Oracle® Communications Services Gatekeeper

Platform Development Studio - Developer's Guide

Release 4.1

January 2009

ORACLE

Oracle Communication Services Gatekeeper Platform Development Studio - Developer’s Guide, Release 4.1

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

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

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

Contents

Document Roadmap

Overview of the Platform Development Studio

Using the Eclipse Wizard

Description of a Generated Project

Communication Service Example

Service Enabler Example with SIP plug-in

Container Services

Communication Service Description

Annotations, EDRs, Alarms, and CDRs

Making Communication Services Manageable

Service Interceptors

Custom Service Level Agreements

Subscriber-centric Policy

Creating an EDR Listener and Generating SNMP MIBs

Converting Traffic Paths and Plug-ins to Communication Services

Policy

Callable Policy Web Service

Checklist

CHAPTER 1

Document Roadmap

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

- Document Scope and Audience
- Guide to This Document
- Terminology
- Related Documentation

Document Scope and Audience

This document describes the Oracle Communications Services Gatekeeper Platform Development Studio, a framework for creating and testing new extension Communication Services. The intended audience of this document consists of system integrators and field engineers who need to extend the out-of-the-box functionality of Oracle Communications Services Gatekeeper.

Guide to This Document

The document contains the following chapters:

Chapter 1, "Document Roadmap": This chapter

Chapter 2, "Overview of the Platform Development Studio": A high level description of the capabilities of the PDS

Chapter 3, “Using the Eclipse Wizard”: Setting up the Eclipse Wizard to generate extension projects

Chapter 4, “Description of a Generated Project”: The elements of a generated Communication Service project.

Chapter 5, “Communication Service Example”: A description of the example Communication Service provided with the Platform Development Studio.

Chapter 6, “Service Enabler Example with SIP plug-in”: A description of a Service Facade that uses the SIP Servlet Container.

Chapter 7, “Container Services”: Accessing Oracle Communications Services Gatekeeper’s Core functionality

Chapter 8, “Communication Service Description”: A component by component description of a Communication Service

Chapter 9, “Annotations, EDRs, Alarms, and CDRs”: Creating EDRs, CDRs, and Alarms

Chapter 10, “Making Communication Services Manageable”: Rendering extension Communication Services manageable by the Oracle Communications Services Gatekeeper Administration Console or other management tools

Chapter 11, “Service Interceptors”: An overview of service interceptors, a description of the standard out of the box ones, and of developing custom versions

Chapter 12, “Custom Service Level Agreements”: An overview of custom SLAs and a description of developing custom SLA enforcement logic

Chapter 13, “Subscriber-centric Policy”: Creating a policy mechanism based on individual subscriber preferences and permissions

Chapter 14, “Creating an EDR Listener and Generating SNMP MIBs”: Using and integrating external EDR listeners and generating SNMP MIBs

Chapter 15, “Converting Traffic Paths and Plug-ins to Communication Services”: Converting extensions built on previous versions of Oracle Communications Services Gatekeeper

Chapter 16, “Policy”: Using and extending Policy mechanisms in Communication Services

Chapter 17, “Callable Policy Web Service”: Integrating using the Callable Policy Web Service

Chapter 18, “Checklist”: A checklist for creating extensions.

Terminology

The following terms and acronyms are used in this document:

- Account—A registered application or service provider. An account belongs to an account group, which is tied to a common SLA
- Account group—Multiple registered service providers or services which share a common SLA
- Administrative User—Someone who has privileges on the Oracle Communications Services Gatekeeper management tool. This person has an administrative user name and password
- Alarm—The result of an unexpected event in the system, often requiring corrective action
- API—Application Programming Interface
- Application—A TCP/IP based, telecom-enabled program accessed from either a telephony terminal or a computer
- Application-facing Interface—The Application Services Provider facing interface
- Application Service Provider—An organization offering application services to end users through a telephony network
- AS—Application Server
- Application Instance—An Application Service Provider from the perspective of internal Oracle Communications Services Gatekeeper administration. An Application Instance has a user name and password
- CBC—Content Based Charging
- End User—The ultimate consumer of the services that an application provides. An end user can be the same as the network subscriber, as in the case of a prepaid service or they can be a non-subscriber, as in the case of an automated mail-ordering application where the subscriber is the mail-order company and the end user is a customer to this company
- Enterprise Operator —See Service Provider
- Event—A trackable, expected occurrence in the system, of interest to the operator
- HA —High Availability
- HTML—Hypertext Markup Language

- IP—Internet Protocol
- JDBC—Java Database Connectivity, the Java API for database access
- Location Uncertainty Shape—A geometric shape surrounding a base point specified in terms of latitude and longitude. It is used in terminal location
- MAP—Mobile Application Part
- Mated Pair—Two physically distributed installations of Oracle Communications Services Gatekeeper nodes sharing a subset of data allowing for high availability between the nodes
- MM7—A multimedia messaging protocol specified by 3GPP
- MPP—Mobile Positioning Protocol
- Network Plug-in—The Oracle Communications Services Gatekeeper module that implements the interface to a network node or OSA/Parlay SCS through a specific protocol
- NS—Network Simulator
- OAM —Operation, Administration, and Maintenance
- Operator—The party that manages the Oracle Communications Services Gatekeeper. Usually the network operator
- OSA—Open Service Access
- PAP—Push Access Protocol
- Plug-in—See Network Plug-in
- Plug-in Manager—The Oracle Communications Services Gatekeeper module charged with routing an application-initiated request to the appropriate network plug-in
- Policy Engine—The Oracle Communications Services Gatekeeper module charged with evaluating whether a particular request is acceptable under the rules
- Quotas—Access rule based on an aggregated number of invocations. See also Rates
- Rates—Access rule based on allowable invocations per time period. See also Quotas
- Rules—The customizable set of criteria - based on SLAs and operator-desired additions - according to which requests are evaluated
- SCF—Service Capability Function or Service Control Function, in the OSA/Parlay sense.

- SCS—Service Capability Server, in the OSA/Parlay sense. Oracle Communications Services Gatekeeper can interact with these on its network-facing interface
- Service Capability—Support for a specific kind of traffic within Oracle Communications Services Gatekeeper. Defined in terms of Communication Services
- Service Provider—See Application Service Provider
- SIP—Session Initiation Protocol
- SLA—Service Level Agreement
- SMPP—Short Message Peer-to-Peer Protocol
- SMS—Short Message Service
- SMSC—Short Message Service Centre
- SNMP—Simple Network Management Protocol
- SOAP—Simple Object Access Protocol
- SPA—Service Provider APIs
- SS7—Signalling System 7
- Subscriber—A person or organization that signs up for access to an application. The subscriber is charged for the application service usage. See End User
- SQL—Structured Query Language
- TCP—Transmission Control Protocol
- Communication Service—offers a service to an application
- USSD—Unstructured Supplementary Service Data
- VAS—Value Added Service
- VLAN—Virtual Local Area Network
- VPN—Virtual Private Network
- Oracle Communications Services Gatekeeper Container—The container hosting communication services.
- WSDL —Web Services Definition Language

- XML—Extended Markup Language

Related Documentation

This developer's guide is part of a set of documentation for Oracle Communications Services Gatekeeper itself. These documents include:

- *System Administrator's Guide*
- *Concepts and Architectural Overview*
- *Integration Guidelines for Partner Relationship Management*
- *SDK User Guide*
- *Managing Accounts and SLAs*
- *Statement of Compliance and Protocol Mapping*
- *Application Development Guide*
- *Communications Service Reference*
- *Handling Alarms*
- *Licensing*
- *Installation Guide*
- *RESTful Application Development Guide*
- *Platform Test Environment*

Overview of the Platform Development Studio

Oracle Communications Services Gatekeeper provides substantial functionality right out of the box. But because all networks are different, matching the particular requirements and capabilities of some networks sometimes means that Oracle Communications Services Gatekeeper must be extended or that certain aspects of it must be closely integrated with existing network functionality.The Platform Development Studio is designed to ease this process. It consists of two main sections:

- Creating New Communication Services
- Integration and Customization

Creating New Communication Services

Networks change. Existing functionality is parsed in new ways to support new features. New nodes with new or modified abilities are added. Because of Oracle Communications Services Gatekeeper's highly modular design, exposing these new features to partners is a straightforward proposition. The extension portion of the Platform Development Studio provides an environment in which much of the mechanics of creating extensions is taken care of, allowing extension developers to focus on only those parts of the system that correspond directly to their specific needs. This aspect consists of three main parts

- The Eclipse Wizard
- Example Communication Service
- The Platform Test Environment

The Eclipse Wizard

At the core of the extension portion of the Platform Development Studio is an Eclipse plug-in that creates projects based on the responses that the developer makes to an Eclipse Wizard. The developer supplies some basic naming information and the location of a WSDL for each application facing interface that the Communication Service is meant to support, and the Wizard generates either a complete Communication Service project, or a network plug-in only project. For more information on setting up the Eclipse Plug-in and running the Wizard, see Chapter 3, "Using the Eclipse Wizard." To see an example of a generated project, see Chapter 4, "Description of a Generated Project." To get an understanding of the Oracle Communications Services Gatekeeper features with which your Communication Service will interact, see Chapter 16, "Policy," Chapter 7, "Container Services,"Chapter 9, "Annotations, EDRs, Alarms, and CDRs," and Chapter 10, "Making Communication Services Manageable."

Example Communication Service

To give you a concrete sense of the task of generating a new Communication Service, the Platform Development Studio contains an entire example Communication Service, which is buildable and runnable. Based on a very simple Web Service interface and an equally simple model of an underlying network protocol, this Communication Service demonstrates the entire range of tasks that you will encounter in creating your own Communication Service. For more information, see Chapter 5, "Communication Service Example."

As an example a network protocol plug-in that uses the SIP Servlet container is provided in Chapter 6, "Service Enabler Example with SIP plug-in."

The Platform Test Environment

To simplify the testing of your Communication Service, the Platform Development Studio includes the Platform Test Environment, which provides an extensible suite of tools for testing Communication Services and the Unit Test Framework, which supplies an abstract base class, `WlngBaseTestCase`, which includes mechanisms for connecting to the Platform Test Environment. As well, there is a complete set of sample tools created to interact with the example Communication Service. For more information, see *Platform Test Environment*, a separate document in this set.

Integration and Customization

New Communication Services are not the only aspect of Oracle Communications Services Gatekeeper that can handled using the Platform Development Studio. To help integrate Oracle Communications Services Gatekeeper into the installation environment, three other aspects of customization are supported:

- Service Interceptors
- Subscriber-centric Policy
- Integration with External Systems

Service Interceptors

Service interceptors provide Oracle Communications Services Gatekeeper with a mechanism for intercepting and manipulating a request as it flows through any arbitrary Communication Service. They offer an easy way to modify the request flow, simplify routing mechanisms for plug-ins, and centralize policy and SLA enforcement. Out of the box, Oracle Communications Services Gatekeeper uses these modules as part of its internal functioning, but operators can also choose to create new interceptors, or to rearrange the order in which the interceptors are used, in order to customize their functionality. Chapter 11, "Service Interceptors" describes the request flow through interceptors, lists the standard interceptors and explains how to rearrange interceptors or to create new custom versions.

Subscriber-centric Policy

Out of the box the Oracle Communications Services Gatekeeper administration model allows operators to manage application service provider access to the network at increasingly granular levels of control. Using the Platform Development Studio, operators can extend that model to encompass their subscribers, giving the operator the ability to offer those subscribers a highly personalized experience while protecting their privacy and keeping their subscriber data safe within the operator's domain.

Operators create a Subscriber SLA, based on a provided schema, which describes sets of *service classes*. The service classes define access relationships with the services of particular Service Provider and Application Groups, along with default rates and quotas. *Profile providers* created by the operator or integrator using the provided Profile Provider SPI then associate those service classes with subscriber URIs, forming *subscriber contracts*. These contracts are used to evaluate requests and to generate subscriber budgets, which are used by the normal request traffic policy evaluation flow. A single subscriber can be covered by multiple subscriber contracts, based on that individual subscriber's desires. Chapter 13, "Subscriber-centric Policy" covers the process for setting this up.

Integration with External Systems

Finally, the Platform Development Studio provides mechanisms to support the integration of Oracle Communications Services Gatekeeper with external network systems, including:

- EDR listeners, covered in Chapter 14, "Creating an EDR Listener and Generating SNMP MIBs"
- Alarm monitoring using SNMP, covered in Chapter 14, "Creating an EDR Listener and Generating SNMP MIBs"
- Callable policy using JMX, covered in Chapter 17, "Callable Policy Web Service"

Additional integration points, not covered in the PDS, are provided by:

- The Partner Relationship Management interfaces, for creating Partner Management portals, covered in *Integration Guidelines for Partner Relationship Management*, a separate document in this set
- JMX for Management, for non-console based management, covered by the WLS documents *Developing Custom Management Utilities with JMX* and *Developing Manageable Applications with JMX*.

Using the Eclipse Wizard

This section describes using the Eclipse Wizard to generate Communication Services:

About the Eclipse Wizard

The Eclipse Wizard is a plug-in that enables an Eclipse user to create Oracle Communications Services Gatekeeper Communication Services. The extension projects are created using wizards that customize the project depending on which type of extension is being developed. Two types of extensions can be created:

- Communication Services
- Network protocol plug-ins for existing Service Facades (application-facing interfaces)

The Eclipse Wizard generates classes and Ant build files for both types of extensions. In addition, there is a separate build file with Ant targets for packaging the extension for deployment.

Configure Eclipse

Prerequisites

- Eclipse 3.2 or higher version must be installed
- Ant 1.6.5 must be installed
  Use the Ant provided with WebLogic Server. It is located in `$BEA_HOME/modules/org.apache.ant_1.6.5`.

Basic configuration of Eclipse environment

To do the basic configuration of the Eclipse environment:

1. Start Eclipse
2. Open the Preferences window, **Window**–>**Preferences...**
   a. In **Java**–>**Installed JREs**, make sure that the JRE used is the JRE installed with the Oracle Communications Services Gatekeeper. This is installed in `$BEA_HOME/<jdk version>/jre`
   b. In **Ant**–>**runtime**, make sure Ant Home is set to the directory in which you have installed Ant.

Configuring of the Eclipse Wizard

To configure the Eclipse Wizard, starting in Eclipse:

1. Open the Preferences window, **Window**–>**Preferences...**
2. In **OCSG Platform Development Studio**, configure the following:

   OCSG Home Directory The directory of the Oracle Communications Services Gatekeeper installation. This provides references to WebLogic Server APIs. In the default installation, this would be `$BEA_HOME/ocsg_4.1`.

   OCSG PDS Home Directory The directory of the Oracle Communications Services Gatekeeper Platform Development Studio installation. This provides references to Oracle Communications Services Gatekeeper APIs, extension points and third party APIs. In the default installation, this would be `$BEA_HOME/wlng_pds400`

   JDK Installation Directory The JDK installation directory for Oracle Communications Services Gatekeeper, for example `$BEA_HOME/jdk160_05`.

Logging Level The logging level of the Eclipse plug-in and the Ant tasks. Determines what level of detail to log by Eclipse. Select **All** for detailed logs, **Standard** for less detailed logs.

Using the Eclipse Wizard

Generating a Communication Service Project

A Communication Service project is based on a WSDL file and a set of attributes given when running the **Communication Service Project** wizard.

The WSDL defining the application-facing interface must adhere to the following:

- Attribute name in `<wsdl:service>` must include the suffix **Service**.
- Attribute name in <wsdl:port> must be the same as the name attribute in `<wsdl:service>`, excluding the suffix **Service**.

To generate a Communication Service project:

1. In Eclipse, choose **File**–>**New Project**.

   This opens the **New Project** window.

In this window...	Perform the following action...
Select Wizard	Make sure **OCSG Platform Development Studio–>Communication Service Project** is selected. Click **Next** to proceed. You may cancel the wizard at any time by clicking **Exit**. You may go back to a previous window by clicking **Previous**.
Create a Communication Service	Enter a **Project Name** and choose a location for your project. You can choose: 1. To create an entirely new Communication Service 2. To create a new Service Facade (application-facing interface) and the common parts of the Service Enabler layer for an existing plug-in 3. To create a new network plug-in that uses the Service Facade and common parts of the Service Enabler of a currently existing Communication Service. If you wish to do **3**, check the check-box **Use predefined communication service** and from the drop-down list select the Service Facade for which you want to create a plug-in. If you wish to do **1** or **2**, leave the box unchecked. Click **Next** to continue. If you checked the **Use predefined** check box, the **Define the Plug-in Information** window is displayed. Go to **Define the Plug-in Information** instructions below. If you did not check it, the **Define the Communication Service** is displayed.
Define the Communication Service • **Configure Service WSDL Files**	For each WSDL file that includes the service definition *to be implemented* by the new Communication Service: Click ➕, browse to the WSDL file, select it, and click **OK**.

In this window...	Perform the following action...
Define the Communication Service • **Configure Callback WSDL Files**	For each WSDL file that includes the callback service definition *to be used* by the new Communication Service in sending information to the service provider’s application: Click ✚, browse to the WSDL file, select it, and click **OK**.

In this window...	Perform the following action...
Define the Communication Service • **Communication Service Properties**	**Company:** Set your company name, to be used in `META-INF/MANIFEST.MF`. **Version:** Set the version, to be used in `META-INF/MANIFEST.MF`. **Identifier:** Create an identifier to tie together a collection of Web Services. Will be a part of the names of the generated war and jar files and the service type for the Communication Service: `<Communication Service identifier>.war` and `<Communication Service identifier>_callback.jar` **Service Type:** Set the service type. Used in EDRs, statistics, etc. For example: SmsServiceType, MultimediaMessagingServiceType. **Java Class Package Name:** Set the package names to be used. For example: com.mycompany.service **Web Services Context path:** Set the context path for the Web Service.For example: myService **SOAP to SOAP**: Check this box to generate a Service Facade that can be a part of a SOAP to SOAP Communication Service. **REST**: Check this box to generate a RESTful Service Facade for the Communication Service.

In this window...	Perform the following action...
Define the Plug-in information	For each plug-in to be created in the Communication Service project: Click This opens a pop-up window with the following fields: **Protocol:** An identifier for the network protocol the plug-in implements. Used as a part of the names of the generated jar file: `<Communication Service identifier>_<protocol>.jar` and the service name `Plugin_<Communication Service identifier>_<protocol>` **Schemes:** Address schemes the plug-in can handle. Use a comma separated list if multiple schemes are supported. For example: tel: or sip: **Package Name:** Package names to be used. **Company:** Used in `META-INF/MANIFEST.MF`. **Version:** Used in `META-INF/MANIFEST.MF`. Choose which **Type** of plug-in to generate: **SOAP**: Select to this radio-button to generate a generic, protocol-neutral, plug-in. **SOAP to SOAP**: Select to this radio-button to generate a plug-in for a SOAP to SOAP Communication Service. **SIP**: select to this radio-button to generate a plug-in that connects to a SIP network using a SIP Servlet. Click **OK**. The plug-in definitions are added to the list of plug-ins. Click to remove the selected plug-in. Click **Finish** to start the code generation for the plug-in(s).

