flow flow) methods, the SIP Servlet container uses the corresponding transport association to send the message.

Example 8–5 illustrates one such implementation.

Example 8–5 Looking up a Flow Object

```
@Invite
public void handleInvite(SipServletRequest request) {
  Address route = request.getPoppedRoute();
  String flowToken = route.getURI().getParameter("flow");
  Proxy proxy = request.getProxy();
  Flow flow = sipServletContext.getFlow(flowToken);
  proxy.setFlow(flow);
  proxy.proxyTo(request.getRequestURI());
}
```

Releasing a Flow

An application can signal to Converged Application Server that it is no longer interested in using Flow by invoking the **Flow.release**() method. Converged Application Server uses that signal while making the decision to stop the keep-alive and/or terminating the underlying transport association. Since more than one application may be using the same transport association, **Flow.release**() should not be considered equivalent to terminating the transport association.

9

Back to Back User Agents

This chapter describes how to manage back-to-back user agents (B2BUAs) in Oracle Communications Converged Application Server.

A B2BUA is a Session Initiation Protocol (SIP) element that acts as an endpoint for two or more dialogs and forwards requests and responses between those two dialogs in some fashion.

B2BUAs are sometimes considered undesirable because of their potential to break services. That potential stems from the fact that they sit between two endpoints and, in some way, mediate the signaling between those endpoints. If the B2BUA doesn't know about an end-to-end service being used between those two endpoints, it may inadvertently break it.

B2BUAs are, however, an important tool for SIP application developers and as such are supported by the SIP Servlet application program interface (API). Oracle Converged Application Server supports a new helper class and some new methods to facilitate implementing B2BUAs as defined by Java Specification Request (JSR) 359, https://jcp.org/en/jsr/detail?id=359.

About Back to Back User Agents

There is no Internet Engineering Task Force (IETF) standard that defines how a B2BUA behaves; however, it is largely assumed to be an entity that receives a request, upstream, as a User Agent Server (UAS) and relays it as a new request based on the received request downstream as a User Agent Client (UAC). Effectively, the application relays the request downstream.

From a SIP servlet perspective, an application is considered a B2BUA if it indicates the routing directive as CONTINUE.

Example 9–1 illustrates explicit linking.

Example 9–1 B2BUA Routing

```
        +-----------------------------------------+
        |                                         |
  req1  | req2 = factory.createRequest(appSession, |  req2
------->| "INVITE", req1.getFrom(), req1.getTo()); |--------->
        | // Copy headers and content from req1... |
        | req2.setRoutingDirective(CONTINUE, req1);|
        | req2.send();                            |
        +-----------------------------------------+
```

Navigating Between the UAC and UAS Sides of a B2BUA

A common behavior of a B2BUA is to forward the requests and responses received on the UAS side of the B2BUA to the UAC side and back again. Since both UAS and UAC sides of the B2BUA contain their own SipSession objects, while receiving a request or response on one side, an application often needs to navigate to a SipSession on the other side. To facilitate this, applications can store session IDs of the peer session as an attribute in the SipSession as shown in Example 9–2.

Example 9–2 Linking SIP Sessions

```
public void linkSessions(SipSession s1, SipSession s2) {
  s1.setAttribute(LINKED_SESSION_ID, s2.getId());
  s2.setAttribute(LINKED_SESSION_ID, s1.getId());
}
```

The linked sessions can then be retrieved by using the stored attributes as shown in Example 9–3.

Example 9–3 Retrieving Linked SIP Sessions

```
private SipSession retrieveOtherSession(SipSession s) {
  String otherSessionId = (String) s.getAttribute(LINKED_SESSION_ID);
  return s.getApplicationSession().getSipSession(otherSessionId);
}
```

Many times, the UAC and UAS sides of the B2BUA need to access the SipServletRequest on the other side. To facilitate this, applications may store the request IDs of the SipServletRequest in each other as shown in Example 9–4.

Example 9–4 Linking Requests

```
@Bye
protected void handleBye(SipServletRequest bye1) throws IOException {
  SipSession otherSession = retrieveOtherSession(bye1.getSession());
  SipServletRequest bye2 = otherSession.createRequest("BYE");
  linkRequests(bye1, bye2);
  bye2.send();
}
private void linkRequests(SipServletRequest r1, SipServletRequest r2) {
  r1.setAttribute(LINKED_REQUEST_ID, r2.getId());
  r2.setAttribute(LINKED_REQUEST_ID, r1.getId());
}
```

When needed, the requests can then be accessed from the session, if they are active as shown in Example 9–5.

Example 9–5 Retrieving Requests

```
@SuccessResponse @Bye
protected void byeResponse(SipServletResponse r1) throws IOException {
  SipServletRequest request = retriveLinkedRequest(r1.getRequest());
  SipServletResponse r2 = request.createResponse(r1.getStatus(),
    r1.getReasonPhrase());
  r2.send();
}
private SipServletRequest retriveLinkedRequest(SipServletRequest r1) {
  String requestId = (String) r1.getAttribute(LINKED_REQUEST_ID);
  SipSession session = retrieveOtherSession(r1.getSession());
  return session.getActiveRequest(requestId);
```

```
}
```

ACK and PRACK Handling in B2BUA

To forward an ACK request, the B2BUA needs to access the servlet on the other side and the application needs to access final response. The application can retrieve the final response of the INVITE request by using the **SipServletRequest.getFinalResponse**() method. When an ACK request arrives, B2BUA first retrieves the INVITE method from the other side by using the **SipSession.getActiveInvite**(*UAMode*) method and then accesses the final response from SipServletRequest as shown in Example 9–6.

Example 9–6 B2BUA Relaying ACK

```
@Ack
protected void handleAck(SipServletRequest uasAck) throws IOException {
  SipSession uacSession = retrieveOtherSession(uasAck.getSession());
  SipServletRequest uacInvite = uacSession.getActiveInvite(UAMode.UAC);
  SipServletRequest uacAck = uacInvite.getFinalResponse().createAck();
  uacAck.send();
}
```

For reliable provisional responses, B2BUA may establish a relationship between an incoming reliable provisional response with the one relayed by the B2BUA. This can be done by saving the reliable sequence number (RSeq) and the request ID as attributes in the response as shown in Example 9–7.

Example 9–7 Linking Reliable Provisional Responses

```
@ProvisionalResponse
void provisionalResponse(SipServletResponse r1) throws Exception {
  SipServletRequest request = retriveLinkedRequest(r1.getRequest());
  SipServletResponse r2 = request.createResponse(r1.getStatus());
  if (r1.isReliableProvisional()) {
    String respId = r1.getProvisionalResponseId();
    r2.setAttribute(LINKED_RELIABLE_RESPONSE_ID, respId);
    r2.sendReliably();
  }
}
```

The linked provisional response may then be retrieved when the B2BUA forwards a PRACK as shown in Example 9–8.

Example 9–8 Forwarding a PRACK

```
@Prack
void handlePrack(SipServletRequest prack) throws Exception {
  SipServletResponse r1 = prack.getAcknowledgedResponse();
  SipSession session = retrieveOtherSession(prack.getSession());
  String respId = (String) r1.getAttribute(LINKED_RELIABLE_RESPONSE_ID);
  SipServletResponse r2 =
  session.getUnacknowledgedProvisionalResponse(respId);
  r2.createPrack().send();
}
```

B2BUA and Forking

When an INVITE request is forked downstream, one request may receive responses on different dialogs (derived sessions). Each response causes a transaction branch and is represented in the SipServlet API by the InviteBranch interface.

A B2BUA forwards such responses on different branches as a UAS to create a new InviteBranch using the **SipServletRequest.createInviteBranch**() method as shown in Example 9–9.

Example 9–9 Sending a Response on a Particular Branch

```
@Invite
void handleInviteResponse(SipServletResponse r1) throws IOException {
  InviteBranch branch = r1.getSession().getActiveInviteBranch();
  if (branch != null) {
    SipSession uasSession = retrieveOtherSession(r1.getSession());
    if (uasSession == null) {
      SipServletRequest req =
      retreiveLinkedRequest(r1.getRequest());
      branch = req.createInviteBranch();
    }
    SipServletResponse r2 =
    branch.createResponse(r1.getStatus(), r1.getReasonPhrase());
    r2.send();
  }
}
```

One SipServletRequest may result in more than one branch. Therefore, when there are circumstances where a B2BUA needs to access a particular branch, the B2BUA must use the corresponding InviteBranch object. For example, while relaying an ACK message, the B2BUA may access the final response from the InviteBranch object as shown in Example 9–10.

Example 9–10 Relaying an ACK on a Particular Branch

```
@Ack
protected void handleAck(SipServletRequest uasAck) throws IOException {
  SipSession uacSession = retrieveOtherSession(uasAck.getSession());
  InviteBranch branch = uacSession.getActiveInviteBranch();
  if (branch != null) {
    SipServletResponse response = branch.getFinalResponse();
    response.createAck().send();
  }
}
```

Although non-INVITE requests can also be forked, the absence of ACK and PRACK messages make their call flow much simpler. Similarly, non-INVITE requests do not support sending responses on more than one dialog from the UAS. Thus, the SIP Servlet API does not provide an API equivalent to the InviteBranch for non-INVITE requests.

The B2BUA Helper Class

The B2BUA helper class contains useful methods for simple B2BUA operations, and can be retrieved from the SipServletRequest by invoking the **getB2buaHelper**() method on it:

```
B2buaHelper getB2buaHelper() throws IllegalStateException;
```

Invoking the **getB2buaHelper**() method also indicates to Converged Application Server that this application wishes to be a B2BUA.

Any UA operation is permitted by the application but the application cannot act as proxy after that, so any invocation to **getProxy**() must then throw a IllegalStateException.

Similarly, the **getB2buaHelper**() method throws an IllegalStateException if the application has already retrieved a proxy handle by an earlier invocation of **getProxy**().

Note: B2BUA helper class functionality can only be used for linking two legs (inbound and outbound) and does not support all cases of forking at the B2B or downstream. A B2BUA application may come across complex scenarios and call flows that can not be implemented by using the B2BUA helper class.

Creating a New B2BUA Request

The behavior described in this section minimizes the risk of breaking end-to-end services by copying all unknown headers from the incoming request to the outgoing request.

When an application receives an initial request for which it wishes to act as a B2BUA, it may invoke the **createRequest**() method available in the B2buaHelper. This method creates a request identical to the one provided as the first argument according to the following rules:

- All unknown headers and *Route*, *From*, and *To* headers are copied from the original request to the new request. The container assigns a new *From* tag to the new request.
- *Record-Route* and *Via* header fields are not copied. The container adds its own *Via* header field to the request when it is actually sent outside the application server.
- The headers in the new request can added to the optional headerMap, a Map of headers used in place of the ones from origRequest, for example:

  ```
  {"From" => {sip:myname@myhost.com},
  "To" => {sip:yourname@yourhost.com} }
  ```

  The only headers that can be set using this headerMap are non-system headers as well as *From*, *To*, and *Route* headers. For *Contact* headers, only the user part and some parameters are to be used as defined in 5.1.3 The Contact Header Field in JSR-359, https://jcp.org/en/jsr/detail?id=359. The values in the map are defined in a java.util.Set to account for multi-valued headers.

  The values in headerMap MUST override the values in the request derived from the origRequest. Specifically, they do not append header values. Attempts to set any other system header results in an IllegalArgumentException.
- The *linked* boolean flag indicates whether the ensuing SipSession and SipServletRequest are to be linked to the original ones. The concept of linking is described in "Linked SIP Sessions and Linked Request".
- For non-REGISTER requests, the Contact header field is not copied but is populated by Converged Application Server as usual.

Like other **createRequest**() methods, the returned request belongs to a new SipSession.

Note: The **SipFactory.createRequest**(SipServletRequest *origRequest*, boolean *sameCallId*) method has been deprecated in release 2.0 as the usage of this method with *sameCallId* flag set to *true* actually breaks the provisions of RFC 3261, https://www.ietf.org/rfc/rfc3261.txt, where the *Call-ID* value must be unique across dialogs. The use of **B2buaHelper.createRequest**(SipServletRequest *origRequest*) is recommended.

Linked SIP Sessions and Linked Request

This section describes using linked SIP sessions and requests in the context of B2BUAs.

Explicit Session Linkage

A B2BUA usually contains two SipSessions (although there can be more than two). The most common function of a B2BUA is to forward requests and responses from one SipSession to the other, after performing some transformation, usually an application of business logic. The B2buaHelper class simplifies the usage of this pattern by optionally linking the two SipSessions and SipServletRequests when you use it to create the new request:

```
SipServletRequest B2buaHelper.createRequest(SipServletRequest origRequest, boolean
linked, java.util.Map<java.lang.String, java.util.Set> headerMap)
```

The effect of this method when the linked parameter is *true* is to create a new SipServletRequest using the original request, such that the two SipSessions and the two SipServletRequests are linked together. When the two SipSessions and requests are linked, you can to navigate from one to the other.

Example 9–11 shows how you can access a linked session.

Example 9–11 Accessing a Linked Session

```
doSuccessResponse(SipServletResponse response) {
  ....
  otherSession = B2buaHelper.getLinkedSession(response.getSession());
  // Do something on otherSession
  ....
}
```

The **getLinkedSession**() is defined in the B2buaHelper class. The helper class works like a Visitor to the SipSession (and other classes) and encapsulates functionality useful to a B2BUA implementation.

Similar to SipSessions, the linked SipServletRequest can be obtained from the method:

```
B2buaHelper.getLinkedSipServletRequest(SipServletRequest)
```

Besides the **B2buaHelper.createRequest**() method, the linking can also be explicitly achieved by calling:

```
B2buaHelper.linkSipSessions(session1, session2) throws IllegalArgumentException;
```

An IllegalArgumentException is thrown when sessions cannot be linked together, such as when one or both sessions have terminated, or belong to different SipApplicationSessions, or one or both have been linked to different SipSessions.

The following helper method unlinks other sessions that are linked with a session:

```
B2buaHelper.unLinkSipSessions(session) throws IllegalArgumentException;
```

Only one SipSession can be linked to another single SipSession belonging to the same SipApplicationSession.

The linkage at the SipServletRequest level is implicit whenever a new request is created based on the original with link argument as true. There is no explicit linking or unlinking of SipServletRequests.

Implicit Session Linkage

Another useful method on B2buaHelper for subsequent requests is:

```
B2buaHelper.createRequest(SipSession session, SipServletRequest origRequest,
java.util.Map<java.lang.String, java.util.Set> headerMap) throws
IllegalArgumentException
```

The *session* is the SipSession on which this subsequent request is to be sent. The *origRequest* is the request received on another SipSession on which this request is to be created. The *headerMap* can contain any non-system header which needs to be overridden in the resulting request. Any attempt to set a system header results in an IllegalArgumentException. A call to the **createRequest**() method also automatically links the two SipSessions, if they are not already linked, as well as the two SipServletRequests.

Access to Uncommitted Messages

The method **SipServletMessage.isComitted**(), defines the committed semantics for a message:

```
public boolean isCommitted();
```

SipServletRequest and SipServletResponse objects always implicitly belong to a SIP transaction. The transaction state machine, as defined by JSR-359, constrains the messages that can legally be sent at various points of processing. If a servlet attempts to send a message that violates the transaction state machine, the container throws an IllegalStateException.

A SipServletMessage is committed when one of the following conditions is true:

- The message is an incoming request for which a final response has been generated.
- The message is an outgoing request that was sent.
- The message is an incoming non-reliable provisional response received by a servlet acting as a UAC.
- The message is an incoming reliable provisional response for which a PRACK was already generated.