Adding a Plug-in to a Communication Service Project

To add a plug-in to an existing Communication Service project:

1. In the Eclipse package explorer, right-click the project for the Communication Service project, and choose **Properties**.

   This opens the **Properties** window for the Communication Service project.

In this window...	Perform the following action...
Plugin Configuration	A list of plug-ins defined for the Communication Service project is displayed. For each plug-in to be created in the Communication Service project: Click This opens a pop-up window with the following fields: **Protocol:** An identifier for the network protocol the plug-in implements. Used as a part of the names of the generated jar file: `<Communication Service identifier>_<protocol>.jar` and the service name `Plugin_<Communication Service identifier>_<protocol>` **Schemes:** Address schemes the plug-in can handle. Use a comma separated list if multiple schemes are supported. For example: tel: or sip: **Package Name:** Package names to be used. **Company:** Used in `META-INF/MANIFEST.MF`. **Version:** Used in `META-INF/MANIFEST.MF`. Choose which **Type** of plug-in to generate: **SOAP**: Select to this radio-button to generate a generic, protocol-neutral, plug-in. **SOAP to SOAP**: Select to this radio-button to generate a plug-in for a SOAP to SOAP Communication Service. **SIP**: select to this radio-button to generate a plug-in that connects to a SIP network using a SIP Servlet. Click **OK**. The plug-in definitions are added to the list of plug-ins. Click **Finish** to start the code generation for the plug-in(s).

Removing a Plug-in from a Communication Service Project

To remove a a plug-in from an existing Communication Service project:

1. In the Eclipse package explorer, right-click the project for the Communication Service project, and choose **Properties**.

   This opens the Properties Window for the Communication Service project.

In this window...	Perform the following action...
Plugin Configuration	A list of plug-ins defined for the Communication Service project is displayed. For each plug-in to be removed from the Communication Service project: • Select the plug-in to be removed and click The plug-in definitions are removed from the list. • Click **Apply** to remove the plug-in part(s) from the Communication Service project. **Warning**: This removes *all* parts of the project, including any manually edited or added files. • Click **Restore Defaults** to restore the plug-in definition list.

2. Click **OK** or **Cancel** to close the **Properties** window.

CHAPTER 4

Description of a Generated Project

The section describes a project generated from the Eclipse Wizard:

Generated project

Communication Service Project

Generating a Communication Service project creates the directory structure illustrated in Listing 4-1.

The base directory is the directory given in the Eclipse Wizard input field **`Identifier`**. It contains the following files:

- `build.properties:`, properties file for the build files:
  - wlng.home is set to $OCSG_HOME, the base directory for the Oracle Communications Services Gatekeeper installation.
  - pds.home is set to $PDS_HOME, the base directory for the Platform Development Studio.
  - wls.home is set to $WLS_HOME, the base directory for the WebLogic Server installation.
- `build.xml`: the main file for the project, that is the build file for the Communication Service and references to any other plug-in specific build files in the project. See Main Build File.
- `common.xml:` properties, ant task and targets used by all build files in the project.

The directories and files in bold in Listing 4-1 are generated when building the common parts of the Communication Service; the others are generated by the Eclipse Wizard.

Listing 4-1 Generated project for Communications Services Common

```
<Eclipse Project Name>
+- build.properties
+- common.xml
+- build.xml
+- <Identifier given in Ecplise Wizard>
|  +- dist //Generated by target dist in <Eclipse Project Name>/build.xml
|  |  +- <Package name>.store_<version.jar // Example store configuration
|  |  +- wlng_at_<Identifier>.ear //Deployable in access tier
|  |  +- wlng_nt_<Identifier>.ear //Deployable in network tier
|  +- common
|  |  +- build.xml //Build file for the common parts of the communication service
|  |  +- dist //Generated by target dist on
             //<Eclipse Project Name>/common/build.xml
|  |  |  +- request_factory_skel //Skeletons for the RequestFactory,
                                 //one class for each service WSDL
|  |  |  +- tmp //Used during build. Contains classes, source,
                //definitions, WSDLs, templates, and more.
|  |  |  +- <Identifier>.war // Web Service implementation
|  |  |  +- <Identifier>_callback.jar // Service callback EJB for
                                      //the communication service
|  |  |  +- <Identifier>_callback_client.jar //Service call-back EJB used by
                                           // the plug-in.
|  |  |  +- <Identifier>_service.jar // Service EJB
                                      // for the communication service
|  |  +- resources // Contains application.xml and weblogic-application.xml
                   // for the access and network tier EAR files respectively.
```

```
                  // The files are packaged in the EAR files META-INF directory
|  |  |  +- handlerconfig.xml //SOAP Message Handler
|  |  +- src // Source directory for communication service common
|  |  |  +- <Package name>/plugin
|  |  |  | +- <Web Services interface>PluginFactory // One per interface
                                                     // defined in the
                                                    // Service WSDL files.
```

The SOAP Message Handler definition file, `handlerconfig.xml`, can be edited in order to change, add, or remove SOAP Message Handlers. If modified, it will be taken into account the next time the Communication Service or plug-in is rebuilt.

The following exception definitions are added:

- PolicyException - Any policy based exceptions.
- RoutingException - Any exceptions during the routing of the request.
- ServiceException - Any other internal exceptions.

The exceptions are added only to the service facade, not to the plug-in to network interface.

If the exceptions listed above are present in the original WSDL they are reused; if not they are added.

RESTFul Service Facade

A RESTful Service Facade can be generated using the Eclipse wizard. The sections below describe the default generation of the RESTful Service Facade and how to modify the default implementation.

Default RESTful Service Facade

When a RESTful Service Facade is generated, the following is generated in addition to the directory structure described in Listing 4-1:

- `rest_<Identifier>.war`, in the directory `common/dist`

- `rest/<Identifier>/index.html`, in the directory `common/dist/tmp/wars/rest_<Identifier>`
- `rest-config.xml`, in the directory `<identifier>/common/resources/facade/rest`

The RESTful Service Facade Web Application `rest_<Identifier>.war` is packaged in the Access Tier EAR file. The context root is `rest/<Identifier>`.

An API description is generated in the directory `common/dist/tmp/wars/rest_<Identifier>`. It describes each operation, including URI, HTTP-method, request- and response content-type, request- and response, and errors.

The generated RESTful API has a default implementation, which can be changed by editing `rest-config.xml` and re-building the Service Facade. The API description is updated so it reflects any changes done in the configuration file.

The default implementation of the generated RESTful Service Facade has the following attributes for application-initiated requests.

The HTTP method is POST.

The URL to a default RESTful resource is:

http://<host>:<port>/rest/<context-root>/<interface>/<operation>/<path-info>?<name[1]>=<value[1]&><name[2]>=<value[2]&...<name[n]>=<value[n]>

Where:

- <host> and <port> depend on the Oracle Communications Services Gatekeeper installation, and on the server where the RESTful Service Facade is deployed.
- <context-root> is specified in the field Web Services Context Path in the Eclipse wizard.
- <interface> is derived from the interface name in the Service WSDL.
- <operation> is derived from the operation name in the Service WSDL.
- <path-info> and the name-value pair should not be present in the URI since the default HTTP method is POST. See Table 4-2 for information on how this behavior can be changed. <path-info> and the queryString are not present by default.

The HTTP content-type for the request is application/json. The HTTP request body contains a JSON formatted object that corresponds to the input message of the operation as defined in the Service WSDL.

The HTTP content-type for the response is application/json. The HTTP response body contains a JSON formatted object that corresponds to the output message of the operation as defined in the

Service WSDL. The HTTP response body for an error contains a JSON formatted object that corresponds to the error message of the operation as defined in the Service WSDL.

For example the Parlay X 2.1 Short Messaging Service defines the operation startSmsNotification. Using the WSDLs for this service, the corresponding RESTful resource is according to Table 4-1. This information is provided in the generated API documentation.

Table 4-1 Example of a RESTful resource as used by an application

URI	rest/sms/SmsNotificationManager/startSmsNotification
HTTP Method	POST
Request Content-Type	application/json

Request Body	{
	"reference": {
	"correlator": "String",
	"endpoint": "URI",
	"interfaceName": "String"
	},
	"smsServiceActivationNumber": "URI",
	"criteria": "String"
	}

Response Body	Empty.

Error Response	{"error":{
	"type":"org.csapi.schema.parlayx.common.v2_1.ServiceException"
	"message":"String"
	}}

Error Response	{"error":{
	"type":"org.csapi.schema.parlayx.common.v2_1.PolicyException"
	"message":"String"
	}}

The Bayeux protocol is used to deliver network-triggered messages, or notifications, to an application. For more information on the Bayeux protocol, see the “Bayeux Protocol 1.0draft1” document at http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html.

The RESTful Service Facades rely on the publish-subscribe model supported by the Publish-Subscribe Server functionality of Oracle WebLogic Server. The communication service delivers the network-triggered traffic to the publish-subscribe server channel, from which the application Bayeux client fetches it. For more information on this model, please see section “Using the HTTP Publish-Subscribe Server” in *Developing Web Applications, Servlets, and JSPs For Oracle WebLogic Server* at http://download.oracle.com/docs/cd/E12840_01/wls/docs103/webapp/.

An application needs to subscribe for notifications. The application provides an endpoint URI to receive notifications on. In Parlay X, the operations are normally named according to start<Service name>Notification, for example startSmsNotification. In a RESTful environment, the endpoint URI is the name of the Bayeux channel, must start with the string `/bayeux/` in order to be recognized as a RESTful endpoint. Immediately following this keyword, the application must provide the application instance ID that uniquely identifies the application. An example of an endpoint is `/bayeux/myApplicationID/myInterface`. The application’s Bayeux client must perform a hand-shake, connect to the publish-subscribe server and subscribe to the channel that is being created for the notification.

The publish-subscribe server URI to use for the Bayeux connect is:

<http:>//<host>:<port>/rest/<context-root>/notifications

Where:

- <host> and <port> depend on the Oracle Communications Services Gatekeeper installation.
- <identifier> is specified in the field Identifier in the Eclipse wizard.

Notifications are sent via Bayeux Deliver Event messages, see http://svn.cometd.org/trunk/bayeux/bayeux.html. The HTTP response body contains a JSON formatted object that corresponds to the output message of the operation as defined in the Service Callback WSDL.

Typically, the publish-subscribe server URI to use for the Bayeux connect should be returned to the application in the in the header of the response to start a notification. Do do this, rest-config.xml should be updated with a <response-header> element, see Customize the RESTful Service Facade.

Customize the RESTful Service Facade

The following can be customized for the RESTFul Service Facade:

- HTTP method
- URI Mapping
  - servlet-path
  - pathinfo
  - request parameter
- Data binding
  - path-info-param
  - request-param
- Other
  - additional response headers
  - custom handler chain for an operation
  - custom data type adapters
  - custom HTTP status code mappings for errors

The mappings are defined in rest-config.xml, according to the XSDs `rest-config.xsd` and `error-mappings.xsd`, located in `$OCSG_HOME/applications/rest.jar`. Table 4-2 contains a description of the mappings.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<resources>	Main tag. Contains: <resource>, one (1) or more. <handler-chain>, zero (0) or more. <data-type-adapter>, zero (0) or more. <notification>, zero (0) or more. <binding>, one (1) or more. <error-mappings>, zero (0) or one (1).
<resource>	Parent element: <resources>. Contains the following element: <operation>, one (1) or more. Has the attribute: • uri Defines a part of the URI for a RESTful resource. All resources used for application-initiated traffic need this definition. If the URI used by an application is: `http://host:port/<context-root/`**`<servlet-path>`**`/<path info>?<name1>=<value1>&<name2>=<value2>` The attribute uri corresponds to `<servlet-path>` in the URI.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<operation>	Parent element: <resource>. Contains the following elements: • <http-method>, exactly one (1). • <request-type>, zero (0) or one (1). • <request-param>, zero (0) or more. • <path-info-param>, zero (0) or one (1). • <target>, exactly one (1). • <handler-chain>, zero (0) or one (1). • <response-header>, zero (0) or more. • <response-type>, zero (0) or one (1). • <empty-response>, zero (0) or one (1). Defines an operation that corresponds to the RESTful resource.
<http-method>	Defines which HTTP operation to use for the resource. Use GET, POST, PUT, or DELETE. By default, the method is POST. For other methods, the request URI will differ and some elements become mandatory or not used.
<request-type>	Parent element: <operation>. Used for API documentation generation only. It has no run-time effect. Defines the content-type header of the incoming HTTP request. Default value is application/json. Enumeration: • application/json • multipart/form-data (for example, when using HTTP attachments)

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description

<request-param>	Parent element: <operation>.
	Has the attributes:
	• name
	• value (optional)
	Defines expected request name value pairs.
	Useful for sending a JSON object via HTTP GET, in which case the value should be an encoded JSON string representing the input object. Only one JSON object is supported.
	Also useful for overloading the resource URI, for example invoking different operations on the same resource, in which case the value will be specified as a constant.
	Every incoming request in the format of:
	`http://host:port/<context-root/<servlet-path>/<path info>?<name1>=<value1>`
	will invoke the given operation.
	If the URI used by an application is:
	`http://host:port/<context-root/<servlet-path>/<path info>?<name1>=<value1>&<name2>=<value2>`
	The attribute name corresponds to either `<name1> or <name2>` in the URI.
	If either <value1> or <value2> is defined as a constant, that attribute value shall be set to this constant. Format the value as a JSON object.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<path-info-param>	Parent element: <operation>. Has the attribute: • name Defines a a part of the URI for a RESTful resource. This element is optional. When present, the value will be taken from the <pathInfo> component of request URI, and used to populate the field of the target operation input parameter. The attribute name specifies the name of the field to be populated. If the URI used by an application is: http://host:port/<context-root/<servlet-path>/**<path info>**?<name1>=<value1>&<name2>=<value2> The attribute name corresponds to `<pathinfo>` in the URI.
<target>	Parent element: <operation>. Has the attributes: • service • class • method Defines how the RESTful resource maps to the Java implementation of the service. The attribute service is derived from the interface type in the WSDL. The attribute class defines the generated class that implements the interface defined in the WSDL. The pattern is: `<package name from Eclipse wizard>.<Service name from wizard>.rest.<Interface name from WSDL>RestImpl` The attribute method defines the method in the class to bind RESTful resource. The name of the method is derived from the operation defined in the WSDL.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<handler-chain>	Parent element: <resources> or <operation>. This element defines a handler chain. When defined under <operation>, it refers to provided handler chain names or custom handler chains. If it is a custom handler chain it also needs to be defined under <resources>. If it is a provided handler chain, it is only necessary to refer to the name. When defined under <resources>, it defines a named handler chain to be invoked prior to the request being handed off to the generated RESFul Service Facade implementation and prior to a response being handed off to the calling application. There are a set of available handler chains available. New ones can be added. The available handler chains include: • default, this is the default handler chain. It has the following sequence defined: SessionIdHandler –> ServiceCorrelationIdHandler –> ExtendingParametersHandler • default-with-attachment, this handler chain shall be used when an a RESTful resource uses attachments. It has the following sequence defined: SessionIdHandler –>AttachmentHandler–> ServiceCorrelationIdHandler –> ExtendingParametersHandler • empty, this handler chain does not do anything. For default behavior use default or default-with-attachment. See Using a Custom Handler Chain for information on how to create a custom handler chain.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<response-header>	Parent element: <operation>. Has the attributes: • name • value Defines HTTP response headers to be returned to the application. The attribute name is the name of the response header. The attribute value attribute can be a constant or a variable. If it is a variable, the format is ${field name of return value}, where the variable is replaced with the runtime value of the field. Nested fields are not supported. The variable tokens for each operation is found in the generated API docs. The variable ${rest-facade-url} is predefined. It is replaced with the URL to the incoming request the RESTFul Service Facade. Example: <response-header name="Location" value="${rest-facade-url}/delivery-status/${result}"/>
<response-type>	Parent element: <operation>. For API documentation only, no run-time effect. Defines the content-type header of the outgoing HTTP response. Enumeration: Defines the content-type header of the outgoing HTTP response. Default value is application/json. Enumeration: • application/json • multipart/mixed
<empty-response>	Parent element: <operation>. Defines that the HTTP response for the enclosing operation does not have an entity body.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<data-type-adapter>	Parent element: <resources>. Contains the following elements: • name, exactly one (1). • target-field, exactly one (1). The element target-field has the attributes: • class • name Defines a data type adapter. This is needed only if the target Java type can be mapped to more than one XML schema types, for example byte[] to xsd:hexBinary or xsd:base64Binary. There are two adapters available: • base64binary • hexBinary The element name defines the data type adapter to use for the given target fields. The element target specifies the class for the object and the member variable in the object. Examples: <data-type-adapter> <name>base64binary</name> <target-field class="org.csapi.schema.parlayx.sms.send.v2_2.local.SendSmsLogo" name="image"/> </data-type-adapter> <data-type-adapter> <name>hexBinary</name> <target-field class="com.acompany.schema.example.data.send.local.SendData" name="binaryField"/> </data-type-adapter>

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<notification>	Parent element: <resources>. Contains the following elements: • <service>, exactly one (1). • <data>, one (1) or more. For API documentation only, no run-time effect. Defines the message format used to notify an application of a network-triggered operation. The operation is defined in the Service Callback WSDL. All resources used for network-triggered traffic needs this definition.
<service>	Parent element: <notification> Derived form the WSDL for the Service Callback WSDL. Example: MessageNotification
<data>	Parent element: <notification> Has the attributes: • id • class Defines the data in a notification sent to an application. The attribute id defines the id of the notification. This is the same as the operation defined in the Service Callback WSDL. The attribute class defines the generated class that specifies the notification. The class is generated based on the Service Callback WSDL. Example: <data id="notifyMessageReception" class="org.csapi.schema.parlayx.multimedia_messaging.notification.v2_4.local.NotifyMessageReception"/>

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<binding>	Parent element: <resources> Has the attributes: • service • schema For API documentation only, no run-time effect. No need to modify. Defines the binding between the attribute service defined in the element <target> and the Service WSDL. The attribute service identifies the service name. The attribute schema identifies the Service WSDL. Example: <binding service="SendMessage" schema="parlayx_mm_send_interface_2_4.wsdl"/>
<error-mappings>	Parent element: <resources> Contains the following elements: • <error-mapping>, zero (0) or more.
<error-mapping>	Parent element: <error-mappings> Contains the following elements: • <http-status-code>, exactly one (1) • <http-method>, zero (0) or one (1) • <error>, one (1) or more. Describes how a set of exceptions thrown by the RESTful Service Facade or Service Enabler maps to a HTTP status code. Default behavior is defined in default-error-mapping.xml. Custom mapping takes precedence.
<http-status-code>	Parent element: <error-mapping> Defines the HTTP status code to return.

Table 4-2 Structure and description of rest-config.xml

Element/Type	Description
<http-method>	Parent element: <error-mapping> Defines the HTTP method used for the original request. If omitted the mapping is valid for all HTTP request methods.
<error>	Parent element: <error-mapping> Has the attributes: • class • id-field (optional) • id-value (optional) The attribute class defines the class that defines the exception. The attribute id-field defines which member variable in the exception to match. The attribute id-value defines the value of the member variable to match.