> **Note:** This scenario applies to containers that support the 100rel extension.

- The message is an incoming final response received by a servlet acting as a UAC for a Non-INVITE transaction.
- The message is a response that has been forwarded upstream.
- The message is an incoming final response to an INVITE transaction and an ACK was generated.

- The message is an outgoing request, the client transaction has timed out, and no response was received from the UAS and the container generates a 408 response locally.

The semantics of the committed message is that it cannot be further modified or sent in any way.

The B2BUA Helper class method, **B2buaHelper.getPendingMessages**() gives the application a list of uncommitted messages in the order of increasing CSeq based on the UA mode, because there may be more than one request/response uncommitted on a SipSession:

```
List<SipServletMessage> B2buaHelper.getPendingMessages(SipSession, UAMode);
```

The UAMode is an ENUM with values UAC or UAS. The same session can act as a UAC or a UAS, and the *UAMode* indicates messages pertaining the particular mode.

For example, consider a B2BUA involved in a typical INV-200-ACK scenario that receives an ACK on one leg and wishes to forward it to the other. The B2BUA could call **B2buaHelper.getPendingMessages**(*leg2Session*, *UAMode.UAC*) to retrieve the pending messages which contain the original 200 response received on the second leg. The B2BUA can then create the ACK using the **SipServletResponse.createAck**() method. A PRACK request can be created in a similar way from a reliable 1xx response.

Original Request and Session Cloning

The incoming request that results in the creation of a SipSession is called the original request. The application can create a response to the original request even if that request was committed and the application does not have a reference to it. That is necessary because the B2BUA application may need to send more than one successful response to a request, for example, in the case when a downstream proxy is forked and more than one success response must be forwarded upstream. A response to the original request can be made using the **createResponseToOriginalRequest**() method:

```
SipServletResponse B2buaHelper.createResponseToOriginalRequest(
                       SipSession session,
                       int status,
                       String reasonPhrase) throws IllegalStateException;
```

This only works on initial requests, since only original requests require multiple responses.

The generated response must have a different *To* tag from the other responses generated to the request and must result in a different SipSession. In this and similar cases, Converged Application Server clones the original SipSession for the second and subsequent dialogs, as defined in 8.2.3.2 Derived SipSessions of JSR-359, https://jcp.org/en/jsr/detail?id=359. The cloned session object contains the same application data but its **createRequest**() method creates requests belonging to that second or subsequent dialog, that is, with a *To* tag specific to that dialog.

Request and Session Cloning and Linking

In the case above when more than one response is received on the UAC side, it results in a cloned UAC SipSession. When the response is sent on the UAS side using the original request, it is in context of the cloned UAS SipSession. Those SipSessions are pair-wise linked for easy navigation.

For example, if UAS-1 is the SipSession on which the incoming request was received and UAC-1 is the SipSession on which the outgoing request was relayed, then in the

case of multiple 2xx responses, one response is processed by the UAC-1 SipSession. When another 2xx response is received, Converged Application Server clones the UAC-1 SipSession to create a UAC-2 SipSession to process this new response. The B2buaHelper has the **getLinkedSession**() method to navigate from UAS-1 to UAC-1 and back again. As for the cloned SipSession UAC-2, the B2buaHelper furnishes the UAS-2 (a clone of UAS-1) when the **getLinkedSession**() method is invoked with UAC-2. The applications can then create the response to the original request but now in the context of UAS-2 by invoking the method:

```
SipServletResponse B2buaHelper.createResponseToOriginalRequest(
            SipSession session-uas-2,
            int status,
            String reasonPhrase) throws IllegalStateException;
```

10

Forking SIP Requests, Dialog Termination, and Session Keep Alive

This chapter describes methods for forking SIP requests, SIP dialog termination and handling session keep alive.

Forking SIP Requests

The forking of SIP requests means that multiple dialogs can be established from a single request. When ever multiple dialogs are created due to forking a derived session is created from the original session. Such derived sessions that effectively represent such sibling dialogs belong to a forking context. The ForkingContext interface helps an application to navigate to all such derived sessions. Thus a SipSession and its derived siblings belong to the same forking context.

A ForkingContext is created whenever the first session on a new dialog is created and it remains valid as long as at least one SipSession that belongs to the forking context remains valid. Applications can obtain an instance of a ForkingContext using the **SipSession.getForkingContext**() method.

When acting as a User Agent (UA), it is possible that an application might want to send a new request on the same forking context, but on a different dialog. In that case such a request establishes a new dialog. An example of such a request is the NOTIFY request as specified in RFC 6665, https://tools.ietf.org/html/rfc6665. SIP servlet applications can create new requests with the same forking Context using the method **ForkingContext.createRequest**(String *method*).

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Binding Attributes to a ForkingContext

A servlet can bind an object attribute to a ForkingContext by name. Hence ForkingContext is also an attribute store. See the discussion of Attribute Stores in Chapter 7 of JSR-359, https://jcp.org/en/jsr/detail?id=359 for more details.

Creating a Request

When **ForkingContext.createRequest**() is called, Converged Application Server derives a new session and creates a request using that session. Only NOTIFY requests can be created since this API is only applicable in that scenario. When an application tries to create any other requests, an IllegalArgumentException is thrown. The new derived session is created based upon the first session setup in the forking context.

Cloning Attributes

A ForkingContext supports the flag, **enableDerivedSessionAttributeCloning**(), for enabling or disabling the cloning of attributes when creating a derived session within the forking context scope. In legacy implementations, when creating a derived session, the attributes in the original session are retrieved and assigned to the new derived session. The new derived session refers to the same attributes object of the original session. When **enableDerivedSessionAttributeCloning**() is set to *true*, the attribute values of the original session are cloned, and the derived session refers to a new attributes object. The default value for **enableDerivedSessionAttributeCloning**() is *false*.

Terminating Dialogs

When the **ForkingContext.terminateDialogs**() method is called, Converged Application Server will terminate all SIP sessions of the forking context if they are not explicitly excluded. This method is available for both UA and proxy modes. When **ForkingContext.terminateDialogs**() is called in proxy mode, the feature *proxy send bye* will be added to the terminate proxy dialogs.

Max-Breadth Header Support

The *Max-Breadth* header limits the number of parallel forks that can be made on a Session Initiation Protocol (SIP) request by downstream proxies. Following RFC 5393, https://tools.ietf.org/html/rfc5393, each downstream proxy distributes the *Max-Breadth* among the active parallel branches so that the outgoing *Max-Breadth* is the sum of the *Max-Breadth* header field values in all forwarded requests in the response contexts that have not received a final response.

In Converged Application Server, the application decides the number of times the request is forked in parallel. An application may explicitly set the breadth on the forked requests by using the **SipServletRequest.setMaxBreadth()** method. Applications may retrieve the forked request by using the **ProxyBranch.getRequest()** method.

If the received request contains no Max-Breadth header, the container uses **60** as the default value.

In Converged Application Server, the SIP container is configured to automatically check the *Max-Breadth* header. The associated configuration parameter, **Max-Breadth Check Support**, is accessible in the Administration Console. The check box entry for **Max-Breadth Check Support** is selected, by default. You can disable the automatic checking of the *Max-Breadth* header. To do so, access the **General** configuration tab for **SIP Server** in the **Domain Structure** for the Administration Console and clear the **Max-Breadth Check Support** check box. See "About Accessing the Administration Console" in *Converged Application Server Administrator's Guide*.

If a Proxy application does not set the breadth explicitly, Converged Application Server distributes the available breadth evenly to the branches when the application invokes the **Proxy.proxyTo**() or **Proxy.startProxy**() methods.

At the time of forking, if the sum of the *Max-Breadth* values of active parallel branches exceeds the *Max-Breadth* of the original incoming request, Converged Application Server throws an InsufficientBreadthException. For a proxy, this check is done when the application invokes the **Proxy.proxyTo**() and **Proxy.startProxy**() methods. Applications may catch this exception and send an error response (440) or attempt to proxy again after adjusting the breadth.