Custom URL Mapping Example

For a URL in the format:

```
http://host:port/<context-root/<servlet-path>/<pathinfo>?<name1>=<value1>&<nam
e2>=<value2>
```

The following applies:

- `servlet-path` must match the attribute `uri` of the resource `element`.
- `pathinfo` must match the attribute `name` of the element `path-info-param`. It identifies a unique resource, such as a correlator. Note that this element is optional. If not present in the XML configuration file, it should not be present in the URL.
- request parameters:
  - `name` must match the attribute `name` of the element `request-param`.
  - `value` must match the attribute `value` of the element `request-param`.

For application-initiated operations, each resource URI is mapped to an HTTP method and an implementing class, for example:

```
<resource uri="/SendSms/sendSms">
```

```
  <operation>
    <httpMethod>POST</httpMethod>
    <target method="sendSms"
    class="com.acompany.arestservice.rest.SendSmsRestImpl"
service="SendSms"/>
  </operation>
</resource>
```

The names of the generated classes are derived from the package name given in the Eclipse wizard and the interface name derived from the WSDL:

```
<package name from wizard>.<service name from wizard>.rest.<Interface name
from WSDL>RestImpl
```

The method name is derived from the WSDL. The resource URI is derived from the namespace definition in the WSDL. The element <httpMethod> defines the HTTP method to use, either POST, GET, PUT or DELETE.

For network-triggered operations, each notification service is mapped to one or more classes that contain the data and the method used to deliver the data, for example:

```
<notification>
  <service>SmsNotification</service>
    <data class=
      "org.csapi.schema.parlayx.sms.notification.v2_2.local.NotifySmsRecep
tion"
      id="notifySmsReception"/>
    <data class=
       "org.csapi.schema.parlayx.sms.notification.v2_2.local.NotifySmsDeli
veryReceipt"
       id="notifySmsDeliveryReceipt"/>
</notification>
```

The classes and the method name are derived from the WSDL.

Using a Custom Handler Chain

A custom handler chain can be defined if additional processing of the request needs to be done before a request is passed on to the Service Enabler or back to an application.

A handler chain is defined as a set of handlers. A handler chain is named and referred to in rest-config.xml.

The existing handlers are:

- SessionIdHandler, which handles session IDs and extracts the IDs from the request.
- ServiceCorrelationIdHandler, which handles service correlation and extracts the IDs from the request.
- ExtendingParametersHandler, which handles tunnelled parameters.
- AttachmentHandler, which handles HTTP attachments.

A custom handler must implement the interface.

```
public interface com.bea.wlcp.wlng.rest.handler.Handler
```

The method handleRequest is invoked before a request is passed on to the Service Enabler.

The method handleResponse is invoked before a response is returned to an application.

The chain is defined in rest-config.xml and all classes in the chain must be packaged in the WAR file for the restful service facade.

Plug-in

When creating a plug-in for a given Communication Service, the directory structure illustrated in Listing 4-2 is created under the top-level directory. The base directory depends on the type of Communication Service the plug-in belongs to, such as, for example, `px21_multimedia_messaging,` or `px21_sms`. It also depends on whether the plug-in is for an existing Communication Service or for a new one.

If the plug-in is for an existing Communication Service, it is generated under one of the following directories:

- `px30_audio_call` for plug-ins for Parlay X 30 Audio Call
- `px21_call_notification` for Parlay X 2.1 Call Notification
- `px30_call_notification` for Parlay X 3.0 Call Notification

- `px21_multimedia_messaging` for Parlay X 2.1 Multimedia Messaging
- `px21_presence` for Parlay X 2.1 Presence
- `ews_push_message` for Extended Web Services WAP Push
- `px21_sms` for Parlay X 2.1 Short Messaging
- `ews_susbcriber_profile` for Extended Web Services Subscriber Profile
- `px21_terminal_location` for Parlay X 2.1 Terminal Location
- `px21_third_party_call` for Parlay X 2.1 Third Party Call
- `px30_third_party_call` for Parlay X 3.0 Thrid Party Call

If it is for a new Communication Service, the base directory is given in the **`Identifier`** entry field in the Eclipse Wizard.

The base directory contains the directory `plugins`, which contains subdirectories for each protocol that is being added. The names of the directories are the same as the name chosen for the **`Protocol`** field in the Eclipse Wizard.

Each of the sub-directories for a plug-in contains the following files:

- `build.xml`: The build file for the plug-in, see Plug-in Build File.

Each plug-in sub-directory also contains the directories:

- `config`: The directory that includes an instancemap that will be used by the InstanceFactory to create instances for the plug-in interface implementations.
- `dist`: The directory where the final deployable plug-in jar will end up. If a new plug-in skeleton is generated from the build file it will be generated here.
- `resources`: The directory that contains deployment descriptors for the plug-in.
- `src`: The directory that contains the generated plug-in code.
- `storage`: The directory that contains the configuration file for the Storage service.

The directories and files in bold in Listing 4-2 are generated when building the plug-in, the others are generated by the Eclipse Wizard.

Listing 4-2 Generated project for a plug-in

```
|  +- plugins // Container directory for all plug-ins for
              // the communication service
|  |  +- <Protocol> // One specific plug-in
|  |  |  +- build.xml // Build file for the plug-in
|  |  |  +- build // Used during the build process
|  |  |  +- config //
|  |  |  |  +- instance_factory
|  |  |  |  |  +- instancemap //Instance map
|  |  |  +- dist // Generated by target dist in build.xml for the plug-in
|  |  |  |  +- <Identifier>_<Protocol>_plugin.jar
|  |  |  |  +- <Package name>.store_<version>.jar
|  |  |  +- resources // Contains parts of weblogic-extension.xml
                      // for the network tier EAR file.
                     // the file is packaged in the EAR file’s META-INF directory
|  |  |  +- src
|  |  |  |  +- <Package name> // Directory structure reflecting
                              // plug-in package name
|  |  |  |  |  +- management // Example MBean
|  |  |  |  |  |  +- MyTypeMBean.java
|  |  |  |  |  |  +- MyTypeMBeanImpl.java
|  |  |  |  |  +- <Web Services interface> // One per Service WSDL
|  |  |  |  |  |  +- north
|  |  |  |  |  |  |  +- <Web Services interface>PluginImpl.java
                                 // Implmentation of the interface
|  |  |  |  |  +- <Type>PluginInstance.java
|  |  |  |  |  +- <Type>PluginService.java
```

```
                                // PluginService implementation
|  |  |  +- storage //Example of a store configuration.
|  |  |  |  +- wlng-cachestore-config-extensions.xml
```

SOAP2SOAP Plug-in

When creating a SOAP2SOAP plug-in, the directory structure described in Plug-in is created under the top-level directory. In addition, the directories and files in Listing 4-3 are generated. The directories and files in bold are created when building the plug-in; the others are generated by the Eclipse Wizard.

Note: Only the deployable artifacts are relevant. The generated code for the SOAP2SOAP type of plug-ins should not be modified.

Listing 4-3 Generated project for a SOAP2SOAP plug-in

```
|  +- plugins // Container directory for all plug-ins for
              // the communication service
|  |  +- <Protocol> // One specific plug-in
|  |  |  +- build.xml // Build file for the plug-in
|  |  |  +- build // Used during the build process
|  |  |  +- config //
|  |  |  |  +- instance_factory
|  |  |  |  |  +- instancemap //Instance map
|  |  |  +- dist // Generated by target dist in build.xml for the plug-in
|  |  |  |  +- <Identifier>_<Protocol>_plugin.jar
|  |  |  |  +- <Package name>.store_<version>.jar //unused, empty
|  |  |  +- resources // Contains parts of weblogic-extension.xml
                      // for the network tier EAR file.
                     // the file is packaged in the EAR file’s META-INF directory
```

```
|  |  |  |  +- client_handlerconfig.xml // SOAP Message Handler
|  |  |  +- src
|  |  |  |  +- <Package name> // Directory structure reflecting
                              // plug-in package name
|  |  |  |  |  +- client // Implementation of Web Service client
|  |  |  |  |  |  +- <Web Services interface>_Stub.java
|  |  |  |  |  |  +- <Web Services interface>.java
|  |  |  |  |  |  +- <Web Services interface>Service_Impl.java
|  |  |  |  |  |  +- <Web Services interface>Service.java
|  |  |  |  |  +- <Web Services call-back interface> // One per Call-back WSDL
|  |  |  |  |  |  +- south
|  |  |  |  |  |  |  +- <Web Services interface>PluginSouth.java
                                 // Interface for network-triggered requests
|  |  |  |  |  |  |  +- <Web Services interface>PluginSouthImpl.java
                                 // Implementation of the interface
|  |  |  |  |  +- <Web Services interface> // One per Service WSDL
|  |  |  |  |  |  +- north
|  |  |  |  |  |  |  +- <Web Services interface>PluginImpl.java
                                 // Implementation of the interface
|  |  |  |  |  +- <Type>PluginInstance.java
|  |  |  |  |  +- <Type>PluginService.java
                                // PluginService implementation
|  |  |  +- storage //Example of a store configuration. Empty.
|  |  |  +- wsdl // WSDLS and XML-to-Java mappings.
|  +- <Identifier>_callback.war // Web Service implementation
                                        // for the SOAP2SOAP plug-in
```

As illustrated in Listing 4-3, a WAR file for the plug-in is generated. This WAR file contains the Web Service for network-triggered requests. It is only generated if there is a notification WSDL defined at generation-time. It will be packaged in the EAR for the Service Enabler.

The SOAP Message Handler definition file, client_handlerconfig.xml, can be edited in order to change, add, or remove SOAP Message Handlers. If modified, the ant target rebuild.ws in the plug-in build file needs to be invoked

SIP Plug-in

When creating a SIP plug-in, the directory structure described in Plug-in is created under the top-level directory. In addition, the directories and files in Listing 4-4 are generated. The directories and files in bold are created when building the plug-in; the others are generated by the Eclipse Wizard.

Listing 4-4 Generated project for a SIP plug-in

```
|  +- plugins // Container directory for all plug-ins for
              // the communication service
|  |  +- <Protocol> // One specific plug-in
|  |  |  +- build.xml // Build file for the plug-in
|  |  |  +- build // Used during the build process
|  |  |  +- config //
|  |  |  |  +- instance_factory
|  |  |  |  |  +- instancemap //Instance map
|  |  |  |  +- sip
|  |  |  |  |  +- WEB-INF
|  |  |  |  |  |  +- sip.xml
|  |  |  |  |  |  +- web.xml
|  |  |  +- dist // Generated by target dist in build.xml for the plug-in
|  |  |  |  +- <Identifier>_<Protocol>_plugin.jar
|  |  |  |  +- <Identifier>_<Protocol>_sip.war
|  |  |  |  +- <Package name>.store_<version>.jar
|  |  |  +- resources
|  |  |  |  +- META-INF
|  |  |  |  |  +-weblogic-extension.xml
|  |  |  |  |  +-application.xml
|  |  |  +- src
|  |  |  |  +- <Package name> // Directory structure reflecting
                              // plug-in package name
|  |  |  |  |  +- servlet // Implementation of the SIP Servlet
|  |  |  |  |  |  +- <Identifier>Servlet.java
|  |  |  |  |  +- <Identifier>SipHelper.java
|  |  |  +- storage //Example of a store configuration. Empty.
```

As illustrated in Listing 4-3, a set of additional classes and configuration files for the SIP type plug-in is generated compared to the standard plug-in.

Table 4-1Contains a summary of the added items.

Table 4-3 Additional files generated for a SIP plug-in

File	Description
sip.xml	SIP Application deployment descriptor. See the document *Developing SIP Applications*, a part of the documentation for Oracle Converged Application Server at http://download.oracle.com/docs/cd/E13153_01/wlcp/wlss40/programming/index.html
web.xml	HTTP Servlet deployment descriptor. See the document *Developing SIP Applications*, a part of the documentation for Oracle Converged Application Server at http://download.oracle.com/docs/cd/E13153_01/wlcp/wlss40/programming/index.html

Table 4-3 Additional files generated for a SIP plug-in

File	Description
<Identifier>_<Protocol>_sip.war	Deployable SIP application.
application.xml	Deployment descriptor. Contains an additional element for elements for the SIP application.
<Identifier>Servlet.java	Implementation of a SIP Servlet.
<Identifier>SipHelper.java	Helper class for getting an instance of javax.servlet.sip.SipFactory and javax.servlet.sip.SipSessionsUtil.

Diameter Plug-in

Network protocol plug-ins can benefit from the Diameter support provided by Oracle Communications Converged Application Server.

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

Oracle Communications Converged Application Server includes an implementation of the base Diameter protocol that supports the core functionality and accounting features described in RFC 3588. Oracle Communications Converged Application Server uses the base Diameter functionality to implement multiple Diameter applications, including the Sh, Rf, and Ro applications.

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

Oracle Communications Services Gatekeeper uses the Diameter support provided by Oracle Communications Converged Application Server in the Parlay X 3.0 Payment Communication Service (Ro), CDR to Diameter service (Rf), and the Credit Control interceptor (Ro).

For an overview of the capabilities of the Diameter API provided with Oracle Communications Converged Application Server, see *Developing Diameter Applications* at http://download.oracle.com/docs/cd/E13153_01/wlcp/wlss40/diameter/. For information about the Diameter API, refer to the JavaDoc at http://download.oracle.com/docs/cd/E13153_01/wlcp/wlss40/javadoc/index.html. This is part of the documentation set for Oracle Communications Converged Application Server.

To create a plug-in that uses this the Diameter API, generate a network protocol plug-in using the Eclipse Wizard and include the JAR to the build path of the project.

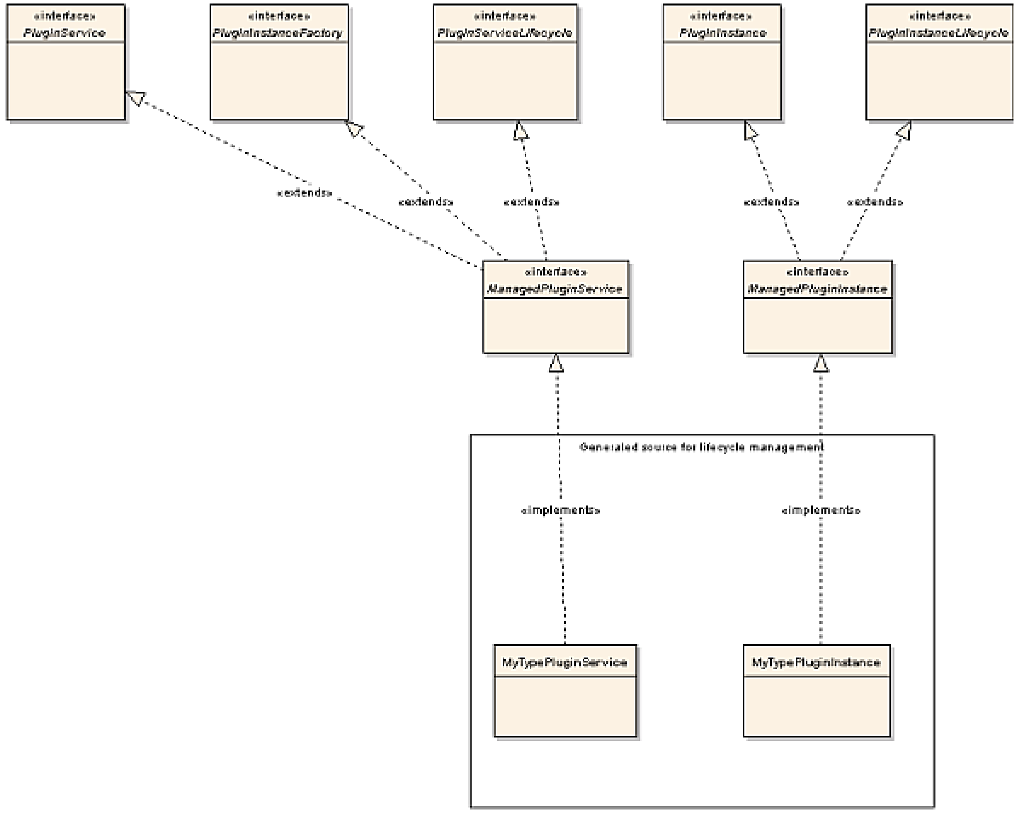
The Diameter API is packaged in `$OCCAS_HOME/server/lib/wlss/wlssdiameter.jar`.

The JAR file needs to be added to the build class path. It is already included in the run-time class path.

Generated classes for a Plug-in

The generated classes are listed below.

Figure 4-1 Example class diagram of the generated plug-in classes for life-cycle management and relationship with other interfaces



Interface: ManagedPluginService

The interface a plug-in service needs to implement.

It extends the interfaces PluginService, PluginInstanceFactory and PluginServiceLifecycle.

Interface: PluginService

The interface that defines the plug-in service when it is registered in the Plug-in Manager.

Interface: PluginInstanceFactory

The factory that allows a plug-in service to create plug-in instances.

Interface: PluginServiceLifecycle

The interface that defines the lifecycle for a plug-in service. See States.

PluginService

Class.

Implements com.bea.wlcp.wlng.api.plugin.ManagedPluginService.

Defines the life-cycle for a plug-in service, see States.

Also holds the data that is specific for the plug-in instance.

The actual class name is `<Communication Service Type>PluginService`. This class manages the life-cycle for the plug-in service, including implementing the necessary interfaces that make the plug-in deployable in Oracle Communications Services Gatekeeper. It is also responsible for registering the north interfaces with the Plug-in Manager. At startup time it uses the InstanceFactory to create one instance of each plug-in service and at activation time it registers these with the Plug-in Manager. The InstanceFactory uses an instancemap to find out which class it should instantiate for each plug-in interface implementation. The instance map is found under the `resource` directory.

ManagedPlugin Skeleton

The `ManagedPlugin` skeleton implements the following methods related to life-cycle management and should be adjusted for the plug-in:

- doStarted() - plug-in specific functionality for being started.
- doActivated() - plug-in specific functionality for being activated.
- doDeactivated() - plug-in specific functionality for being deactivated.
- doStopped() - plug-in specific functionality for being stopped.
- handleForceSuspending() - Called when a FORCE STOP/SHUTDOWN has been issued.
- handleResuming() - Transitions the plug-in from ADMIN to ACTIVE state in which it begins to accept traffic.
- handleSuspending(CompletionBarrier barrier) - Called in a normal re-deployment when the plug-in is taken from ACTIVE do ADMIN state.

- isActive() - reports back true or false. If false, no application-initiated requests are routed to the plug-in.

In addition, this class defines which address schemes the plug-in can handle, in `private static final String[] SUPPORTED_SCHEMES`.

PluginInstance

Class.

Implements com.bea.wlcp.wlng.api.plugin.ManagedPluginInstance.

Defines the life-cycle for a plug-in instance, see States.

The actual class name is `<Communication service Type>PluginInstance`. This class manages the life-cycle for the plug-in instance including implementing the necessary interfaces that make the plug-in an instance in Oracle Communications Services Gatekeeper.

It is also responsible for instantiating the classes that implement the traffic interfaces, and initializing stores to use and relevant MBeans.

See Interface: ManagedPluginInstance.

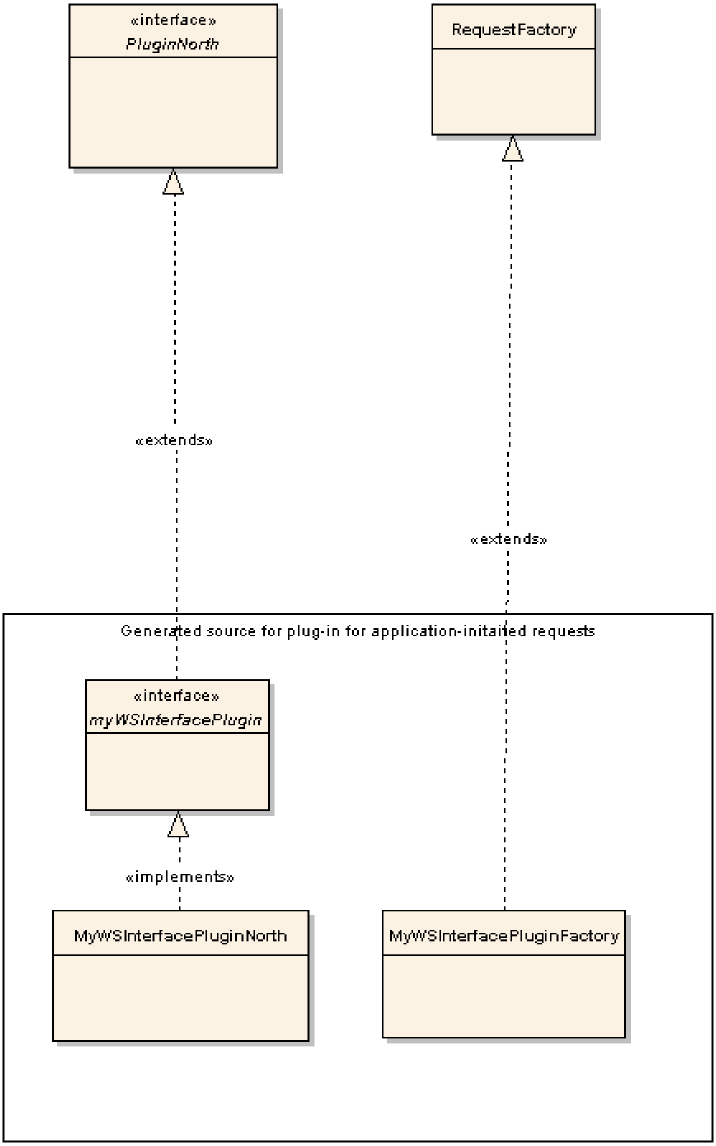
PluginNorth

This is an empty implementation of the Plug-in North interface. This interface is generated based on the WSDL files that define the application-facing interface. This is the starting point for the plug-in implementation.

The following files will be generated in the directory under `src/...../<service name>/north`:

- `<web service interface name>PluginNorth`: This class implements the plug-in interface. One file is generated for each plug-in interface. There is one plug-in interface for each service WSDL.

Figure 4-2 Class diagram of the generated PluginNorth and RequestFactory.



PluginNorth skeleton

Below outlines what needs to be implemented in the plug-in skeleton.

The class contains a Java mapping of the methods defined in the Web Service. The methods are mapped one-to-one. The name of each method is the same as the name of the operation defined in the WSDL. The parameter is a class that mirrors the parameters in the input message in the Web Service request. The return type is a class that represents the output message in the Web Service Request.

RequestFactory Skeleton

The actual class name is `<Communication service identifier>PluginFactory`, such as, for example, `NotificationManagerPluginFactory`. This is a helper class used by the Service EJB. It serves two purposes:

- It creates the routing information requested by the Plug-in Manager when routing the method call to a plug-in.
- It converts exceptions thrown either by the Plug-in Manager or by the plug-in to exception types that are supported by the application-facing interface. This is the place to convert exceptions specific to an extension plug-in to exceptions specific to the application-facing interface. It is a *best practice* to have one single place for performing these conversions in order to document and locate exception mappings.

The following files will be generated in the `dist` directory under `request_factory_skel/src`:

- `<webservice_interface_name>PluginFactory`: This class extends the `RequestFactory` class. There will be one file generated for each plug-in interface.

Generated classes for a SOAP2SOAP Plug-in

In addition to the generated classes for a regular plug-in, a SOAP2SOAP plug-in adds a few extra classes, because the network protocol is known.

Note: Only the deployable artifacts are relevant. The generated code for SOAP2SOAP type of plug-ins should not be modified.

See Managing and Configuring SOAP2SOAP Communication Services in the *System Administrator's Guide* for information on how to configure and manage a SOAP2SOAP plug-in

Comparison with a Non-SOAP2SOAP Plug-in

The following generated code is similar to the code generated for the non-SOAP2SOAP plug-ins:

- Interface: ManagedPluginService
- Interface: PluginService
- Interface: PluginInstanceFactory
- Interface: PluginServiceLifecycle
- ManagedPlugin Skeleton
- RequestFactory Skeleton

Client Stubs

These classes and interfaces are generated for each interface, based on the Service WSDLs:

- <Web Services Interface>_Stub
- <Web Services Interface>
- <Web Services Interface>Service_Impl
- <Web Services Interface>Service

<Web Services Interface>_Stub

Class.

Extends weblogic.wsee.jaxrpc.StubImp

Implements <Web Services Interface>

Used by the corresponding PluginNorth class.

<Web Services Interface>

Interface.

Extends java.rmi.Remote.

Implemented by <Web Services Interface>_Stub.

Defines the traffic methods.

<Web Services Interface>Service_Impl

Class.

Extends weblogic.wsee.jaxrpc.ServiceImpl.

Implements the Web Service.

<Web Services Interface>Service

Interface.

Extends javax.xml.rpc.Service.

Defines the traffic interfaces.

PluginInstance

In addition to the functionality in described in PluginInstance, in the PluginInstance generated for SOAP2SOAP plug-ins, the following occurs:

- In the implementation of activate() it:
  - instantiates and registers a class implementing com.bea.wlcp.wlng.httpproxy.management.HTTPProxyManagement
  - instantiates and registers a a class implementing com.bea.wlcp.wlng.heartbeat.management.HeartbeatManagement
- It unregisters the above in the implementation of deactivate().
- In the implementation of isConnected(), HeartbeatManagement is used to check the connection towards the network node.
- getHttpProxyManagement() is added for use by PluginSouth.

HTTPProxyManagement is described in Managing and Configuring SOAP2SOAP Communication Services in *Oracle Communications Services Gatekeeper System Administrator's Guide*.

HeartbeatManagement is described in Configuring Heartbeats in *Oracle Communications Services Gatekeeper System Administrator's Guide*.

PluginNorth

In addition to the functionality described in PluginNorth, this class:

- Checks whether there is an endpoint to the network node registered in the HttpProxyManagement MBean.
- Instantiates the client stubs used to make Web Services call to the network node: see Client Stubs.
- Invokes the corresponding method on the stubs.

PluginSouth

This class implements a Java representation of the Web Service implementation. It implements PluginSouth: see Interface: PluginSouth. When a network-triggered method is invoked, it:

- gets the handle to the callback EJB, see Class: CallbackFactory.
- Resolves the endpoint used for the application instance by querying the PluginInstance for the endpoint by calling getApplicationEndPoint(getApplicationInstanceId).
- Passes on the request to the callback EJB.

RequestFactory

The RequestFactory for a SOAP2SOAP plug-in has the same functionality as described in RequestFactory Skeleton, but instead of serving as a skeleton, it is an implementation. It contains an implementation of createRequestInfo(...) which means that the Plug-in Manager does no routing based on destination address.

Build Files and Targets for a Communication Service Project

Main Build File

The main build file for the Communication Service contains the following targets:

- `build_csc`, builds the common parts of the Communication Service .
- `build_plugins`, builds the plug-ins for the Communicaiton Service .
- `stage`, copies the JARs for the plug-ins to the directory `stage`.
- `make-facade`, creates a deployable EAR for the access tier in the directory `dist`.

- `make-enabler`, creates a deployable EAR for the network tier in the directory `dist`.
- `deploy-facade`, deploys the service facade EAR to the access tier.
- `undeploy-facade`, undeploys the service facade EAR from the access tier.
- `deploy-enabler`, deploys the service enabler EAR from the network tier.
- `undeploy-enabler`, undeploys the service enabler EAR from the network tier
- `clean`, clears the directory `dist.`
- `dist`, calls the `prepare,build_csc,build_plugins,stage,make-facade,make-enabler` targets.

Note: When using the deploy and undeploy targets, make sure to adapt the settings for user, password, adminurl, targets, and appversion in the parameters to wldeploy. By default Web Services Security is not enabled for new Communication Services. See section Setting up WS-Policy and JMX Policy in *System Administrator's Guide* for instructions on how to configure this.

Communication Service Common Build File

The build file for the common parts of the Communication Service contains the following targets:

- `dist`, Calls the csc_gen ant task that generates the Java source for each PluginFactory. The source is generated under the directory `dist/request_factory_skel/src`
- `clean:` Deletes the `dist` directory.

Plug-in Build File

The build file for the plug-in contains the following targets:

- `compile,` compiles the source code under the `src` directory and puts the class files under the `build` directory.
- `jar`, calls the `compile` target and then creates a plug-in jar file under the `dist` directory.
- `instrument,` weaves the aspects that should apply into the plug-in.
- `build.schema`, builds the schema file and the classes used by the storage service.
- `dist`, calls the `clean`, `compile`, `jar` and `instrument`, and `build.schema` targets.
- `clean`, deletes the `build` and `dist` directories.

Ant Tasks

The build files use a set of ant tasks and macros, described below:

- cs_gen
- plugin_gen
- cs_package
- javadoc2annotation

The ant tasks are defined in `$PDS_HOME/lib/wlng/ant-tasks.jar`

cs_gen

This ant task builds the common parts of the Communication Service. Below is a description of the attributes.

Table 4-4 cs_gen ant task

Attribute	Description
destDir	Defines the destination directory for the generated files.
packageName	Defines the package name to be used. Example: com.mycompany.service
serviceType	Defines the service type. Used in EDRs, statistics, etc. Example: SmsServiceType, MultimediaMessagingServiceType.
company	Defines the company name, to be used in META-INF/MANIFEST.MF.
version	Defines the version, to be used in META-INF/MANIFEST.MF.
contextPath	Defines the context path for the Web Service. Example: myService
soapAttachmentSupport	Use `true` if SOAP with attachments shall be supported. Use `false` if not.
wlngHome	Path to $OCSG_HOME, this depends on the installation. Example: `c:/bea/ocsg_4.1`

Table 4-4 cs_gen ant task

Attribute	Description
pdsHome	Path to $PDS_Home, this depends on the installation. Example `c:/bea/ocsg/wlng_pds400`

classpath	Defines the necessary classpaths. Must include:
	`$OCSG_HOME/server/lib/weblogic.jar`
	`$OCSG_HOME/server/lib/webservices.jar`
	`$OCSG_HOME/server/lib/api.jar`
	`$PDS_HOME/lib/wlng/wlng.jar`
	`$PDS_HOME/lib/log4j/log4j.jar`

servicewsdl	URL to the WSDL that defines the service.

Example:

```
<cs_gen destDir="${dist.dir}"
   packageName="com.bea.wlcp.wlng.example"
   name="say_hello"
   serviceType="example"
   company="BEA"
   version="4.1"
   contextPath="sayHello"
   soapAttachmentSupport="false"
   wlngHome="${wlng.home}"
   pdsHome="${pds.home}">
   <classpath refid="wls.classpath"/>
   <classpath refid="wlng.classpath"/>
   <servicewsdl file="${wsdl}/example_hello_say_service.wsdl"/>
</cs_gen>
```

plugin_gen

This ant task builds a plug-in for a Communication Service. Below is a description of the attributes.

Table 4-5 plugin_gen ant task

Attribute	Description
destDir	Defines the destination directory for the generated files.
packageName	Defines the package name to be used. Example: com.mycompany.service
name	Name and directory of the plug-in JAR.
serviceType	Defines the service type. Used in EDRs, statistics, etc. Example: SmsServiceType, MultimediaMessagingServiceType.
esPackageName	Communication Service package name used to import relevant classes.
protocol	An identifier for the network protocol the plug-in implements. Used as a part of the names of the generated jar file: `<Communication Service identifier>_<protocol>.jar` and the service name `Plugin_<Communication Service identifier>_<protocol>`.
schemes	Address schemes the plug-in can handle. Use a comma separated list if multiple schemes are supported. For example: `tel:` or `sip:`.
company	Defines the company name, to be used in META-INF/MANIFEST.MF.
version	Defines the version, to be used in META-INF/MANIFEST.MF.
pluginifjar	The name of the JAR file for the plug-in.

Table 4-5 plugin_gen ant task

Attribute	Description

classpath	Defines the necessary classpaths. Must include:
	`$OCSG_HOME/server/lib/weblogic.jar`
	`$OCSG_HOME/server/lib/webservices.jar`
	`$OCSG_HOME/server/lib/api.jar`
	`$PDS_HOME/lib/wlng/wlng.jar`
	`$PDS_HOME/lib/log4j/log4j.jar`

servicewsdl	URL to the WSDL that defines the service.

Example:

```
<plugin_gen destDir="${dist.dir}"
   packageName="com.bea.wlcp.wlng.example.bla"
   name="say_hello"
   serviceType="example"
   esPackageName="com.bea.wlcp.wlng.example"
   protocol="bla"
   schemes=""
   company="BEA"
   version="4.1"
   pluginifjar="${dist.dir}/say_hello/common/dist/say_hello_service.jar">
   <classpath refid="wls.classpath"/>
   <classpath refid="wlng.classpath"/>
   <servicewsdl file="${wsdl}/example_hello_say_service.wsdl"/>
</plugin_gen>
```

cs_package

This ant task packages a Communication Service. Below is a description of the attributes.

Table 4-6 cs_package ant task

Attribute	Description
destfile	Defines the destination directory for the generated files.
duplicate	Defines the package name to be used. Example: com.mycompany.service
displayname	Used in application.xml for the display name of the application.
descriptorfileset	Defines the service type. Used in EDRs, statistics, etc. Example: SmsServiceType, MultimediaMessagingServiceType.
manifest	Description of the manifest file use. Enter values for the following attributes: name="Bundle-Name" value should be the name of the EAR for the service enabler. name="Bundle-Version" value should be the version to use. name="Bundle-Vendor" value should be vendor name name="Weblogic-Application-Version" value should be the version of the EAR
fileset	Should point to the Communication Service JAR.
zipfileset	Should point to the plug-in JAR(s).

Example:

```
<cs_package destfile="${cs.dist}/${enabler.ear.name}.ear"
   duplicate ="fail"
   displayname="${enabler.ear.name}">
   <descriptorfileset dir="${csc.dir}/resources/enabler/META-INF"
    includes="*.xml"/>
  <descriptorfileset dir="${cs.name}/plugins"
    includes="*/resources/META-INF/*.xml"/>
   <manifest>
```

```
      <attribute name="Bundle-Name"
         value="${enabler.ear.name}"/>
      <attribute name="Bundle-Version"
         value="${manifest.bundle.version}"/>
      <attribute name="Bundle-Vendor"
         value="${manifest.bundle.vendor}"/>
      <attribute name="Weblogic-Application-Version"
         value="${manifest.bundle.version}"/>
   </manifest>
   <fileset dir="${csc.dist}">
      <include name="*_service.jar"/>
   </fileset>
   <zipfileset dir="${cs.stage}">
      <include name="*plugin.jar"/>
   </zipfileset>
</cs_package>
```

javadoc2annotation

This ant macro annotates an MBean interface based on the JavaDoc. The macro is defined in the common.xml build file for the

The annotations are rendered as descriptive information by the Gatekeeper Administration console. Below is a description of the attributes.

Table 4-7 javadoc2annotation ant macro

Attribute	Description
tempDir	Temporary directory for the generated files.
destDir	Destination directory for the generated MBean interface.

Table 4-7 javadoc2annotation ant macro

Attribute	Description
sourceDir	Source directory for the MBean interface with JavaDoc annotations.

classpath	Defines the necessary classpaths. Depending on which interfaces that are used from the MBean, include:
	`$OCSG_HOME/server/lib/weblogic.jar`
	`$OCSG_HOME/server/lib/webservices.jar`
	`$OCSG_HOME/server/lib/api.jar`
	`$PDS_HOME/lib/wlng/wlng.jar`
	`$PDS_HOME/lib/log4j/log4j.jar`

Example:

```
<javadoc2annotation
   tempDir="${plugin.generated.dir}/mbean_gen_tmpdir"
   destDir="${plugin.classes.dir}"
   sourceDir="${plugin.src.dir}"
   classpath="javadoc.classpath">
</javadoc2annotation>
```

Communication Service Example

This section describes the example Communication Service in the Platform Development Studio:

Overview

The Communication service example demonstrates the following:

- Structure and execution workflow in a Communication Service.
- Parameter validation

- Hitless upgrade
- Retry
- Simple TCP/IP protocol-based simulator
- Testability with the PTE

The example is based on an end-to-end Communication Service, with a set of simple interfaces

- SendData, which defines the operation `sendData` used to send data to a given address.
- NotificationManager, which defines these operations:
  - `startEventNotification`, which starts a subscription for network-triggered events.
  - `stopEventNotification`, which ends the subscription for network-triggered events.
- Notification, which defines the operation:
  - `notifyDataReception`, used to notify the application on a network-triggered event.

The SendData and NotificationManager interfaces are used by an application and implemented by the Communication Service.

The Notification interface is used by the Communication Service and implemented by an application.

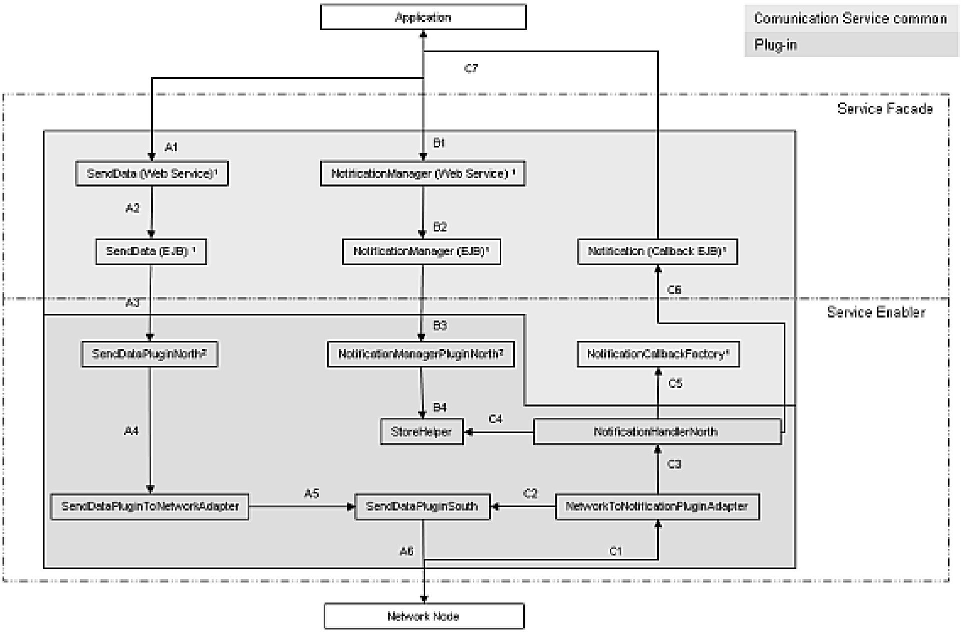
The Communication Service to network node interface is a simple TCP/IP based interface that defines the two commands:

- `sendDataToNetwork`, that sends data to the network node.
- `receiveData`, that is used by the network node to send data to a receiver - in this case the network protocol plug-in.