Max-Breadth is used within Converged Application Server for loop detection.

Note: For details on the **SipServletRequest.setMaxBreadth**() method, see the *Java SIP Servlet API 2.0* JavaDocs.

Loop Detection

There is a possibility that loops may occur in invoked applications, such as when **A** proxies to **B** which proxies back to **A**. It is important that such loops be detected and handled. Converged Application Server decrements the value of the *Max-Forwards* header whenever a request is proxied internally, or whenever a request is forwarded by a servlet acting as a Back to Back User Agent (B2BUA).

Converged Application Server also checks the *Max-Breadth* header to verify whether the request can be proxied in parallel or forwarded to multiple destinations by a servlet acting as a B2BUA in parallel. Converged Application Server generates an InsufficentBreadthException when the *Max-Breadth* check fails.

SIP Dialog Termination

An application may terminate a Session Initiation Protocol (SIP) dialog using the SipSession.**terminateDialog** method at any time. If the application is acting as a SIP user agent on the SipSession, and if the SipSession corresponds to an established SIP dialog, or a dialog-establishing transaction is pending, **terminateDialog** method call instructs the container to send the appropriate SIP messages to terminate the dialog, otherwise the call results in a no-op.

Note: SIP dialogs are only created by the **INVITE**, **REFER**, and **SUBSCRIBE** methods, and, therefore, **terminateDialog** only affects dialogs created by these methods. For any other session, the method call will result in an IllegalStateException.

Once the terminateDialog method is called, the application cannot send any more messages in the SipSession. Any attempt to do so will cause IllegalStateException to be thrown by the **SipServletMessage.send**() method. Furthermore, the container will not invoke the application's SipServlet **service**() or **doXXX**() methods from that point on. When the dialog is terminated, the app is notified by the current SipSessionListener.**sessionReadyToInvalidate**() method.

An application may terminate all dialogs belong to a forking context using ForkingContext.**terminateDialogs**() method. The container will also terminate all derived sessions that belong to the same ForkingContext, created after executing this method. For more information, see "Forking SIP Requests".

Note: This method will not terminate dialogs that belong to a proxy application.

In certain cases, an application may wish to modify the outgoing SIP message that the container is sending in order to terminate a dialog. For example, the application may want to add a Reason header to a BYE method, or a message content to a NOTIFY method. There are also cases where the application may wish to be notified of incoming SIP messages. The application may provide a listener using the following method:

```
SipSession.terminateDialog(AutomaticProcessingListener listener)
```

The AutomaticProcessingListener should not make changes to message that subvert the RFCs and container behavior in terminating the dialog. Oracle also recommends that applications do not throw any exceptions during the execution of AutomaticProcessingListener. Any exception thrown by the application will be ignored by the SIP servlet container.

Table 10–1 describes the AutomaticProcessingListener methods.

Table 10–1 AutomaticProcessingListener Methods

Method Name	Description
outgoingRequest(*request*)	This method is invoked before Converged Application Server sends a SIP request. The application may modify the request in-place.
outgoingResponse(*response*)	This method is invoked before the Converged Application Server sends a SIP response. The application may modify the response in-place.
incomingRequest(*request*)	This method is invoked when the Converged Application Server receives a SIP request.
incomingResponse(*response*)	This method is invoked when the Converged Application Server receives a SIP response.

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Terminating Proxy Dialogs

As defined in RFC 3261, https://www.ietf.org/rfc/rfc3261.txt, a SIP proxy must not create and send requests in an established dialog. However, some 3GPP applications need this ability (see 3GPP TS 24.229, http://www.3gpp.org/dynareport/24229.htm, section 5.2.8.1.2 Release of an existing session). To support that use case, a separate method is provided so that an application can terminate proxy dialogs.

Note: Given the way that this method breaks RFC 3261, this method should be used with caution and any application that uses it should be carefully tested. 3GPP specifications also require a way where the termination messages are sent for either one side of the proxy or both.

If the SipSession corresponds to an established SIP dialog, or a dialog-establishing transaction is pending, the methods in Table 10–2 instruct the container to send the appropriate SIP messages to terminate the dialog.

Table 10–2 Proxy Dialog Termination Methods

Method Name	Description
terminateProxiedDialog(direction)	Terminates the proxied dialog by sending appropriate messages in the direction specified. For example, if the direction is UAC, Converged Application Server will send BYE message to terminate an INVITE dialog in the direction of UAC.

Table 10–2 (Cont.) Proxy Dialog Termination Methods

Method Name	Description
terminateProxiedDialog(direction, listener)	Terminates the proxied dialog by sending appropriate messages in the direction specified. Applications may intercept messages during termination.
terminateProxiedDialog()	Terminates the proxied dialog by sending appropriate messages in both directions.
terminateProxiedDialog(listener)	Terminates the proxied dialog by sending appropriate messages in both the directions. Applications may intercept messages during termination.

Note: An application should invoke the first two **terminateProxiedDialog** methods *only* when it find using some other means that other side of the specified direction will not respond. Any message on the other side may be ignored by Converged Application Server.

Notes on Container Behavior

Converged Application Server uses the following RFCs as guidelines for correct dialog termination behavior:

- RFC 3261, https://www.ietf.org/rfc/rfc3261.txt
- RFC 6665 (for SUBSCRIBE), https://tools.ietf.org/html/rfc6665
- RFC 3515 (for REFER), https://www.ietf.org/rfc/rfc3515.txt
- RFC 5057 (multiple dialogs), https://tools.ietf.org/html/rfc5057
- RFC 5407 (race states), https://tools.ietf.org/html/rfc5407
- RFC 6026 (an update to RFC 3261), https://tools.ietf.org/html/rfc6026

The following sections provide examples of Converged Application Server behavior in terminating SIP dialogs based on those RFCs. In the examples, the terms UA, UAC and UAS should be understood to mean the Converged Application Server acting on behalf of an UA application that has called SipSession.**terminateDialog**(). Similarly, when a proxy dialog is being terminated using SipSession.**terminateProxiedDialog**(), these terms corresponds to the direction(s) in which the messages are being sent.

INVITE Dialog

Keep in mind the following INVITE dialog points:

- When a dialog is in its early state, a caller UA must send CANCEL to terminate the dialog. (While RFC3261 allows sending CANCEL or BYE, RFC 5407 section 2 says UAC MAY send BYE, but not recommended, so Converged Application Server sends a CANCEL request.). The callee UA can send an error as a final response.
- The callee's UA must not send a BYE on a confirmed dialog until it has received an ACK for its 2xx response or until the server transaction times out. (See RFC3261 sec 15, and RFC6026).
- When a UA receives a BYE, it must respond to any pending requests received for that dialog with a 487 response (RFC3261 sec 15.1.2).

- While not specified explicitly, a UA should respond to any pending requests before sending BYE. Converged Application Server sends a 487 response code when responding to pending requests.
- Various race conditions as described by RFC5407, for example when a requests have arrived after a UA has sent BYE, are responded to.

SUBSCRIBE Dialog

Keep in mind the following SUBSCRIBE dialog points:

- A SUBSCRIBE dialog can be created explicitly by subscriber sending SUBSCRIBE.
- A SUBSCRIBE dialog can also be created implicitly by REFER.
- A subscription is destroyed when a UA acting as notifier sends a NOTIFY request with a "Subscription-State" of "terminated". If the dialog is not otherwise in use, the SIP dialog is terminated.
- A UA acting as a subscriber cannot explicitly terminate a subscription. A UA acting as subscriber may send a SUBSCRIBE request with an "Expires" header of 0 in order to trigger the notifier to send a NOTIFY request that destroys the subscription.