Figure 5-1 illustrates the flow for these operations.

Figure 5-1 Overview of example Communication Service

g


Application
Comunication Service common
Plug-in
C7
Service Facade
A1
B1
SendData (Web Service)¹
NotificationManager (Web Service) ¹
A2
B2
SendData (EJB) ¹
NotificationManager (EJB)¹
Notification (Callback EJB)¹
C6
A3
Service Enabler
B3
SendDataPluginNorth²
NotificationManagerPluginNorth²
NotificationCallbackFactory¹
C5
B4
C4
A4
StoreHelper
NotificationHandlerNorth
C3
A5
C2
SendDataPluginToNetworkAdapter
SendDataPluginSouth
NetworkToNotificationPluginAdapter
A6
C1
Network Node

The flow marked A* is for `sendData`, the flow marked B* is for `startNotification` and `stopNotification`, and the flow marked C* is for `notifyDataReception`.

The modules marked with 1 are automatically generated based on the WSDL files that define the application-facing interface and code generation templates provided by the Platform Development Studio. The modules marked with 2 are skeletons generated at build time.

High-level Flow for sendData (Flow A)

1. A1: An application invokes the Web Service SendData, with the operation `sendData`.
2. A2: The request is passed on the EJB for the interface, which passes it on to the network protocol plug-in. The diagram is simplified, but at this stage the Plug-in Manager is invoked and makes a routing decision to route to the appropriate plug-in.

3. A3: The Plug-in Manager invokes the `sendData` method in the class SendDataPluginNorth. It will always invoke a class named PluginNorth, that has a prefix that is the same as the Java representation of the Web Service interface.
4. A4: The request is passed on to class SendDataPluginToNetworkAdapter that performs the protocol translation according to the network-interface.
5. A5: The request is passed to SendDataPluginSouth.
6. A6: The request is handed off to the network node.

High-level Flow for startNotification and stopNotification (Flow B)

The initial steps (B1-B3) are similar to flow A*. Instead of translating the request to a command on the network node, NotificationManagerNorth uses the StoreHelper to either store a new or remove a previously registered subscription for notifications. The data stored, the NotificationData, is used in network-triggered requests to resolve which application started the notification and the destination to which to send it. In the example the notification is started on an address, so the address is stored together with information to which endpoint the application wants the notification to be sent.

High-level flow for notifyDataReception (Flow C)

1. C1: The network protocol plug-in receives the network-triggered command `receiveData` on NetworkToNotificationPluginAdapter.
2. C2: SendDataPluginSouth can be used to add additional information to the request before passing in on.
3. C3: NetworkToNotificationPluginAdapter performs the protocol translation.
4. C4: StoreHelper is used to examine if the request matches any stored NotificationData. If so, the information in NotificationData is retrieved. This information includes which application instance that the request resolves to and on which endpoint this application wants to be notified about the network triggered event.
5. C5: NotificationCallbackFactory is used to get a hold of an active NotificationCallback EJB to pass on the request to.
6. C6: The request is passed on to the NotificationCallback EJB.
7. C7: The request is passed on to an application.

Interfaces

The example Communication Service translates between an application-facing interface, defined in WSDL, see Web Service Interface Definition and a network interface, TCP/IP based, see Network Interface Definition.

Web Service Interface Definition

Interface: SendData

This interface is a simple interface containing operations for sending data.

Operation: sendData

Send data to the network.

Input message: sendDataMessage

Part name	Part type	Optional	Description
data	xsd:string	N	The data to be sent to the target device
address	xsd:anyURI	N	Address of the target device. Example: tel:4154011234

Output message: sendDataResponse

Part name	Part type	Optional	Description
none			

Interface: NotificationManager

The Notification Manager Web Service is a simple interface containing operations for managing subscriptions to network triggered events.

Operation: startEventNotification

Start the subscription of event notification from the network.

Input message: startEventNotificationRequest

Part name	Part type	Optional	Description
correlator	xsd:string	N	Service unique identifier provided to set up this notification.
endPoint	xsd:string	N	Endpoint address. Endpoint of the application to receive notifications. Example: http://www.hostname.com/NotificationService/services/Notification
address	xsd:anyUR	N	Service activation number. Example: tel:4154567890

Output message: invokeMessageResponse

Part name	Part type	Optional	Description
none			

Operation: stopEventNotification

Stop the subscription of event notification from the network.

Input message: stopEventNotificationRequest

Part name	Part type	Optional	Description
correlator	xsd:string	N	Service unique identifier provided to set up this notification.

Output message: stopEventNotificationResponse

Part name	Part type	Optional	Description
none			

Interface: NotificationListener

The NotificationListener interface defines the methods that the Communication Service invokes on a Web Service that is implemented by an application.

Operation: notifyDataReception

Method used for receiving a notification.

Input message: notifyDataReceptionRequest

Part name	Part type	Optional	Description
correlator	xsd:string	N	Service unique identifier provided to set up this notification.
originatingAddress	xsd:anyURI	N	Address of the device where the data originated. Example: tel:4153083412
data	xsd:string		Data sent by the originating device.

Output message: notifyDataReceptionResponse

Part name	Part type	Optional	Description
none			

Network Interface Definition

sendDataToNetwork

Send data from the Communication Service to the network node.

Argument	Type	Description
fromAddress	String	The address from which the request is sent.
toAddress	String	The address to which the request shall be sent.
data	String	The data to send.

receiveData

Send data from the network node to the Communication Service.

Argument	Type	Description
fromAddress	String	The address from which the request is sent.
toAddress	String	The address to which the request shall be sent.
data	String	The data to send.

Directory Structure

Below is a description of the directory structure for the example Communication Service.

```
communication_service
+- build.properties
+- common.xml
+- build.xml
+- example
| +- common
| | +- build.xml
| | +- dist
| | | +- request_factory_skel
| | | +- tmp
| | | +- example.war
| | | +- example_callback.jar
| | | +- example_callback_client.jar
| | | +- example_service.jar
| | | +- resources
| | | | +- enabler
| | | | + facade
| | | +- src
| | | | +- com/<package name>Plugin
| | | | | +- ExceptionType.java
| | | | | +- NotificationManagerPluginFactory.java
| | | | | +- SendDataPluginFactory.java
| | | | | +- handlerconfig.xml
| | | | | +- weblogic.xml
| | +- wsdl
| +- dist
| | +- com.acompany.plugin.example.netex.store_4.1.jar
| | +- example_enabler.ear
| | +- example_facade.ear
| +- plugins
| | +- nextex
| | | +- build.xml
| | | +- dist
| | | | +- example_netex_plugin.jar
| | | | +- com.acompany.plugin.example.nextex.store_4.1.0.0.jar
| | | +- build
| | | +- config
| | | | +- edr
| | | | | +- alarm.xml
| | | | | +- cdr.xml
| | | | | +- edr.xml
| | | | | +- alarm.xml
| | | | +- instance_factory
| | | | | +- instancemap
| | | +- dist
| | | | +- com.acompany.plugin.example.netex.store_4.1.jar
| | | | +- example_netex_plugin.jar
| | | +- src/com/acompany/plugin/example/netex/
| | | |           +- context
| | | |           +- management
| | | |           +- notification
| | | |           +- notificationmanager
| | | |          +- senddata
| | | |          +- store
| | | +- storage
| | | | +- wlng-cachestore-config-extensions.xml
```

Directories for WSDL

Below is a list of WSDL files that define the application-facing interface and the Java representation of these in the plug-in.

Application-initiated traffic

$PDS_HOME/example/communication_service/example/common/wsdl/service

```
example_common_faults.wsdl
example_common_types.xsd
example_data_send_interface.wsdl
example_data_send_service.wsdl
example_notification_manager_interface.wsdl
example_notification_manager_service.wsdl
```

Network-triggered traffic

$PDS_HOME/example/communication_service/example/common/wsdl/callback

```
example_notification_interface.wsdl
example_notification_service.wsdl
```

Directories for Java Source

Below is a list of Java source directories for the Communication Service Common and the Plug-in.

Communication Service Common

$PDS_HOME/example/communication_service/example/common/src

com.acompany.example.plugin.ExceptionType

com.acompany.example.plugin.NotificationManagerPluginFactory

com.acompany.example.plugin.SendDataPluginFactory

Plug-in

$PDS_HOME/example/communication_service/example/plugins/netex/src

com.acompany.plugin.example.netex.context.ContextTranslatorImpl

com.acompany.plugin.example.netex.management.ConfigurationStoreHandler

com.acompany.plugin.example.netex.management.ExampleMBean

com.acompany.plugin.example.netex.management.ExampleMBeanImpl

com.acompany.plugin.example.netex.management.Management

com.acompany.plugin.example.netex.notification.north.NotificationHandlerNorth

com.acompany.plugin.example.netex.notification.south.NetworkToNotificationPluginAdapter

com.acompany.plugin.example.netex.notification.south.NetworkToNotificationPluginAdapterImpl

com.acompany.plugin.example.netex.notificationmanager.north.NotificationManagerPluginNorth

com.acompany.plugin.example.netex.senddata.north.SendDataPluginNorth

com.acompany.plugin.example.netex.senddata.south.SendDataPluginSouth

com.acompany.plugin.example.netex.senddata.south.SendDataPluginToNetworkAdapter

com.acompany.plugin.example.netex.senddata.south.SendDataPluginToNetworkAdapterImpl

com.acompany.plugin.example.netex.store.FilterImpl

com.acompany.plugin.example.netex.store.NotificationData

```
com.acompany.plugin.example.netex.store.StoreHelper
```

```
com.acompany.plugin.example.netex.ExamplePluginInstance
```

```
com.acompany.plugin.example.netex.ExamplePluginService
```

Directories for resources

Only the Communication Service common components have associated resources. The resources are XML files that serve as deployment descriptors for the network tier EAR and the access tier EAR.

$PDS_HOME/example/communication_service/example/common/resources/at/META-INF

Contains deployment descriptors for the access tier EAR file. These must be present in the META-INF directory of the EAR. *See Oracle WebLogic Server Developing Applications with WebLogic Server* at http://download.oracle.com/docs/cd/E12840_01/wls/docs103/programming for a description of the enterprise application deployment descriptor elements.

```
application.xml
```

```
weblogic-application.xml
```

The code generation creates these files, and the build script takes care of the packaging.

$PDS_HOME/example/communication_service/example/common/resources/nt/META-INF

Contains deployment descriptors for the network tier EAR file. These must be present in the META-INF directory of the EAR. *See Oracle WebLogic Server Developing Applications with WebLogic Server* at http://download.oracle.com/docs/cd/E12840_01/wls/docs103/programming for a description of the enterprise application deployment descriptor elements.

```
application.xml
```

```
weblogic-application.xml
```

```
weblogic-extension.xml
```

The code generation creates these files, and the build script takes care of the packaging.

Directories for Configuration of Plug-in

$PDS_HOME/example/communication_service/example/plugins/netex/config/edr

Sample entries to add in the EDR, CDR, and Alarm filters.

`alarm.xml`

`cdr.xml`

`edr.xml`

These serves as examples. Add the contents of these to the EDR configuration file. Use the **EDR Configuration Pane** as described in Managing and Configuring EDRs, CDRs and Alarms in the *System Administrator's Guide.*

$PDS_HOME/example/communication_service/example/plugins/netex/instance_factory

Sample instance map for mapping of classes, interfaces, and abstract classes.

When using com.bea.wlcp.wlng.api.util.InstanceFactory to retrieve instances for a given interface, class, or abstract class, this mapping is referenced. The mapping can be overridden. See JavaDoc for InstanceFactory for details.

`instancemap`

$PDS_HOME/example/communication_service/example/plugins/netex/storage

Sample store configuration file. Defines how the Storage service is used by the plug-in, store type, table names, query definitions, and get and set methods. See StoreHelper, FilterImpl, and NotificationData.

`wlng-cachestore-config-extensions.xml`

Directories for Build and Configuration of Builds

$PDS_HOME/example/communication_service/

`build.properties`

Defines the installation directory for Oracle Communications Services Gatekeeper and for the Platform Development Studio.

`common.xml`

Defines properties, class paths, task definitions, and macros for the build.

`build.xml`

Main build file to build the Communication Service. This build file also contains targets for packaging deployable artifacts into the access and network tier.

$PDS_HOME/example/communication_service/example/common

`build.xml`

Build file for the common parts of the Communication Service.

$PDS_HOME/example/communication_service/example/plugins/netex

`build.xml`

Build file for the plug-in.

Directories for Classes, JAR, and EAR Files

$PDS_HOME/example/communication_service/example/dist

Deployment artefacts for the Communication Service.

`example_facade.ear`

The part of the Communication Service that is deployed in the access tier.

`example_enabler.ear`

The part of the Communications Service that is deployed in the network tier.

$PDS_HOME/example/communication_service/example/common/dist

JAR and WAR files for the common parts of the Communication Service.

`example_callback_client.jar`

`example_callback.jar`

`example_service.jar`

`example.war`

$PDS_HOME/example/communication_service/example/common/dist/request_factory_skel

Auto generated source for skeleton classes extending com.bea.wlcp.wlng.api.plugin.RequestFactory.

One class is generated per Service WSDL, that is per interface that defines application-initiated operations.

The classes are named <PreFix>PluginFactory, where <PreFix> is picked up from the WSDL binding in the WSDL file.

In the subdirectory that corresponds to the package name, the following classes are generated:

`NotificationManagerPluginFactory.java`

`SendDataPluginFactory.java`

These are generated as skeletons, but in the example they are adapted to the specific use cases.

$PDS_HOME/example/communication_service/example/plugins/netex/dist

Contains individual JAR files comprises the plug-in.

`com.acompany.plugin.example.netex.store_4.1.jar`

Includes the schema file for the store used by the plug-in, packaged together with the classes for which instances are stored. This file must be put in `$DOMAIN_HOME/config/store_schema` on each server in the network tier. The server needs to be restarted if any changes have been done to the store schema or the classes referred to in the store schema.

`example_netex_plugin.jar`

The JAR for the plug-in.

$PDS_HOME/example/communication_service/example/plugins/netex/dist/mbean_generationdir

Output directory for the MBean that has been processed by the javadoc2annotation ant task.

Classes

Below is a description of the classes and the methods defined in these classes:

- Communication Service Common
  - ExceptionType
  - NotificationManagerPluginFactory
  - SendDataPluginFactory
- Plug-in Layer

- ContextTranslatorImpl
- ExamplePluginService
- ConfigurationStoreHandler
- ExampleMBean
- ExampleMBeanImpl
- Management
- NotificationHandlerNorth
- NetworkToNotificationPluginAdapter
- NetworkToNotificationPluginAdapterImpl
- NotificationManagerPluginNorth
- SendDataPluginNorth
- SendDataPluginSouth
- SendDataPluginToNetworkAdapter
- SendDataPluginToNetworkAdapterImpl
- FilterImpl
- NotificationData
- StoreHelper

Communication Service Common

ExceptionType

Class.

Enumeration for exception types:

Defines:

- SERVICE_ERROR
- POLICY_ERROR

NotificationManagerPluginFactory

Class.

Extends RequestFactory.

Helper class that is used by the service EJB for two purposes:

- Creating routing information requested by the Plug-in Manager when routing the method call to a plug-in.
- Converting Exceptions, thrown either by the Plug-in Manager or by the plug-in, to Exceptions that are supported by the application-facing interface.

Note: This class needs to remain in this package and the class name must not be changed.

public void validateRequest(Method method, Object... args)

Validates the request to make sure that mandatory parameters are present. Operates on a Java representation of the Web Service call.

public RequestInfo createRequestInfo(Class<? extends Plugin> type, Method method, Object... args)

Used by the service EJB to extract routing data from the method call. The routing data is then given to the Plug-in Manager. This method returns the routing data in a RequestInfo object.

Returns a:

- AddressRequestInfo if the request contains an actual address that can be routed to a specific plug-in.
- CorrelatorRequestInfo if the request contains an correlator that relates to an operation that relates to states (to start or to stop something). Most often it is the starting and stopping of notifications that use a correlator.

public Throwable convertEx(Method method, Throwable e)

Called by the service EJB in order to convert Exceptions thrown by the Plug-in Manager and the Plug-in to Exceptions defined by the called method.

private Throwable convertEx(Method method, PluginException e)

Converts a PluginException to an Exception that can be thrown by the method called by the application.

Plug-in Layer

ContextTranslatorImpl

Class.

Implements interface com.bea.wlcp.wlng.api.plugin.context.ContextTranslator.

Responsible for setting any non-simple parameter into the RequestContext.

public void translate(Object param, ContextInfo info)

Puts the member variables of a complex data type into the ContextInfo.

Checks the interface type.

Gets the simple data types provided in the parameter param.

Puts each of the parameters into the ContextInfo object.

These parameters are provided in each subsequent EDR that is emitted in the request.

ExamplePluginService

Package: com.acompany.plugin.example.netex

Implements ManagedPluginService.

Initial point for the network protocol plug-in.

Defines the life-cycle for a plug-in service.

Also holds the data that is specific for the plug-in instance.

This class manages the life-cycle for the plug-in service, including implementing the necessary interfaces that make the plug-in deployable in Oracle Communications Services Gatekeeper. It is also responsible for registering the north interfaces with the Plug-in Manager. At startup time it uses the InstanceFactory to create one instance of each plug-in service and at activation time it registers these with the Plug-in Manager. The InstanceFactory uses an instancemap to find out which class it should instantiate for each plug-in interface implementation. The instance map is found under the resource directory. It also has

public boolean isRunning()

Checks to see if the plug-in service is in running state.

public String[] getSupportedSchemes()

Returns a list of address schemes the plug-in supports.

public void init(String id, PluginPool pool)

Initializes the plug-in service with its ID and a reference to its plug-in pool.

public void doStarted()

When entering state Started, the plug-in instantiates a TimerManager.

public void doStopped()

No action.

public void doActivated()

No action.

public void doDeactivated()

No action.

public void handleSuspending(CompletionBarrier barrier)

The plug-in service does not handle graceful shutdown: it propagates the request to public void handleForceSuspending().

public void handleForceSuspending()

When the plug-in is being forcefully suspended, the plug-in service iterates through all plug-in instances and calls public void handleSuspending() on each.

public boolean isActive()

While there is a connection to the network node and the plug-in is in state ACTIVE/RUNNING this method must return true, in all other cases false. This method is invoked by the Plug-in Manager during route selection.

public ServiceType getServiceType()

Returns the type of the service. Used by the Plug-in Manager to route requests to a plug-in instance that can manage the type of request. The ServiceType is auto-generated based on the WSDL that defines the application-facing interfaces.

public String getNetworkProtocol()

Returns a descriptive name of the network protocol being used.

createInstance(String)

Creates a new plug-in instance.

ExamplePluginInstance

Package: com.acompany.plugin.example.netex.

Implements ManagedPluginInstance

Defines the life-cycle for a plug-in instance/

This class manages the life-cycle for the plug-in instance including implementing the necessary interfaces that make the plug-in an instance in Oracle Communications Services Gatekeeper.

It is also responsible for instantiating classes that implement the traffic interfaces and for initializing stores to use and MBeans.

public String getId()

Returns the plug-in instance ID.

public void activate()

- Instantiates the classes implementing the PluginNorth interface:
  - SendDataPluginNorth
  - NotificationManagerPluginNorth
  - NotificationHandlerNorth
- Instantiates the class implementing the PluginSouth interface:
  - SendDataPluginSouth
- Instantiates the classes that implements the southbound and northbound adapter instances:
  - NetworkToNotificationPluginAdapterImpl
  - SendDataPluginToNetworkAdapterImpl
- Creates the network proxy:
- Registers the PluginNorth interfaces into the Plug-in Manager.

- Registers the PluginSouth interfaces into the Plug-in Manager.
- Registers the NetworkToNotificationPluginAdapter into the network proxy to be notified when a request arrives from the network node.
- Sets NotificationHandlerNorth to NetworkToNotificationPluginAdapter in order to forward request to the application.
- Sets the network proxy into the SendDataPluginToNetworkAdapter in order to send request to the network.
- Sets SendDataPluginToNetworkAdapter into SendDataPluginNorth.
- Instantiates ConfigurationStoreHandler.
- Instantiates Management and registers the plug-in into it.

private void rethrowServiceDeploymentException(Exception e)

Re-throws a ServiceDeploymentException if any other exception is encountered. The exception is wrapped in a ServiceDeploymentException.

public ConfigurationStoreHandler getConfigurationStore()

Returns a handle to the ConfigurationStore used by the plug-in instance. The ConfigurationStore was initiated in public void activate().

public NetworkProxy getNetworkProxy()

Returns handle to the NetworkProxy. The NetworkProxy was initiated in public void activate().

public void connect()

Connects to the network using NetworkProxy.

ConnectTimerTask

Inner class of ExamplePluginService.

Extends java.util.TimerTask.

It has one method, run(), that tries to connect to the network node, if not connected. This class is instantiated and scheduled as a java.util.Timer in public void handleResuming().

ConfigurationStoreHandler

Handles storage of configuration data using the StorageService.

A set of default settings are defined as static final variables. These are used to populate the ConfigurationStore with default values the first time the plug-in is deployed.

Takes the plug-in ID as a parameter. The plug-in ID is the key in the ConfigurationStore.

Uses ConfigurationStoreFactory to get a handle to the ConfigurationStoreService and gets the local ConfigurationStore that handles configuration data for the plug-in instance.

The plug-in only deals with configuration data that is unique for the instance in a specific server, so the store is fetched as outlined in Listing 5-1.

Listing 5-1 Get a server-specific (local) ConfigurationStore

```
ConfigurationStoreFactory factory = ConfigurationStoreFactory.getInstance();

localConfigStore = factory.getStore(pluginId, LOCAL_STORE,
ConfigurationStore.STORE_TYPE_LOCAL);
```

If the plug-in uses a ConfigurationStore that is shared between the plug-in instances in the cluster, it must fetch that one as well, as outlined in Listing 5-2

Listing 5-2 Get a cluster-wide (shared) ConfigurationStore

```
ConfigurationStoreFactory factory = ConfigurationStoreFactory.getInstance();

sharedConfigStore = factory.getStore(pluginId, SHARED_STORE,
ConfigurationStore.ConfigurationStore.STORE_TYPE_SHARED);
```

After the ConfigurationStore is fetched, it is initialized with default values for the available configuration settings. These default values can be changed later on, using the MBeans, see ExampleMBean.

public void setLocalInteger(String key, Integer value),

public Integer getLocalInteger(String key),

public void setLocalString(String key, String value), and

public String getLocalString(String key)

The methods above are used to set and get data to and from the ConfigurationStore. One set/get pair must be implemented per data type in the ConfigurationStore. It is only necessary to implement set/get methods for the data types actually used by the plug-in.

In the set methods, the parameter name/key is provided as the first parameter and the actual value is provided in the second parameter.

In the get methods, the parameter name/key is provided as the parameter and the actual value is returned.

ExampleMBean

Interface.

Management interface for the example simulator.

It defines the following methods:

- public void setNetworkPort(int port) throws ManagementException;
- public int getNetworkPort() throws ManagementException;
- public void connect() throws ManagementException;
- public void disconnect() throws ManagementException;
- public boolean connected();

Implemented by ExampleMBeanImpl.

All MBean methods should throw com.bea.wlcp.wlng.api.management.ManagementException or a subclass thereof if the management operation fails.

Management

Class.

Handles registration of the ExampleMBean in the MBean Server.

NotificationHandlerNorth

NotificationHandlerNorth()

Constructor.

Empty.

public void deliver(String data, String destinationAddress, String originatingAddress)

Delivers data originating from the network node to the application.

NetworkToNotificationPluginAdapterImpl calls this method upon a network triggered request.

The actual delivery is not done directly to the application. Instead it is done via the service callback client EJB which forwards the request to the service callback EJB. Both of these are generated during the build process.

First, the NotificationData associated with the destination address is fetched.

NotificationCallback, which is a generated class, is fetched using private NotificationCallback getNotificationCallback().

NotifyDataReception, a generated class that is a Java representation of the operation defined in the callback WDSL is instantiated.

The correlator associated with the NotificationData is set on NotifyDataReception.

The data (payload) in the network triggered request is set on NotifyDataReception.

The originating address in the network-triggered request is converted to a URI and set on NotifyDataReception.

The endpoint associated with NotificationData is fetched.

A remote call is done to the method notifyDataReception on the Callback EJB in the access tier. The endpoint and NotifyDataReception are supplied as parameters.

private NotificationCallback getNotificationCallback()

Helper method to get the object representing the Callback EJB.

If the object is already retrieved it is returned, otherwise the NotificationCallbackFactory is used to get a new object. This is the preferred pattern.

Using the CallBackFactory ensures high-availability between the network tier and the access tier for network triggered requests.

The Callback is generated during the build process when the access tier is generated. Three files are generated per callback WSDL. The names are based on the interface name defined in the WSDL. The interface in the WSDL is *Notification*, so:

- the factory is named *Notification*CallbackFactory.
- the implementation class is named *Notification*CallbackImpl
- an interface is named is named *Notification*Callback.

The classes are completely based on the WSDL file for the callback interface. The factory is used to retrieve the implementation class that implements the interface.

private NotificationData getNotificationData(String destinationAddress)

Helper method to fetch the NotificationData from the StoreHelper. The NotificationData is retrieved based on the key destination address.

NetworkToNotificationPluginAdapter

Interface

extends PluginSouth, NetworkCallback

Defines the interface between NetworkToNotificationPluginAdapter and the network node.

public void setNotificationHandler(NotificationHandlerNorth notificationHandlerNorth)

Sets the NotificationHandler.

NetworkToNotificationPluginAdapterImpl

Class.

Implements NetworkToNotificationPluginAdapter.

public void setNotificationHandler(NotificationHandlerNorth notificationHandlerNorth)

Sets NotificationHandlerNorth in the class.

public String resolveAppInstanceGroupdId(ContextMapperInfo info)

From interface com.bea.wlcp.wlng.api.plugin.PluginSouth

Gives the plug-in an opportunity to add additional values to the RequestContext before the network-triggered requests is passed on to public void receiveData(@ContextKey(EdrConst ants.FIELD_ORIGINATING_ADDRESS) String fromAddress, @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS) @MapperInfo(C) String toAddress, String data).

This method is called only once per network-triggered request. It is invoked after `resolveAppInstanceGroupId(ContextMapperInfo)`, when the RequestContext for the current request has been rebuilt.

The default implementation is supposed to be empty.

RequestContext contains the fully rebuilt RequestContext.

ContextMapperInfo contains the annotated parameters in public void receiveData(@ContextKey(EdrConst ants.FIELD_ORIGINATING_ADDRESS) String fromAddress, @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS) @MapperInfo(C) String toAddress, String data).

public void receiveData(@ContextKey(EdrConst ants.FIELD_ORIGINATING_ADDRESS) String fromAddress, @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS) @MapperInfo(C) String toAddress, String data)

From NetworkCallback.

The network node invokes this method when a network-triggered events occurs.

The parameter:

- fromAddress is the address representing the originator of the request
- toAddress is the address representing the destination of the request.
- data contains the payload of the request.

The method is annotated with @Edr, so the method is woven with annotation EDR.

fromAddress and toAddress are annotated with @ContextKey, which means that they will be put it the current RequestContext under the key specified by the string in the argument of the annotation. As illustrated in Listing 5-3, they are put in the RequestContext under the keys EdrConstants.FIELD_ORIGINATING_ADDRESS and EdrConstants.FIELD_DESTINATION_ADDRESS, respectively. These keys ensure that the values will be available in all subsequent EDRs emitted during this request.

toAddress is also annotated with @MapperInfo, which means that the value should be registered in ContextMapperInfo under the key specified by the string in the argument of the annotation. In Listing 5-3, the key is C.

Listing 5-3 Annotation of network-triggered method

```
...
@Edr
public void receiveData(
   @ContextKey(EdrConstants.FIELD_ORIGINATING_ADDRESS)
   String fromAddress,
   @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS)
   @MapperInfo(C)
   String toAddress,
  String data) {
...
```

NotificationManagerPluginNorth

Class.

Implements NotificationManagerPlugin.

public StartEventNotificationResponse startEventNotification(@ContextTranslate(ContextTranslatorImpl.class) StartEventNotification parameters)

Starts a subscription for notifications on network-triggered requests.

The method is a Java representation of the application-facing operation startEventNotification, defined in the WSDL that was used as input for the code generation.

As illustrated in Listing 5-4, the method is annotated with @EDR, and the parameter is put in the RequestContext using the annotation @ContextTranslate, since the parameter is a complex data type that requires traversal in order to resolve the simple data types. When using this annotation, the class is provided as an ID.

Listing 5-4 Annotations for startEventNotification

```
...

@Edr

public StartEventNotificationResponse startEventNotification(

@ContextTranslate(ContextTranslatorImpl.class) StartEventNotification
parameters)

throws ServiceException {

...
```

In the operation, these parameters are included:

```
<xsd:element name="correlator" type="xsd:string"/>

<xsd:element name="endPoint" type="xsd:string"/>

<xsd:element name="address" type="xsd:anyURI"/>
```

The values of correlator and endPoint are put in NotificationData.

The application instance ID for the originator of the request, the application that uses the Web Services interface, is resolved from the RequestContextManager and put in NotificationData.

Using StoreHelper, NotificationData is put in the StorageService.

public StopEventNotificationResponse stopEventNotification(@ContextTranslate(ContextTranslatorImpl.class) StopEventNotification parameters)stopEventNotification(StopEventNotification)

Ends a previously started subscription for notifications on network-triggered requests.

The method is a Java representation of the application-facing operation `stoptEventNotification`, defined in the WSDL that was used as input for the code generation.

The method is annotated in a similar manner to public StartEventNotificationResponse startEventNotification(@ContextTranslate(ContextTranslatorImpl.class) StartEventNotification parameters).

Using StoreHelper, NotificationData corresponding to the correlator provided in the requests is removed from the StorageService.

SendDataPluginNorth

Class.

Implements SendDataPlugin.

public void setPluginToNetworkAdapter(SendDataPluginToNetworkAdapter adapter)

Sets SendDataPluginToNetworkAdapter to be used for application-initiated requests.

public SendDataResponse sendData(@ContextTranslate(ContextTranslatorImpl.class) SendData parameters)

Sends data to the network

The method is a Java representation of the application-facing operation `sendData`, defined in the WSDL that was used as input for the code generation.

The method is annotated in a similar manner to public StartEventNotificationResponse startEventNotification(@ContextTranslate(ContextTranslatorImpl.class) StartEventNotification parameters).

Passes on the request to SendDataPluginToNetworkAdapter.

If there is a need to retry the request, this method re-throws a PluginRetryException, so the request can be retried by the service interceptors.

SendDataPluginSouth

Class.

implements PluginSouth.

public SendDataPluginSouth()

Constructor.

Empty.

public void send(NetworkProxy proxy, String address, String data)

Sends data to the network node.

Passes on the request to sendDataToNetwork using the NetworkProxy.

The method is annotated with @Edr.

public String resolveAppInstanceGroupdId(ContextMapperInfo info)

Empty implementation that returns null. This method has meaning, and is used, only in network-triggered requests.

The application instance ID is already known in the RequestContext, since the class only handles application-initiated requests.

public void prepareRequestContext(RequestContext ctx, ContextMapperInfo info))

From interface com.bea.wlcp.wlng.api.plugin.PluginSouth

Gives the plug-in an opportunity to add additional values to the RequestContext before the application-initiated requests is passed on to public void send(NetworkProxy proxy, String address, String data).

Empty in this example. Normally all data about the request should be known at this point, so no additional data needs to be set.

SendDataPluginToNetworkAdapter

Interface.

Defines the interface between the plug-in and the network node for application-initiated requests.

SendDataPluginToNetworkAdapterImpl

Class.

public SendDataPluginToNetworkAdapterImpl()

Constructor.

Instantiates SendDataPluginSouth.

public void setNetworkProxy(NetworkProxy networkProxy)

Sets the NetworkProxy object. This is a remote object in the network node.

public void send(String address, String data)

Hands off the request to the network node using SendDataPluginSouth.

FilterImpl

Class.

Implements interface com.bea.wlcp.wlng.api.storage.filter.Filter.

This is the query filter used for the named store NotificationData.

Evaluates whether an entry in the named store NotificationData matches the filter. The filter is defined in XML, see Store configuration.

public boolean matches(Object value)

Must be invoked after public void setParameters(Serializable ... parameters).

Returns true if the value provided in Object matches parameters[0], as set in public void setParameters(Serializable ... parameters).

public void setParameters(Serializable ... parameters)

Sets the query parameters for the filter.

The parameters are ordered as provided to the StoreQuery and it is the responsibility of the implementation to handle them in this order.

NotificationData

Class.

Implements Serializable

The data structure representing a notification. The notification is registered and de-registered by applications using the application-facing Web Services interfaces and represents a subscription for network-triggered events. The NotificationData is used for:

- Matching a network-triggered event with a subscription started by an application. The match is usually based on the destination address in the requests from the network.
- Resolving information on which application instance created the subscription, and the endpoint on which the application expects to be notified of the event.

NotificationData is stored using the storage service, normally using the invalidating cache storage provider for cluster-wide access and high performance.

Each of the attributes to be stored must have a corresponding set method and get method.

The class must be serializable.

public NotificationData()

Constructor.

Empty.

StoreHelper

Class.

Singleton.

Helper class for storing NotificationData using the StorageService.

public static StoreHelper getInstance()

Returns the single instance of StoreHelper.

public void addNotificationData(URI address, NotificationData notificationData)

Stores the NotificationData using the Storage Service.

The named store is retrieved using private Store<String, NotificationData> getStore().

The NotificationData is put into the named store. The address is the key and the object is the value.

The named store is released. This should always be done in a finally{...} block.

public void removeNotificationData(String correlator)

Removes NotificationData using the StorageService.

The named store is retrieved using private Store<String, NotificationData> getStore().

A Set of matching entries are returned using private Set<Map.Entry<String, NotificationData>> getEntries(String correlator, Store<String, NotificationData> store).

If there are matching entries, all are removed using private void removeEntries(Set<Map.Entry<String, NotificationData>> set, Store<String, NotificationData> store).

The named store is released. This should always be done in a finally{...} block.

public NotificationData getNotificationData(String destinationAddress)

Gets NotificationData using the StorageService

The named store is retrieved using private Store<String, NotificationData> getStore().

The NotificationData that is keyed on destinationAddress is fetched from the store.

The named store is released. This should always be done in a finally{...} block.

private Store<String, NotificationData> getStore()

Gets a named stored from com.bea.wlcp.wlng.api.storage.StoreFactory.

private Set<Map.Entry<String, NotificationData>> getEntries(String correlator, Store<String, NotificationData> store)

Gets a java.util.Set of entries of NotificationData from a named store using the StorageService. The query being used is a named query, com.bea.wlcp.wlng.plugin.example.netex.Query, defined in wlng-cachestore-config-extensions.xml.

private void removeEntries(Set<Map.Entry<String, NotificationData>> set, Store<String, NotificationData> store)

Removes a java.util.Set of entries of NotificationData using the StorageService. The NotificationData is removed from a named store.

ExamplePluginInstance

Class.

Implements com.bea.wlcp.wlng.api.plugin.ManagedPluginInstance.

Defines the life-cycle for a plug-in instance.

Also holds the data that is specific to the plug-in instance.

public ExamplePluginInstance(String id, ExamplePluginService parent)

Constructor.

The id is the plug-in instance ID, and the parent is the Plug-in service the of which the plug-in is an instance.

public String getId()

The plug-in instance returns the ID that it was instantiated with.

public void activate()

Called when the plug-in instance is activated, so the plug-in:

- Instantiates the traffic interfaces.
- Registers the traffic interfaces with the Plug-in Manager.
- Register callbacks between the interfaces.

- Initiates the Store.
- Instantiates and registers the MBean interface.

If the plug-in service is in state ACTIVE (RUNNING), public void handleResuming() is called.

public void handleResuming()

Connects to the network node.

If the connection fails, a timer is triggered to retry the connection setup.

public void deactivate()

Called when the plug-in instance is deactivated.

If the plug-in service is in state ACTIVE (RUNNING), public void handleSuspending() is called.

The call-back is unregistered from the network node.

The MBean is unregistered.

public void handleSuspending()

If existing, the timer associated with connection setup is cancelled.

The plug-in disconnects from the network node.

public List<PluginInterfaceHolder> getNorthInterfaces()/ public List<PluginInterfaceHolder> getSouthInterfaces()

Returns a list of the interfaces.

public boolean isConnected()

Returns true if there is a connection to the network node, that is if the plug-in instance is ready to accept traffic.

public int customMatch(RequestInfo requestInfo)

Checks the operation that is about to be invoked on the plug-in instance by introspection of the RequestInfo associated with request.

If the operation is StopEventNotification and the correlator provided is cached using the Storage service, the request must be sent to all instances of the plug-in, since the request depends on an earlier request (startNotification). MATCH_REQUIRED is returned.

If the operation is any other than StopEventNotification, the request is unrelated to any previous operation and any plug-in instance can be used. MATCH_OPTIONAL is returned.

private void rethrowDeploymentException(Exception e)

Re-throws a DeploymentException given another exception. The exception is wrapped in a DeploymentException.

public ConfigurationStoreHandler getConfigurationStore()

Gets the ConfigurationStoreHandler.

ExamplePluginService

Class.

Implements com.bea.wlcp.wlng.api.plugin.ManagedPluginService.

Defines the life-cycle for a plug-in service.

Also holds the data that is specific for the plug-in instance.

public ExamplePluginService()

Constructor.