Multiple Dialogs

Keep in mind the following points when using multiple dialogs:

- It is possible for a dialog to have multiple usages (RFC 5057). For the purpose of dialog termination, in order to terminate the dialog the UA must terminate each of those usages independently in order to terminate the entire dialog.
- The order in which messages are sent to terminate the usages is not important. For example, the UA may need to send BYE and NOTIFY to terminate the INVITE and SUBSCRIBE usages in a dialog respectively, and the UA may send these messages in either order.

Session Keep Alive

Session Initiation Protocol (SIP) user agents (UAs) and proxies depend on session terminating messages to end the session and clean up resources. When a terminating message does not arrive, Converged Application Server supports a mechanism to determine whether the session should be kept alive or not as explained in this section. For instance, when a UA fails to send a BYE message at the end of a session, or when the BYE message is lost due to network problems, a Call Stateful Proxy (CSP) needs to know when the session has ended.

RFC 4028, https://tools.ietf.org/html/rfc4028, defines an extension that defines a keep alive mechanism for SIP sessions. UAs send periodic re-INVITE or UPDATE requests to keep the session alive. The interval for the session refresh requests is determined through a negotiation mechanism defined in RFC 4028. If a session refresh request is not received before the interval passes, the session is considered terminated. Both UAs are supposed to send a BYE, and CSPs can remove any state for the call. Converged Application Server supports this keep alive mechanism as defined in this section.

Enabling Session Keep Alive

A SIP servlet can enable the session keep alive by setting appropriate keep alive preference to generate an initial session refresh request, and can retrieve a

SessionKeepAlive.Preference object using the method SipServletMessage.**getSessionKeepAlivePreference**(). The servlet can then enable the keep alive by invoking SessionKeepAlive.**Preference**.**setEnabled**(*true*). Converged Application Server sets the headers (*Session-Expires*, *Supported*, *Min-SE*, and others) as specified in RFC 4028 by using the values specified in SessionKeepAlive.**Preference** if the session keep alive is enabled.

If a User Agent Client (UAC) wants the session timer to be applied to the session, the UAC is required to enable the session keep alive by invoking SessionKeepAlive.**Preference**.**setEnabled**(*true*)before sending the initial session refresh request. Converged Application Server sets *Session-Expires* to a default value of 1800 seconds once the keep alive is enabled, if it has not been set previously.

For proxies and User Agent Servers (UAS), session keep alive is enabled, if the initial session refresh request contains a *Session-Expires* header. If there is no *Session-Expires* header in the request, then the UAS or proxy may apply a session timer to the session by enabling the session keep alive.

Once session keep alive is enabled, Converged Application Server follows the procedures specified in RFC 4028 for UAC, Proxy, and UAS for activating session timer for the session.

Note: For details on the API described in this section, see the *Java SIP Servlet API 2.0* JavaDocs.

Disabling Session Keep Alive

The proxy and UAS may receive a session refresh request with a *Session-Expires* header. In that case, by default, session keep alive is enabled. If a proxy or UAS does not want to take part in the session keep alive activity, then it may choose to disable the session keep alive by invoking **SessionKeepAlive**.**Preference**.**setEnabled**(*false*).

Note: Disabling session keep alive does not mean that a UAC will not send session refresh requests any more. It may continue to send session refresh requests if the session keep alive remains enabled.

Refreshing Sessions

When the UA that enabled session keep alive assumes the role of refresher, Converged Application Server schedules a keep alive timer. However Converged Application Server will not send a refresh request on its own. A SIP servlet can provide a refresh callback that will be executed by Converged Application Server at the appropriate time for sending the session refresh request. The refresh callback needs to implement the **SessionKeepAlive**.**Callback** interface.

Example 10–1 illustrates such a callback.

Example 10–1 Refreshing a Session

```
request.getSessionKeepAlivePreference().setEnabled(true);
SessionKeepAlive skl = request.getSession().getKeepAlive();
skl.setRefreshCallback(new SessionKeepAlive.Callback() {
  @Override
  public void handle(SipSession session) {
    try {
      session.createRequest("UPDATE").send();
    } catch (IOException e) {
```

```
    }
  }
});
```

Converged Application Server invokes the refresh callback once half of the session expiration interval has elapsed. If the application sends a refresh request on its own by that time, Converged Application Server will re-calculate the next session expiration time and invoke the refresh callback accordingly.

An application can remove the refresh callback by setting a null refresh task by invoking **SessionKeepAlive.setRefreshCallback**(*null*);

Expiring Sessions

When a Proxy or UA does not receive a session refresh request before the expiration interval, Converged Application Server invokes the **expiry** callback. The expiry callback needs to implement SessionKeepAlive.Callback interface as shown in Example 10–2.

Example 10–2 Expiring a Session

```
request.getSessionKeepAlivePreference().setEnabled(true);
SessionKeepAlive skl = request.getSession().getKeepAlive();
skl.setExpiryCallback(new SessionKeepAlive.Callback() {
  @Override
  public void handle(SipSession session) {
    try {
      session.createRequest("BYE").send();
    } catch (IOException e) {
    }
  }
});
```

Sending Provisional Responses to Non-Invite Requests

In Converged Application Server, if the application or proxy does not respond to a non-invite request before the TimerE reaches T2, the container responds with a 100 TRYING to the request. It does so, if the container has not otherwise responded after the amount of time it takes a client transaction's TimerE to be reset to T2. For more information about TimeE and SIP Non-Invite actions, please refer to RFC 4320 at

http://tools.ietf.org/html/rfc4320

The associated configuration parameter, **Enable Sending 100 For Non-Invite Request Support**, is accessible in the Administration Console. The check box entry for **Enable Sending 100 For Non-Invite Request Support** is selected, by default. You can disable the container from responding with a 100 TRYING to such non-invite requests. To do so, access the **General** configuration tab for **SIP Server** in the **Domain Structure** for the Administration Console and clear the **Enable Sending 100 For Non-Invite Request Support** check box. See "About Accessing the Administration Console" in *Converged Application Server Administrator's Guide*.

422 Responses

UAS and Proxy applications are expected to generate a 422 response if they find that the session expiration interval is too small. Similarly, UACs are expected to handle the 422 response and retry the request as specified in RFC 4028.

11

Using Compact and Long Header Formats for SIP Messages

This chapter describes how to use the Oracle Communications Converged Application Server `SipServletMessage` interface and configuration parameters to control Session Initiation Protocol (SIP) message header formats.

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. Java Specification Request (JSR) 359 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` application program interface (API) provided in JSR 359 is the ability to set long or compact header formats for the entire SIP message using the `setHeaderForm` method.

In addition to `SipServletMessage`, Converged Application 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.

Summary of Compact Headers

Table 11–1 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 11–1 Compact Header Abbreviations

Header Name (Long Format)	Compact Format
Call-ID	i
Contact	m
Content-Encoding	e
Content-Length	l
Content-Type	c
From	f
Subject	s

Table 11–1 (Cont.) Compact Header Abbreviations

Header Name (Long Format)	Compact Format
Supported	k
To	t
Via	v

Summary of API and Configuration Behavior

Header formats can be specified at the header, message, and SIP Servlet container levels. Table 11–2 shows the header format that results when adding a new header with `SipServletMessage.setHeader`, given different container configurations and message-level settings with `WlssSipServletResponse.setUseHeaderForm`.

Table 11–2 API Behavior when Adding Headers

SIP Servlet Container Header Configuration (use-compact-form Setting)	WlssSipServletMessage.setUseHeaderForm 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”
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 11–3 shows the system header format that results when setting the header format with `WlssSipServletResponse.setUseHeaderForm` given different container configuration values.

Table 11–3 API Behavior for System Headers

SIP Servlet Container Header Configuration (use-compact-form Setting)	WlssSipServletMessage. setUseHeaderForm 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”

12

Developing Custom Profile Service Providers

This chapter describes how to use the Profile Service API to develop custom profile providers in Oracle Communications Converged Application Server.

Overview of the Profile Service API

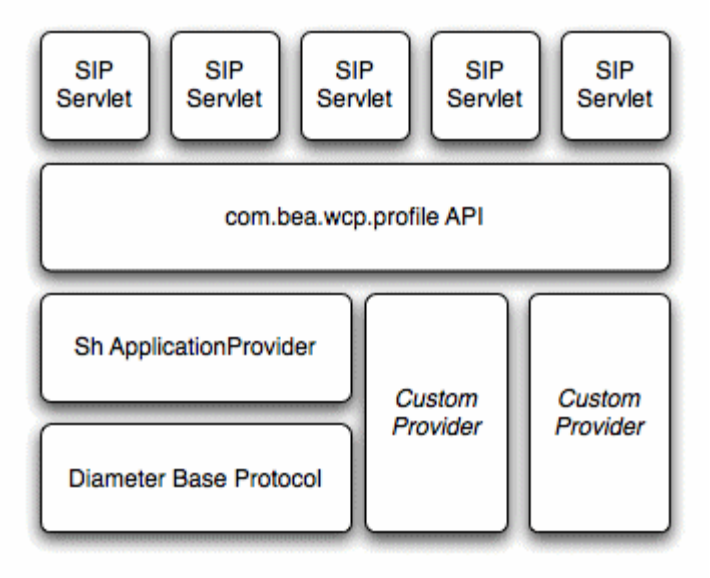
Converged Application Server includes a profile service application program interface (API), `com.bea.wcp.profile.API`, that may have multiple profile service provider implementations can be used to create profile provider implementations. A profile provider performs the work of accessing XML documents from a data repository using a defined protocol. Deployed Session Initiation Protocol (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 Converged Application 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 Document Object Model (DOM) format, so client code can work with many different types of profile data in a common way.

You can also use the profile service API to create a custom provider for retrieving document schemas using another protocol. For example, a profile provider could be created to retrieve subscription data from an Lightweight Directory Access Protocol (LDAP) store or a relational database management system (RDBMS).

Note: The Diameter Sh application also accesses profile data from a Home Subscriber Server using the Sh protocol. Profile. Although applications access this profile data using a simple URL, the Diameter applications are implemented using the Diameter base protocol implementation rather than the profile provider API.

Figure 12–1 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 Converged Application 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.

Implementing Profile Service API Methods

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

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

The Converged Application Server *Java 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.

Configuring and Packaging Profile Providers

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

See "Creating Shared Java EE Libraries and Optional Packages" in *Developing Applications for Oracle WebLogic Server* for information on how to assemble Java EE libraries. For most profile providers, you can simply package the implementation classes in a JAR file and then register the library with Converged Application 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 12–1 shows a sample configuration for a provider that accesses data using XCAP.

Example 12–1 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>
```

Mapping Profile Requests to Profile Providers

When an application makes a request using the Profile Service API, Converged Application Server must find a corresponding provider to process the request. By default, Converged Application 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 12–1, Converged Application 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 12–2 shows a mapping with two alternate prefixes.

Example 12–2 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 12–3 shows an example configuration.

Example 12–3 Using a Custom Mapping Class

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

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 12–4 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.

13

Using Content Indirection in SIP Servlets

This chapter describes how to develop Session Initiation Protocol (SIP) Servlets that work with indirect content specified in the SIP message body, and how to use the Oracle Communications Converged Application Server content indirection API.

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).

Converged Application Server provides a simple application program interface (API) that you can use to work with indirect content specified in SIP messages.

Using the Content Indirection API

The content indirection API provided by Converged Application 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.ICParsedData`.

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.

Additional Information

Complete details about content indirection are available in RFC 4483:

`http://www.ietf.org/rfc/rfc4483.txt`

See also the Converged Application Server *Java API Reference* for additional documentation about the content indirection API.

14

Securing SIP Servlet Resources

This chapter describes how to apply security constraints to Session Initiation Protocol (SIP) Servlet resources when deploying to Oracle Communications Converged Application Server.

Overview of SIP Servlet Security

The SIP Servlet application program interface (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.

For SIP servlet Plain Old Java Objects (POJOs), annotations available which provide identical functionality:

- @SipSecurity: specifies security constraints to be enforced on SIP protocol messages.
- @SipConstraint: used within the @SipSecurity annotation to represent the security constraint to be applied to all SIP protocol methods for which a corresponding @SipMethodConstraint does not occur within the @SipSecurity annotation.
- @SipMethodConstraint: used within the @SipSecurity annotation to represent security constraints on specific SIP protocol messages.

For information on using the @SipSecurity annotation see section 22.3.10.1 in JSR-359, `https://jcp.org/en/jsr/detail?id=359`.

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**.

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 documentation:

- The discussion on securing web applications in *Programming WebLogic Security* provides an overview of declarative and programmatic security models for Servlets.
- The discussion on EJB security-related deployment descriptors in "Securing Enterprise JavaBeans (EJBs)" in *Programming WebLogic Security* 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 14–1, "Declarative Security Constraints in sip.xml".

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.

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>
```

Converged Application Server 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. Converged Application 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. 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 the **sip.xml** and **weblogic.xml** deployment descriptors, 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 the **sip.xml** deployment descriptor's `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 Converged Application Server role assignment in more detail.

Using Implicit Role Assignment

With implicit role assignment, Converged Application Server assigns a `security-role` name in **sip.xml** to a role of the exact same name, which must be configured in the Converged Application 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**, Converged Application Server implicitly maps the **sip.xml** deployment descriptor's `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. Converged Application Server displays 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 Converged Application 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.

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.

Important Requirements

If you specify a `security-role-assignment` element in the **weblogic.xml** deployment descriptor, Converged Application 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**, Converged Application Server generates the error message:

`The security-role-assignment references an invlaid 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 14–1 shows a portion of a **sip.xml** deployment descriptor that defines a security constraint with the role, `roleadmin`. Example 14–2 shows that a `security-role-assignment` element has been defined in **weblogic.xml** to assign principals and roles to `roleadmin`. In Converged Application Server, this Servlet *must* be deployed with a **web.xml** deployment descriptor that also defines the `roleadmin` role, as shown in Example 14–3.

If the **web.xml** contents were not available, Converged Application 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 14–1 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 14–2 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 14–3 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>
```

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 Converged Application Server security realm. If the `security-role` is used in combination with the `run-as` element in **sip.xml**, Converged Application Server assigns the first principal or role name specified in the `security-role-assignment` to the `run-as` role.

Example 14–2, "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 14–1, "Declarative Security Constraints in sip.xml". To change the role assignment, you must edit the **weblogic.xml** descriptor and redeploy the SIP Servlet.

Dynamically Assigning Roles Using the Administration 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 on the **Security** control panel in 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 14–4 shows an example of using the `externally-defined` element with the `roleadmin` role defined in Example 14–1, "Declarative Security Constraints in sip.xml". To assign existing principals and roles to the `roleadmin` role, the Administrator would use the Converged Application Server Administration Console.

See “Users, Groups, and Security Roles” in *Securing Resources Using Roles and Policies for Oracle WebLogic Server* for information about adding and modifying security roles by using the Administration Console.

Example 14–4 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>
```

Assigning run-as Roles

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

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

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

Example 14–5 shows an example of a configured run-as role in a **sip.xml** descriptor.

Example 14–5 run-as Roles in sip.xml

```
...
  <servlet>
    <servlet-name>myservlet</servlet-name>
    <servlet-class>com.mycompany.MyServlet</servlet-class>
    <run-as>
      <role-name>weblogic</role-name>
    </run-as>
  </servlet>
...
```

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.

Role Assignment Precedence for SIP Servlet Roles

Converged Application 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:** Converged Application 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 the **sip.xml** deployment descriptor's `run-as` role in a `run-as-principal-name` element defined within `servlet-descriptor`, the `run-as-principal-name` assignment is used.

> **Note:** Converged Application 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 the **sip.xml** deployment descriptor's `run-as` role in a `run-as-role-assignment` element, the `run-as-role-assignment` element is used.