Empty.

public TimerManager getTimerManager()

Gets a handle to the TimerManager.

public boolean isRunning()

Checks if the plug-in service is in RUNNING state.

public String[] getSupportedSchemes()

Returns an array of supported address schemes.

public void init(String id, PluginPool pool)

Initializes the plug-in service with the ID and a reference to the plug-in pool.

The PluginPool holds all plug-in instances.

public void doStarted()

Instantiates a TimerManager to be used.

public void doStopped()/public void doActivated()/public void doDeactivated()

Empty implementation. Nothing to do here.

public void handleResuming()

Iterates over all plug-in instances using the PluginInstancePool and calls public void handleResuming() on ExamplePluginInstance

public void handleSuspending(CompletionBarrier barrier)

The nature of the example network protocol is that is does not have connections to maintain. Because it is possible to treat this event as in public void handleForceSuspending () the request is passed on to that method.

public void handleForceSuspending ()

When the plug-in service is being forcefully suspended, the plug-in instances are disconnected from the network node immediately, without waiting for any in-flight requests to complete.

This is done by iterating over the PluginInstancePool and calling public void handleSuspending() on ExamplePluginInstance

public ServiceType getServiceType()

Returns the service type, com.acompany.example.servicetype.ExampleServiceType.type. The type is automatically generated when the service EJB is generated.

public String getNetworkProtocol()

Returns the network protocol. A string used for informational purposes.

public ManagedPluginInstance createInstance(String pluginInstanceId)

Creates a new instance of the plug-in service. The ID for the new plug-in is supplied together with the object that created the instance (this).

Store configuration

The store configuration file `wlng-cachestore-config-extensions.xml` defines:

- Which data to store
- The get and set methods to retrieve and store the data
- The database table structure use to store the data
- Queries to perform on the store

Listing 5-5 shows the store configuration file for the example Communication Service.

The configuration file defines:

- The store type ID: since the store type ID is prefixed with wlng.db.wt (wlng.db.wt.es_example), the store is a write-through cache.
- The table to be used: es_example
- The identifier for the store is a combination of the type of the key column (java.lang.String) and the type of the value column (com.acompany.plugin.example.netex.store.NotificationData). These are used when the store is retrieved from the StoreFactory, see private Store<String, NotificationData> getStore()
- The key column: address
- The value columns for the key:
  - correlator
  - endpoint
  - appinstance
- The get and set methods for the value columns.
- The query to use when doing lookups in the store.

The configuration file, together with any non-complex data types must be packaged into a .jar and put in the directory `$DOMAIN_HOME/config/store_schema` so it can be accessed by the storage service.

Listing 5-5 Store configuration for the example Communication Service

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

<store-config xmlns="http://www.bea.com/ns/wlng/30"
```

```
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.bea.com/ns/wlng/30
wlng-cachestore-config.xsd">

  <db_table name="es_example">
    <key_column name="address" data_type="VARCHAR(100)"/>
    <value_column name="correlator" data_type="VARCHAR(100)">
      <methods>
        <get_method name="getCorrelator"/>
        <set_method name="setCorrelator"/>
      </methods>
    </value_column>
    <value_column name="endpoint" data_type="VARCHAR(255)">
      <methods>
        <get_method name="getEndPoint"/>
        <set_method name="setEndPoint"/>
      </methods>
    </value_column>
    <value_column name="appinstance" data_type="VARCHAR(100)">
      <methods>
        <get_method name="getApplicationInstance"/>
        <set_method name="setApplicationInstance"/>
      </methods>
    </value_column>
  </db_table>

  <store type_id="wlng.db.wt.es_example" db_table_name="es_example">
    <identifier>
```

```
      <classes key-class="java.lang.String"
value-class="com.acompany.plugin.example.netex.store.NotificationData"/>

    </identifier>

    <index>

      <get_method name="address"/>

    </index>

  </store>


  <query name="com.bea.wlcp.wlng.plugin.example.netex.Query">

    <sql><![CDATA[SELECT * FROM es_example WHERE correlator LIKE ?]]></sql>

  </query>


</store-config>
```

SLA Example

Below is an example SLA for the example Communication Service. There are examples of service provider group and application group SLAs in:

`$PDS_HOME\pte\resource\sla`

Listing 5-6 Example SLA for the example Communication Service

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<Sla xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
applicationGroupID="default_app_group"
xsi:noNamespaceSchemaLocation="app_sla_file.xsd">

    <serviceContract>

        <startDate>2008-04-17</startDate>

        <endDate>2099-04-17</endDate>

        <scs>com.acompany.example.plugin.SendDataPlugin</scs>
        <contract/>
    </serviceContract>
    <serviceContract>
        <startDate>2008-04-17</startDate>
        <endDate>2099-04-17</endDate>
        <scs>com.acompany.example.plugin.NotificationManagerPlugin</scs>
        <contract/>
    </serviceContract>
    <serviceContract>
        <startDate>2008-04-17</startDate>
        <endDate>2099-04-17</endDate>
        <scs>com.acompany.example.callback.NotificationCallback</scs>
        <contract/>
    </serviceContract>
</Sla>
```

Service Enabler Example with SIP plug-in

This section describes the example network protocol plug-in for SIP connectivity provided in the Platform Development Studio:

Overview

The SIP Plug-in example demonstrates the following:

- Structure and execution workflow in a Communication Service.
- Parameter validation

- Hitless upgrade
- Retry
- SIP connectivity using a SIP Servlet
- Testability with the PTE

The example is based on an end-to-end Communication Service, with a set of simple interfaces

- SendData, which defines the operation `sendData` used to send data to a given address.
- NotificationManager, which defines these operations:
  - `startEventNotification`, that starts a subscription for network-triggered events.
  - `stopEventNotification`, that ends the subscription for network-triggered events.
- Notification, which defines the operation:
  - `notifyDataReception`, used to notify the application on a network-triggered event.

The SendData and NotificationManager interfaces are used by an application and implemented by the Communication Service.

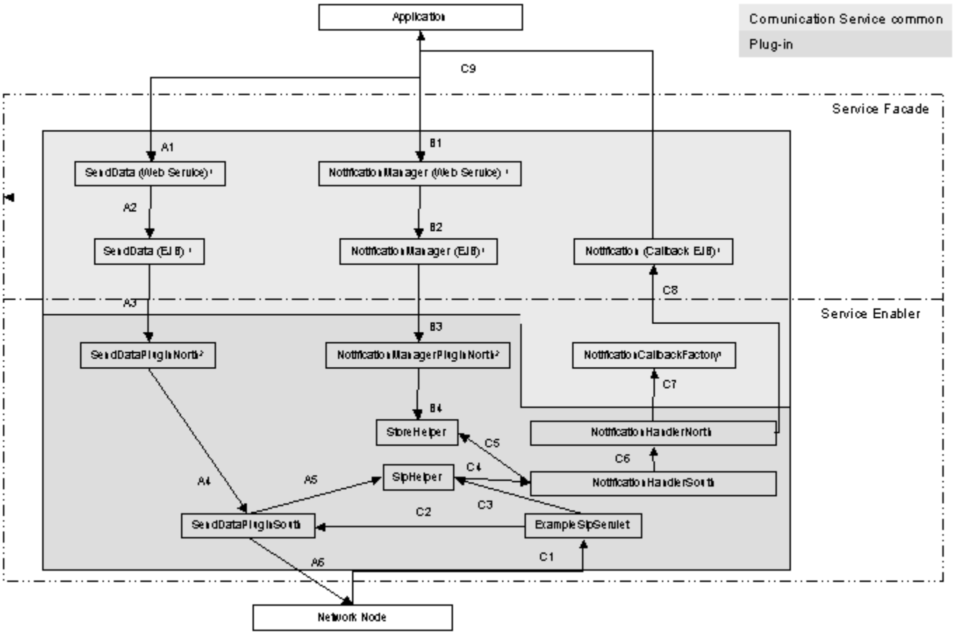
The Notification interface is used by the Communication Service and implemented by an application.

The Communication Service to network node interface is a simple SIP based interface that defines the two commands:

- `send`, that sends data to the SIP network.
- `receiveData`, that is used by the network node to send data to a receiver - in this case the network protocol plug-in.

Figure 6-1 illustrates the flow for these operations.

Figure 6-1 Overview of example Communication Service with SIP plug-in



The flow marked A* is for `sendData`, the flow marked B* is for `startNotification` and `stopNotification`, and the flow marked C* is for `notifyDataReception`.

The modules marked with 1 are automatically generated based on the WSDL files that defines the application-facing interface and code generation templates provided by the Platform Development Studio. The modules marked with 2 are skeletons generated at build time.

High-level Flow for sendData (Flow A)

1. A1: An application invokes the Web Service SendData, with the operation `sendData`.
2. A2: The request is passed on the EJB for the interface, which passes it on to the network protocol plug-in. The diagram is simplified, but at this stage the Plug-in Manager is invoked and makes a routing decision to the appropriate plug-in.

3. A3: The Plug-in Manager invokes the `sendData` method in the class SendDataPluginNorth. It will always invoke a class named PluginNorth, that has a prefix that is the same as the Java representation of the Web Service interface.
4. A4: The SIP request is created.
5. A5: The the SIPFactory is fetched from ExampleSIPHelper.
6. A6: The request is handed off to the network node.

High-level Flow for startNotification and stopNotification (Flow B)

The initial steps (B1-B3) are similar to flow A*. Instead of translating the request to a command on the network node, NotificationManagerNorth uses the StoreHelper to either store a new or remove a previously registered subscription for notifications. The data stored, the NotificationData, is used in network-triggered requests to resolve which application started the notification and the destination to which to send it. In the example the notification is started on an address, so the address is stored together with information to which endpoint the application wants the notification to be sent.

High-level flow for notifyDataReception (Flow C)

1. C1: The network protocol plug-in receives the network-triggered SIP message to ExampleSipServlet.
2. C2: SendDataPluginSouth can be used to add additional information to the request before passing in on.
3. C3: ExampleSipHelper finds a plug-in instance to pass on the request to.
4. C4: ExampleSipHelper calls NotificationHandlerSouth.
5. C5: StoreHelper is used to examine if the request matches any stored NotificationData. If so, the information in NotificationData is retrieved. This information includes which application instance that the request resolves to and on which endpoint this application wants to be notified about the network triggered event.
6. C6: NotificationCallbackFactory is used to get a hold of an active NotificationCallback EJB to pass on the request to.
7. C7: The request is passed on to the NotificationCallback EJB.

8. C8: The request is passed on to an application.

Interfaces

The SIP plug-in translates between an application-facing interface, defined in WSDL, see Web Service Interface Definition and a SIP network interface, see Network Interface Definition.

Web Service Interface Definition

The WSDL, and Service Facade used is the same as for the Example Communication Service, see Web Service Interface Definition in Communication Service Example.

Network Interface Definition

The network interface is SIP and the plug-in uses the Oracle Converged Application Server SIP Servlet container to process and create SIP messages.

Application-initiated requests are converted to regular SIP messages. It is configurable whether to send it to a SIP Proxy or not.

All SIP messages that arrive to the plug-in are processed and passed on the application that has subscribed for notifications that matches the network-triggered request.

Directory Structure

The directory structure is similar to the directory structure for the example Communication Service, see Directory Structure in Communication Service Example but adds a set of classes, descriptors, and artifacts as described below:

```
| +- plugins

| | +- sip

| | | +- config

| | | | +- sip

| | | | | +- WEB-INF

| | | | | | +- sip.xml

| | | | | | +- web.xml

| | | +- dist
| | | | +- com.acompany.plugin.example.sip.store_4.0.jar
| | | | +- example_sip_plugin.jar
| | | | +- example_sip.war
| | | +- src/com/acompany/plugin/example/sip/
| | | |           +- context
| | | |           +- management
| | | |           +- notification
| | | |           +- notificationmanager
| | | |           +- senddata
| | | |           +- servlet
| | | |           +- store
| | | +- storage
| | | | +- wlng-cachestore-config-extensions.xml
```

Differences Compared to the Example netex Plug-in

The source for the example SIP plug-in is very similar to the netex plug-in described in Communication Service Example. Below is a list of the classes that are added or changed.

The SIP plug-in has a different package structure compared to the netex plug-in:

```
package com.acompany.plugin.example.sip.*
```

The following classes are new, and relates to the SIP protocol:

- com.acompany.plugin.example.sip.servlet.ExampleServlet
- com.acompany.plugin.example.sip.ExampleSipHelper

The class com.acompany.plugin.example.netex.senddata.south.SendDataPluginToNetworkAdapter has been replaced by direct calls from SendDataPluginNorth to SendDataPluginSouth.

The class com.acompany.plugin.example.netex.notification.south.SendDataPluginToNetworkAdapter has been replaced by com.acompany.plugin.example.sip.notification.south.NotificationHandlerSout

h. The class also does a lookup for matching subscriptions. In the netex plug-in, this is done by NotificationHandlerNorth.

The class `com.acompany.plugin.example.sip.senddata.south.SendDataPluginSouth` has been updated to use `ExampleSipHelper`.

The MBean `com.acompany.plugin.example.sip.management.ExampleMBean` has been changed to contain SIP-related attributes.

The store definition classes:

- `FilterImpl`
- `NotificationData`
- `StoreHelper`

and the storage service configuration `wlng-cachestore-config-extensions.xml` is updated to use another store.

Configuration files for the SIP Servlet has been added:

- sip.xml
- web.xml

The build artifacts have been changed to:

- `com.acompany.plugin.example.sip.store_4.0.jar`
- `example_sip_plugin.jar`
- `example_sip.war`

Configuration Files and Artifacts

The SIP Servlet-defined configuration files for the SIP application is added to `WEB-INF/sip.xml` in `example_sip.war`.

The Java EE standard configuration file for the Web application is added to `WEB-INF/application.xml` in `example_sip.war`.

Both configuration files are found in `$PDS_Home/example/communication_service/example/plugins/sip/config/sip`.

The following artifacts are generated when the plug-in is built:

- `com.acompany.plugin.example.sip.store_4.0.jar`, the store definition for the plug-in.

- `example_sip_plugin.jar`, the plug-in where most of the processing logic takes place.
- `example_sip.war, the servlet part of the plug-in.`

The build artifacts are created in `$PDS_Home/example/communication_service/example/plugins/sip/dist`

The deployable Service Enabler is created when the Communication Service is built. It is packaged in the EAR file `example_enabler.ear` in `$PDS_Home/example/communication_service/example/dist`.

The store definition JAR file is also generated to this directory.

Note that both the netex plug-in and the SIP plug-in will be packaged in `example_enabler.ear`.

The configuration files:

- alarm.xml
- cdr.xml
- edr.xml

are provided in `$PDS_Home/example/communication_service/example/plugins/sip/config/edr`

Add the contents of these files to Oracle Communications Services Gatekeeper when deploying the Service Enabler.

Classes

Below is a description of classes that are new or have been changed compared to the netex plug-in described in Communication Service Example.

ExampleServlet

Package: com.acompany.plugin.example.sip.servlet

Extends javax.servlet.sip.SipServle

The SIP Servlet part of the plug-in. Uses ExampleSipHelper to manage network-triggered requests.

public void init()

Initialization for the SIP Servlet.

Calls init() on ExampleSipHelper and provides the ServletContext to ExampleSipHelper.

protected void doMessage()

Handles network-initiated SIP messages.

Returns a SIP 200 OK Response to the network.

Extracts the to and from URIs, and the content of the SIP message and calls notifyCallbacks with these parameters on ExampleSipHelper.

ExampleSipHelper

Package: com.acompany.plugin.example.sip

Singleton class that holds the SIPFactory, the SipSessionsUtil, and list of plug-in instances that can be used to process network-triggered messages.

public void init(ServletContext servletContext)

Initialization for ExampleSipHelper.

Called by ExampleSIPServlet, when it is being deployed.

Fetches the SipFactory and the SipSessionsUtil from the ServletContext and stores them in member variables.

public SipSessionsUtil getSessionsUtil()

Get method for SipSessionsUtil.

public SipFactory getSipFactory()

Get method for SipFactory.

public synchronized void registerCallback(NetworkCallback callback)

Called by the plug-in instance when it is being activated. Registers NotificationHandlerSouth in ExampleSipHelper. NotificationHandlerSouth is responsible for processing of network-triggered requests.

public synchronized void unregisterCallback(NetworkCallback callback)

Called by the plug-in instance when it is being deactivated. Unregisters NotificationHandlerSouth from ExampleSipHelper.

public synchronized void notifyCallbacks(String fromAddress, String toAddress, String message)

Called by ExampleSipHelper when a network-triggered SIP message arrives.

Resolves a plug-in instance to deliver a network-triggered request to. Since all plug-in instances register their own instance of NotificationHandlerSouth, there are as many possible plug-in instances to use as there are plug-in instances. In the example, only one instance is picked since they all have the same logic and access to the same notification data.

An alternative way to implement it is to call all instances. The notification data in the store may or may not be shared among plug-in instances. It is up to the designer of the plug-in to decide which pattern to use. If the notification data is tied to the plug-in instance, the alternatives are to call all plug-in instances or to establish communication channels between the different plug-in instances in order to resolve which instance that shall be targeted for the request.

SendDataPluginSouth

Class.

Implements PluginSouth.

public SendDataPluginSouth()

Constructor.

Empty.

public void send(String address, String data)

Sends data to the SIP network.

Creates a SipApplicationSession and a SipServletRequest and sends the request to the SIP network.

The SipServletRequest is created as a SIP Message, with the From: address set to identify Oracle Communications Services Gatekeeper, and the To: address to the address provided by the application.

The content of the SIP message contains the SIP Proxy URI fetched from the configuration store.

The method is annotated with @Edr.

public String resolveAppInstanceGroupdId(ContextMapperInfo info)

Empty implementation that returns null. This method has meaning, and is used, only in network-triggered requests.

The application instance ID is already known in the RequestContext, since the class only handles application-initiated requests.

public void prepareRequestContext(RequestContext ctx, ContextMapperInfo info))

From interface com.bea.wlcp.wlng.api.plugin.PluginSouth

Gives the plug-in an opportunity to add additional values to the RequestContext before the application-initiated requests is passed on to public void send(String address, String data).

Empty in this example. Normally all data about the request should be known at this point, so no additional data needs to be set.

NotificationHandlerSouth

Class.

Implements PluginSouth, NetworkCallback.

public NotificationHandlerNorth()

Constructor.

Empty.

public void receiveData(@ContextKey(EdrConstants.FIELD_ORIGINATING_ADDRESS) String fromAddress, @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS) @MapperInfo(C) String toAddress, String data)

Handles network-triggered requests from ExampleSipHelper.

Generates EDRs, finds the application instance that has subscribed for notifications, and passes on the request to NotificationHandlerNorth.

public String resolveAppInstanceGroupdId(ContextMapperInfo info)

Resolves which application instance that has subscribed for notifications that matches the data in the network-triggered request. Use StoreHelper to find the subscription for notifications.

public void prepareRequestContext(RequestContext ctx, ContextMapperInfo info))

From interface com.bea.wlcp.wlng.api.plugin.PluginSouth

Empty implementation in this example. Normally all data about the request should be known at this point, so no additional data needs to be set. This method has meaning, and is used, only in network-triggered requests.