> **Note:** Converged Application 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 the **sip.xml** deployment descriptor's `run-as` role in a `security-role-assignment` element, the `security-role-assignment` is used.

> **Note:** Converged Application 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.

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 Converged Application Server. Using this debug option causes Converged Application Server to display additional security-related messages in the standard output.

weblogic.xml Deployment Descriptor Reference

The **weblogic.xml** DTD contains detailed information about each of the role mapping elements discussed in this section. See "weblogic.xml Deployment Descriptor Elements" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

15

Enabling Message Logging

This chapter describes how to use Oracle Communications Converged Application Server message logging features on a development system.

Overview

Message logging records Session Initiation Protocol (SIP) and Diameter messages (both requests and responses) received by Converged Application 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.

> **Note:** The message logging functionality logs all SIP requests and responses; do not enable this feature in a production system. In a production system, you can instead configure one or more logging Servlets, which enable you to specify additional criteria for determining which messages to log. See "Logging SIP Requests and Responses" in *Converged Application Server Administrator's Guide*.

When you enable message logging, Converged Application 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".

Enabling Message Logging

You enable and configure message logging by adding a `message-debug` element to the **sipserver.xml** configuration file. Converged Application 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 left pane of the Administration Console, select **Configuration**, then select the **Message Debug** tab of the SipServer console extension node.

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 15–1 shows a configuration entry that specifies the `full` logging level.

Example 15–1 Sample Message Logging Level Configuration in sipserver.xml

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

Customizing Log Records

Converged Application 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 `token`s to log in each record.

> **Note:** When `level` is set to `full`, `format` is overridden.

Table 15–1 describes the nested elements used in the `format` element.

Table 15–1 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 15–2 provides a list of available string tokens. You can define multiple `token` elements as needed to customize your log records.

Table 15–2 describes the string `token` values used to specify information in a message log record:

Table 15–2 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

Table 15–2 (Cont.) Available Tokens for Message Log Records

Token	Description	Example or Type
%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_port	The port number portion of the From header.	7002
%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_port	The port number portion of the To header.	7002
%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.

Specifying Content Types for Unencrypted Logging

By default Converged Application 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, Converged Application 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, Converged Application 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, Converged Application 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 15–2 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 15–2 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>
```

Example Message Log Configuration and Output

Example 15–3 shows a sample message log configuration in **sipserver.xml**. Example 15–4, "Sample Message Log Output" shows sample output from the Managed Server log file.

Example 15–3 Sample Message Log Configuration in sipserver.xml

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

Example 15–4 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
 >
```

Configuring Log File Rotation

Message log entries for SIP and Diameter messages are stored in the main Converged Application 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 Converged Application 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 15–3 describes each element. Note that a server restart is necessary in order to initiate independent logging and log rotation.

Table 15–3 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`, where *Domain_Home* is the domain's home directory.
`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.

Table 15–3 (Cont.) XML Elements for Configuring Log Rotation

Element	Description
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 15–5 shows a sample message-debug configuration using log rotation.

Example 15–5 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>
```

16

Generating SNMP Traps from Application Code

This chapter describes how to use the Oracle Communications Converged Application Server `SipServletSnmpTrapRuntimeMBean` to generate Simple Network Management Protocol (SNMP) traps from within a SIP Servlet.

See "Configuring SNMP" in the *Converged Application Server Administrator's Guide* for information about configuring SNMP in a Converged Application Server domain.

Overview

Converged Application Server includes a runtime MBean, `SipServletSnmpTrapRuntimeMBean`, that enables applications to easily generate SNMP traps. The Converged Application Server management information base (MIB) contains seven new object identifiers (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, as a single parameter, the String value of the trap message to generate.

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

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 Converged Application Server administrator `role-name` entry to the

`security-role` and `run-as` role elements in the **sip.xml** deployment descriptor. Example 16–1 shows a sample **sip.xml** file with the required role elements highlighted.

Example 16–1 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_2_0.dtd">
   <sip-app xmlns="http://xmlns.jcp.org/xml/ns/sipservlet"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/sipservlet
         http://xmlns.jcp.org/xml/ns/sipservlet/sip-app_2_0.xsd"
         version="2.0">
  <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>
```

Obtaining a Reference to SipServletSnmpTrapRuntimeMBean

Any SIP Servlet that generates SNMP traps must first obtain a reference to the `SipServletSnmpTrapRuntimeMBean`. Example 16–2 shows the sample code for a method to obtain the MBean.

Example 16–2 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;
}
```

Generating an SNMP Trap

In combination with the method shown in Example 16–2, Example 16–3 demonstrates how a SIP Servlet would use the MBean instance to generate an SNMP trap in response to a SIP INVITE.

Example 16–3 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());
  }
}
```

17

Using the Location Service RESTful Interface

This chapter describes the Oracle Communications Converged Application Server RESTful API, a service interface based on Representational State Transfer (REST) that creates, modifies, and deletes address-of-record (AOR) entries in the Location Service.

About the Location Service RESTful Interface

This chapter lists RESTful operations for the Location Service, including the parameters accepted and returned by each operation and examples of HTTP requests and responses.

These operations store, lookup, and clear address-of-record registrations in the Location Service. An address on record (AOR) is a Session Initiation Protocol (SIP) or SIPS URI that points to a domain with a location service that can map the URI to another URI where the user might be available. Typically, the location service is populated through registrations. An AOR is frequently thought of as the "public address" of the user.

For example, to create objects that represent the AOR, the client application sends the following request to the API:

POST / *context-root*/locationservice/registration/sip:alice@example.com

About REST

The Location Service API follows the style of a REST interface.

In a RESTful API, functions are distinguished by the combination of a particular URI and the HTTP method used to access it. In general, the URI identifies the resource on which to act, and the HTTP method identifies the type of action to perform.

The methods in the HTTP protocol used in the Location Service RESTful API - POST, GET, PUT, and DELETE - correspond to the programming operations commonly known as CRUD operations. CRUD, which stands for create, read, update, and delete, represent the common operations applicable in data-oriented APIs. The equivalent function calls in a traditional API may be similar to createUser(), getUser(), setUser(), and deleteUser(). In this case, the instance on which the function operates is typically identified through an input parameter.

About JSON Body Parameters

Operations that are performed by using the GET or DELETE HTTP methods do not require input values other than what is provided in the URL and headers of the client

request. That is, they do not require HTTP body content to be supplied in the invocation request.

However, Location Service RESTful API operations are performed by using the POST methods require additional input data. The API takes input parameters in the form of JSON (JavaScript Object Notation) data in the body of the request.

JSON is a data exchange format based on JavaScript that is commonly used to pass information between web clients and servers over HTTP. In the body of the request, JSON data appears as one or more name-value pairs.

About the Context Root

The context root is set when the application is deployed. By default Converged Application Server uses **proxyregistrarssr** as context root. You can change the context root by editing the **application.xml** in the file

Middleware_Home/**occas/applications/proxyregistrarear.ear**.

To learn more, see the chapter on configuring the Proxy Registrar in *Converged Application Server Administrator's Guide*.

Using Authentication and Authorization

The Location Service RESTful interface's security consists of authentication and authorization. Authentication supports HTTP Digest and X-3GPP-Asserted-Identity header. Authorization allows only the AOR owner to access and update their registration information.

To use HTTP Digest and X-3GPP-Asserted-Identity authentication you must configure Converged Application Server to handle these header types for authentication. To learn more, see the chapters on configuring Digest authentication and 3GPP HTTP authentication assertion providers in *Converged Application Server Security Guide*.

RESTful APIs for the Location Service

The RESTful Location Service APIs are:

- Store Registrations for Address-of-Record
- Lookup an Address-of-Record
- Clear All Address of Record Bindings

Store Registrations for Address-of-Record

Stores registrations for the AOR.

An **HTTP response 200** message is returned on success.

HTTP Method

POST

URI

`proxyregistrarssr/locationservice/registration/`*`uri`*

Where the URI is of the form: sip:*username@domain.com*

Request Header

```
Accept application/json, Content-Type application/json
```

Request Body

The `cseq`, `contactAddress`, and `callId` parameters are required and case-sensitive.

```
[
{
"cseq":7,
"callId":"a97d0d177949304c@enpoYWkwMS5hcGFjLmJlYS5jb20.",
"contactAddress":"<sip:alice@10.182.101.231:5060>;expires=3600",
"methodsParam":"INVITE,BYE,CANCEL,ACK"
}
]
```

Response Body

```
[{
"aor":"sip:alice@example.com",
"contactUri":"sip:alice@10.182.101.231:5060",
"contactAddress":"<sip:alice@10.182.101.231:5060>;expires=3600",
"callId":"a97d0d177949304c@enpoYWkwMS5hcGFjLmJlYS5jb20.",
"cseq":7,
"qvalue":1.0,
"methodsParam":"INVITE,BYE,CANCEL,ACK",
"expiresParam":3600,
"expires":1335173253469,
"path":null,
"sipInstanceId":null,
"regId":null,
"remoteIP":"localhost",
"remotePort":2169,
"created":1335169653469,
"updated":1335169653469
}]
```

Example

Example 17–1 stores an AOR registration in the Proxy Registrar using the following parameters:

```
final String DEST_URL =
"/proxyregistrarssr/locationservice/registration/sip:alice@example.com";
private String account_name = "alice";
private String account_pass = "welcome1";
```

Example 17–1 Storing AOR Registrations

```
public void StoreRegistration() throws Exception {
String restUrl = "http://127.0.0.1:7001" + DEST_URL;
URL url = new URL(restUrl);
HttpURLConnection httpConn = (HttpURLConnection)url.openConnection();
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("POST");
httpConn.connect();
OutputStreamWriter outWriter = new OutputStreamWriter(httpConn.getOutputStream(), "UTF-8");
String requestRegistration = "[{\"cseq\":2," +
"\"callId\":\"LSRestfulTest.testStoreRegistration@10.182.107.197:5071\"," +
"\"contactAddress\":\"<sip:alice@10.182.107.197:5071>;expires=300\"," +
"\"methodsParam\":\"INVITE,BYE,CANCEL,ACK\"}]";
outWriter.write(requestRegistration);
outWriter.flush();
outWriter.close();
int responseCode = httpConn.getResponseCode();
assertEquals(HttpURLConnection.HTTP_UNAUTHORIZED, responseCode);
String digestValue = httpConn.getHeaderField(HttpAuthenticationUtils.HEADER_WWW_
AUTHENTICATE);
// Calculate the authorization header value from the authenticate header
String authorizeValue = caculateAuthorizationFromDigest(digestValue, DEST_URL, account_name,
account_pass, "POST");
httpConn = (HttpURLConnection)url.openConnection();
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("POST");
httpConn.setRequestProperty(HttpAuthenticationUtils.HEADER_AUTHORIZATION, authorizeValue);
httpConn.connect();
outWriter = new OutputStreamWriter(httpConn.getOutputStream(), "UTF-8");
outWriter.write(requestRegistration);
outWriter.flush();
outWriter.close();
responseCode = httpConn.getResponseCode();
assertEquals(HttpURLConnection.HTTP_OK, responseCode);
InputStream input = httpConn.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(input, "UTF-8"));
String responseContent = "";
String line = null;
while ((line = reader.readLine()) != null) {
responseContent += line;
}
      reader.close();

httpConn.disconnect();
  }
  void setHttpConnReqProperty(HttpURLConnection httpConn) throws ProtocolException {
    httpConn.setRequestProperty("Accept", "application/json");
    httpConn.setRequestProperty("Content-Type", "application/json");
    httpConn.setDoInput(true);
    httpConn.setDoOutput(true);
  }
```

Lookup an Address-of-Record

This API looks up all bindings of an AOR. An **HTTP response 200** message is returned on success.

HTTP Method

GET

URI

`proxyregistrarssr/locationservice/registration/`*`uri`*

Request Header

```
Accept application/json, Content-Type application/json
```

Request Body

None.

Response Body

```
[{
"aor":"sip:alice@example.com",
"contactUri":"sip:alice@10.182.101.231:5060",
"contactAddress":"<sip:alice@10.182.101.231:5060>;expires=3600",
"callId":"a97d0d177949304c@enpoYWkwMS5hcGFjLmJlYS5jb20.",
"cseq":7,
"qvalue":1.0,
"methodsParam":"INVITE,BYE,CANCEL,ACK",
"expiresParam":3600,
"expires":1335173253469,
"path":null,
"sipInstanceId":null,
"regId":null,
"remoteIP":"localhost",
"remotePort":2169,
"created":1335169653469,
"updated":1335169653469
}]
```

Example

Example 17–2 stores an AOR registration in the Proxy Registrar using the following parameters:

```
final String DEST_URL =
"/proxyregistrarssr/locationservice/registration/sip:alice@example.com";
private String account_name = "alice";
private String account_pass = "welcome1";
```

Example 17–2 Looking Up An AOR

```
public void LookupRegistration() throws Exception {
String restUrl = "http://127.0.0.1:7001" + DEST_URL;
URL url = new URL(restUrl);
HttpURLConnection httpConn = (HttpURLConnection)url.openConnection();
```

```
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("GET");
httpConn.connect();
String digestValue = httpConn.getHeaderField(HttpAuthenticationUtils.HEADER_WWW_
AUTHENTICATE);
String authorizeValue = caculateAuthorizationFromDigest(digestValue, DEST_URL, account_name,
account_pass, "GET");
httpConn = (HttpURLConnection)url.openConnection();
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("GET");
httpConn.setRequestProperty(HttpAuthenticationUtils.HEADER_AUTHORIZATION, authorizeValue);
httpConn.connect();
responseCode = httpConn.getResponseCode();
assertEquals(HttpURLConnection.HTTP_OK, responseCode);
InputStream input = httpConn.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(input, "UTF-8"));
String responseContent = "";
String line = null;
while ((line = reader.readLine()) != null) {
responseContent += line;
}
reader.close();
httpConn.disconnect();

  }
```

Clear All Address of Record Bindings

An **HTTP response 204** message is returned on success.

HTTP Method

DELETE

URI

`proxyregistrarssr/locationservice/registration/`*`uri`*

Where the URI is of the form: sip:*username@domain.com*

Parameters

None.

Request Header

None.

Request Body

None.

Response Body

None.

Examples

Example 17–3 clears an AOR registration in the Proxy Registrar using the following parameters:

```
final String DEST_URL =
"/proxyregistrarssr/locationservice/registration/sip:alice@example.com";
private String account_name = "alice";private String account_pass = "welcome1";
```

Example 17–3 Clearing AOR Binding

```
public void testClearAllBindings() throws Exception {
String restUrl = "http://127.0.0.1:7001" + DEST_URL;
URL url = new URL(restUrl);
HttpURLConnection httpConn = (HttpURLConnection)url.openConnection();
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("DELETE");
httpConn.connect();
String digestValue = httpConn.getHeaderField(HttpAuthenticationUtils.HEADER_
WWW_AUTHENTICATE);
String authorizeValue = caculateAuthorizationFromDigest(digestValue, DEST_URL,
account_name, account_pass, "DELETE");
httpConn = (HttpURLConnection)url.openConnection();
setHttpConnReqProperty(httpConn);
httpConn.setRequestMethod("DELETE");

httpConn.setRequestProperty(HttpAuthenticationUtils.HEADER_AUTHORIZATION,
```

```
authorizeValue);
httpConn.connect();
respCode = httpConn.getResponseCode();
assertEquals(HttpURLConnection.HTTP_NO_CONTENT, respCode);
httpConn.disconnect();

 }
```