Gives the plug-in an opportunity to add additional values to the RequestContext before the network-triggered requests are passed on to NotificationHandlerNorth...

ExampleMBean

Interface.

Management interface.

It defines the following methods:

- public void setProxyURI(String uri) throws ManagementException;
- public String getProxyURI() throws ManagementException;

Implemented by ExampleMBeanImpl.

Stores the URI to the SIP proxy to send application-initiated requests to in the configuration store for the plug-in instance.

All MBean methods should throw com.bea.wlcp.wlng.api.management.ManagementException or a subclass thereof if the management operation fails.

SLA

The SLA is on Communication Service level and identical to the one for the example Communication Service, see Communication Service Example.

Container Services

This chapter provides a high-level description of Oracle Communications Services Gatekeeper container services. It also provides an overview of other parts of the API available for the use of extension developers:

  - StorageService
- Shared libraries

JavaDoc for the container API is available in the `$PDS_Home/doc/javadoc` directory of the Platform Development studio installation and on the Oracle Communications Services Gatekeeper site at `http://www.oracle.com/technology/documentation/bea.html`.

Container service APIs

The Oracle Communications Services Gatekeeper container service APIs provide the basic infrastructure by which a Communication Service and the container services of Oracle Communications Services Gatekeeper can communicate.

All APIs for inter-working with the container services are found in com.bea.wlcp.wlng.api.*.

In order for a network protocol plug-in of a Communication Service to interact with Oracle Communications Services Gatekeeper it must be *deployable* in the context of Oracle Communications Services Gatekeeper. Once it is deployable, it can have access to certain utility functions.

.

Table 7-1 Summary of the container services APIs

Package	Summary
com.bea.wlcp.wlng.api.account	Represents an application instance and the related accounts and groups and the states of the accounts.
com.bea.wlcp.wlng.api.corba	Factory to retrieve an ORB.
com.bea.wlcp.wlng.api.edr.*	Annotations, interfaces and classes used when annotating EDRs. Descriptor classes for alarms, EDRs, and CDRs. Helper classes for EDR listeners. See Annotations, EDRs, Alarms, and CDRs.
com.bea.wlcp.wlng.api.event_channel	Classes to publish and listen to events over cluster-wide event channels. See Service: EventChannel Service.
com.bea.wlcp.wlng.api.interceptor	Interfaces and classes for service interceptors. See Service Interceptors.

Table 7-1 Summary of the container services APIs

Package	Summary
com.bea.wlcp.wlng.api.management.*	MBean helper classes. See Making Communication Services Manageable.
com.bea.wlcp.wlng.api.plugin.*	Plug-in related classes and interfaces. See Plug-in.
com.bea.wlcp.wlng.api.servicecorrelation	Interface to implement if extending the existing service correlation mechanism. See Service Correlation.
com.bea.wlcp.wlng.api.statistics	Annotation for statistics. See Service: Statistics service.
com.bea.wlcp.wlng.api.storage	Interfaces and classes for the Storage Service. See Storage Services.
com.bea.wlcp.wlng.api.timers	Factory for using commonj.timers API.
com.bea.wlcp.wlng.api.util	Classes and interfaces for commonly used functions, for example ID generator, InstanceFactory, and clustering.
com.bea.wlcp.wlng.api.work	Factory for using commonj.work API.

Class: InstanceFactory

The Instance Factory is the mechanism used in Oracle Communications Services Gatekeeper to retrieve instances of a given interface, class, or abstract class. You retrieve an instance of the Instance Factory using the public static method `getInstance()`. The factory itself has a single method:

> `getImplementation(Class theClass)` - Retrieves a class that implements a given interface or extends a given class

The implementation to be used is located and used based on the following rules:

1. First, check the jar file's `instancemap`, a standard `java.util.Properties` file. Every jar file can have its own `instancemap`. The `instancemap` provides a list that maps a given interface, class, or abstract class to the preferred implementation of that functionality. See Listing 7-1 for an example.

Note: The interface name used in the instancemap must be unique across all plug-ins for a given Service Enabler. It is not possible to use the same interface in two `instancemap` files belonging to two different plug-ins and still map them to two different implementations.

2. If a mapping is provided and the target class has a public constructor or static singleton method, instantiate it.
3. If there is no explicit mapping, or if there is no public constructor or static singleton method for a mapped class, instantiate an object named according to the following pattern: `theClass.getClass().getName() +”Impl”` if this exists and has a public constructor or static singleton method.

Listing 7-1 Example instancemap file

```
com.bea.wlcp.wlng.MyInterface=com.bea.wlcp.wlng.MyImplementation

com.bea.wlcp.wlng.MyOtherInterface=com.bea.wlcp.wlng.MyOtherImplementation
```

For details see Javadoc for Package `com.bea.wlcp.wlng.api.util` Class InstanceFactory.

Class: ClusterHelper

`com.bea.wlcp.wlng.api.util.cluster.ClusterHelper`

Helper class for getting the JNDI Context for the network and access tier.

For details see Javadoc for Package `com.bea.wlcp.wlng.api.util.cluster` Interface ClusterHelper.

Service: EventChannel Service

This service is used to broadcast events to other Oracle Communications Services Gatekeeper server instances and to register listeners for events originating in other Oracle Communications Services Gatekeeper server instances.

Interface: EventChannel

Use this interface to broadcast events to other instances of Oracle Communications Services Gatekeeper, and to register listeners for events originating in them. It is used, for example, in propagating changes of cached data. It is retrieved using the com.bea.wlcp.wlng.api.event_channel.EventChannelFactory.

An event has a name and a value, where the name is an identifier for the event and the value is any object implementing `java.io Serializable`.

The following methods are available:

- `deactivateAllListeners()` - Deactivates all registered listeners.
- `publishEvent` - Publishes an event to all registered listeners.
- `publishEventToOneNode` - Publishes an event to one Oracle Communications Services Gatekeeper instance.
- `registerEventListener` - Registers an `EventListener`.
- `unregisterEventListener` - Unregisters an `EventListener`.

Interface: EventChannelListener

This interface is used to receive events published using `EventChannel`.

The following method is available:

- `processEvent(String eventType, Serializable event, String source)` - Receives an event.

Service: Statistics service

Standard statistics are generated automatically when a plug-in implements `PluginNorth` and `PluginNorthCallBack` interfaces. In addition to this, custom statistics can be generated explicitly.

To explicitly generate statistics, annotate the method where you wish to generate statistics.

The syntax of the annotation is:

```
@Statistics(id=<My_Statistics_Type>)
```

```
@ExceptionStatistics(id=<My_Statistics_Type>)
```

The annotations are defined in:

```
om.bea.wlcp.wlng.api.statistics.Statistics
```

```
com.bea.wlcp.wlng.api.statistics.ExceptionStatistics
```

The @Statistics annotation generates a statistics event when the method returns, while the @ExceptionStatistics annotation generates a statistics event if an exception is thrown.

The statistics type must be registered. Use the addStatisticType operation in the Management Console. For more information, see “Managing and Configuring Statistics and Transaction Licenses” in the *System Administration Guide*.

For extensions, the statistics ID shall be in the range 1000 to 2250.

Plug-in

The com.bea.wlcp.wlng.api.plugin.* packages contain a range of interfaces and classes for use by the extension developer.

See Communication Service Description.

Management

Base classes and annotations for giving the Oracle Communications Services Gatekeeper Management Console or other JMX tools management access to Communication Services. See Chapter 10, “Making Communication Services Manageable” for more information. Also see the JavaDoc for the packages: com.bea.wlcp.wlng.api.management.*

EDR

See Chapter 9, “Annotations, EDRs, Alarms, and CDRs.”. Also see the JavaDoc for the packages com.bea.wlcp.wlng.api.edr.*

SLA Enforcement

SLA enforcement operates on methods identified by the Java representation of the interface, and the operation on the application-facing interface for the Communication Service or the service type of the Communication Service.

The content of the tag <scs> defined in the <serviceContract> tag in the SLA is the plug-in type for the plug-in.

An operation on the application-facing interface is represented in the rules according to the following scheme: <service name> and <operation name>.

Parameters in the operation are represented in the rules according to the following scheme:

arg<n>.<parameter name>

where <n> in arg<n> depends on the WSDL that defines the application-facing interface; normally this is arg0.

If the parameter in <parameter name > is

- a composed parameter, the notation is according to the Java Bean notation for that parameter.
- an enumeration, the notation is according to the Java-representation of that parameter, <parameter name >.<enumeration value>. The <enumeration value> is the String representation.

SLA enforcement can also be done for a certain service type. The service type is defined when generating the Communication Service or network protocol plug-in using the Eclipse Wizard. SLA enforcement for service types relates to quotas and request rates and are defined under the element <serviceTypeContract>.

For enforcement of custom SLAs, see Custom Service Level Agreements.

Service Correlation

It is often the case that service providers would like to be able to bundle what are to Oracle Communications Services Gatekeeper separate services into a single unit for charging purposes. An end user could send an SMS to the provider requesting the location of the coffee shop closest to her current location. The application would receive the network-initiated SMS (one service), do a user location lookup on the customer (one service), and then send the customer an MMS with a map showing the requested information (one service). So three Oracle Communications Services Gatekeeper services need to be grouped into a single service charging unit. To do this, Oracle Communications Services Gatekeeper provides the framework for a Service Correlation service that uses a Service Correlation ID (SCID) to combine/correlate all the services.

- The Service Correlation ID is optional.
- The Service Correlation ID is captured in the CDRs and EDRs generated from Oracle Communications Services Gatekeeper.
- The Service Correlation ID is propagated as a String.
- The Service Correlation ID is propagated to and from the application in the SOAP header.

The SCID itself is provided either by the application or by an external mechanism that the Communication Service must provide (see Interface: ExternalInvocation). Oracle

Communications Services Gatekeeper does not check whether or not it is unique. The SCID is stored in the OLS Work Context, so that it can be accessed by both the Access Tier and the Network Tier. The Service Correlation class registers itself as a `RequestContextListener`. When application-initiated request traffic enters the plug-in, the Service Correlation service takes the SCID from the Work Context and places it in the `RequestContext` object, where it will be available to the EDR service. When network-initiated request traffic is leaving the plug-in, the Service Correlation service takes the SCID from the `RequestContext` object and places it in the Work Context, where it can be retrieved by the SOAP Handler and passed along to the application.

Interface: ExternalInvocation

Because Service Correlator IDs may need to be stored across several invocations and a `RequestContext` object exists only for the lifetime of a single request, a Communication Service needs to create a way of storing and retrieving the SCIDs. This is done by implementing the `ExternalInvocation` interface. This interface has two methods: one stores the Service Correlation ID and one retrieves it. The implementor is free to modify the ID once it has been stored, or to use the Invocation object to create IDs in the first place.

When the Service Correlation service takes the SCID (should there be one) out of the Work Context of an application-initiated request, it automatically attempts to store it in an object of this type before putting the SCID in the `RequestContext`.

When a network-initiated request is leaving the plug-in, the Service Correlation service automatically attempts to retrieve an SCID from an object of this type, using the SCID (should there be one) it finds in the `RequestContext` object before it sets the Work Context. In this way, if the `ExternalInvocation` object has modified the SCID in any way, it is this modified version that is put in the Work Context and thus sent on to the application. The `ExternalInvocation` implementation class should have an empty public constructor or a static method that returns itself.

Class: ExternalInvocatorFactory

This class is used by the Service Correlation service to locate and instantiate the correct `ExternalInvocation` object. It does this by using an `instancemap.` The instancemap entry should look like this:

```
com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocation=myPackageStruc
ture.myImplClass
```

where `myImplClass` is the `ExternalInvocation` implementation.

Class: ServiceCorrelation

This class manages the transport and storage of the Service Correlation ID across multiple service invocations.

Implementing the ExternalInvocation Interface

There are four basic steps in creating a custom service correlation:

1. Create a jar file that includes your code. For example:

Listing 7-2 Sample Custom Service Correlation

```
package myPackageStructure;

import com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocation;

import
com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocationException;


public class MyImplClass implements ExternalInvocation {

  public MyImplClass() {

  }

  public String pushServiceCorrelationID(String scID, String serviceName,
String methodName, String spID, String appID, String appInstGrp) throws
ExternalInvocationException {

    // your code here

    return scID;

  }


  public String getServiceCorrelationID(String scID, String serviceName,
String methodName, String spID, String appID, String appInstGrp) throws
ExternalInvocationException {

    // your code here

    return scID;
  }

}
```

2. Create the instancemap. See Class: ExternalInvocatorFactory.
3. Put the instancemap file in the JAR. This makes your custom service correlation available to the service interceptor InvokeServiceCorrelation.
4. Put the JAR file in `$DOMAIN_Home/lib`.

Parameter Tunneling

Parameter tunneling is a feature that allows an application to send additional parameters to Oracle Communications Services Gatekeeper and lets a plug-in use these parameters. This feature makes it possible for an application to tunnel parameters that are not defined in the interface that the application is using and can be seen as an extension to the application-facing interface.

The application sends the tunneled parameters in the SOAP header of a Web Services request.

The tunneled parameter can be retrieved in a plug-in by the key. The parameter is fetched from the RequestContext, using the method `getXParam(String key)`. If a value for the key cannot be found, null is returned.

Listing 7-3 Get the value of the tunneled parameter 'aParameterName'.

```
RequestContext.getCurrent().getXParam("aParameterName");
```

If the same parameter is defined in the <**contextAttribute**> SLA tag, it should override the parameter tunneled from the application. This behavior, however, is defined per plug-in.

Storage Services

The storage services provided in Oracle Communications Services Gatekeeper are of two types, described below:

- “ConfigurationStore” on page 7-11
- “StorageService” on page 7-16

ConfigurationStore

The Oracle Communications Services Gatekeeper container exposes a `ConfigurationStore` Java API that Communication Services can use to store simple configuration parameters instead of using JDBC and caching algorithms in each module.

Note: This utility is intended for configuration parameters only, not traffic data

All data stored in a `ConfigurationStore` are stored in a database table and cached in memory.

Below are the characteristics of a `ConfigurationStore`:

- It is a named store.
- Parameters stored in it must be initialized before they can be used.
- Stores can be either domain wide (shared) or limited to a single Oracle Communications Services Gatekeeper server (local). The domain wide store type replicates all data changes to all servers in the cluster, while the local store type keeps a different view of the parameters on different servers and data changes affect only the view for that particular server.
- Parameters stored in a `ConfigurationStore` are persisted to database.
- Data in all `ConfigurationStores` are also cached in memory.
- Only one instance of each named `ConfigurationStore` is cached in memory per server.
- Updates to a cluster wide named `ConfigurationStore` is reflected in all cluster nodes.
- The named `ConfigurationStore` only supports parameters of type Boolean, Integer, Long, String, and Serializable.

Interfaces

The Java interface APIs are found in the package `com.bea.wlcp.wlng.api.storage`.

The entry point to configuration stores is through the com.bea.wlcp.wlng.api.storage.configuration.ConfigurationStoreFactory using the following method:

```
public abstract ConfigurationStore getStore(String moduleName, String name,
int storeType) throws ConfigurationException;
```

The `ConfigurationStore` service exposes an interface with the following features:

- Methods to initialize the store with the following data types:
  - Boolean,
  - Integer,
  - Long,
  - String
  - Serializable

  A ConfigurationStore is initialized using a name in key/value pair. You get and set configuration parameters using the key.

- Methods to set and get the following data types:
  - Boolean,
  - Integer,
  - Long,
  - String
  - Serializable

- Methods to add and remove listeners for notifications on updates. When a parameter has been updated in one instance of the `ConfigurationStore`, a notification is broadcast to all other instances of the `ConfigurationStore`.

Listing 7-4 is an example of using the Configuration Store.

Listing 7-4 Example of a ConfigurationStoreHelper

```
package com.acompany.plugin.example.netex.management;

import com.bea.wlcp.wlng.api.storage.configuration.*;

/**

 * Class used for handling the configuration store.

 *
 * @author Copyright (c) 2007 by BEA Systems, Inc. All Rights Reserved.
 */
public class ConfigurationStoreHandler {
/**
   * Constants used for the values stored in the store.
   */
  public static final String KEY_NETWORK_HOST = "KEY_NETWORK_HOST";
  public static final String KEY_NETWORK_PORT = "KEY_NETWORK_PORT";
/**
   * Constant to access either the local store. Note that these are
   * just names for the store.
   */
  private static final String LOCAL_STORE = "local";
/**
   * Local configuration store instance.
   */
  private ConfigurationStore localConfigStore;
/**
   * Constructor.
   *
   * @param pluginId The plugin id
   * @throws ConfigurationException An exception thrown if the initialization
failed
   */
  public ConfigurationStoreHandler(String pluginId)
    throws ConfigurationException {

   ConfigurationStoreFactory factory = ConfigurationStoreFactory.getInstance();
```

```
    localConfigStore = factory.getStore(pluginId, LOCAL_STORE,
            ConfigurationStore.STORE_TYPE_LOCAL);
    // To obtain a shared configuration store, use
ConfigurationStore.STORE_TYPE_SHARED

    localConfigStore.initialize(KEY_NETWORK_HOST, "localhost");
    localConfigStore.initialize(KEY_NETWORK_PORT, 5001);
  }

  /**
   * Sets an integer value in the local store.
   *
   * @param key The key associated with the value.
   * @param value The value to store.
   * @throws ConfigurationException An exception thrown if the operation failed
   */
  public void setLocalInteger(String key, Integer value)
    throws ConfigurationException {
    localConfigStore.setInteger(key, value);
  }
/**
   * Gets an integer value from the local store.
   *
   * @param key The key associated with the value.
   * @return The value associated with the key.
   * @throws InvalidTypeException thrown if type is invalid.
   * @throws NotInitializedException thrown if key value has not been
   * initialized.
```

```
    */
   public Integer getLocalInteger(String key)
     throws InvalidTypeException, NotInitializedException {
     return localConfigStore.getInteger(key);
   }
 /**
    * Sets a string value in the local store.
    *
    * @param key The key associated with the value.
    * @param value The value to store.
    * @throws ConfigurationException An exception thrown if the operation failed
    */
   public void setLocalString(String key, String value)
     throws ConfigurationException {
     localConfigStore.setString(key, value);
   }
 /**
    * Gets a string value from the local store.
    *
    * @param key The key associated with the value.
    * @return The value associated with the key.
    * @throws InvalidTypeException thrown if type is invalid.
    * @throws NotInitializedException thrown if key value has not been
    * initialized.
    */
   public String getLocalString(String key)
     throws InvalidTypeException, NotInitializedException {
```

```
    return localConfigStore.getString(key);
  }
}
```

StorageService

The Storage Service is used for storing data that is not configuration-related, but related to the traffic flow through a Communication Service, in a cluster-wide store.

It provides mechanisms for:

- Store initialization

  A store is created using the `StoreFactory` singleton class, by specifying either a key/value class pair where the value class should be a class that is unique to the Store (recommended), or a Store name.

- Basic Map usage

  Since the Store interface extends the `java.util.Map` interface, it can be used as any other Map, and it is extended to be a cluster-wide view of the store.

- Named queries

  In addition to the standard `java.util.Map` interface, Stores have support for a `StoreQuery` interface. The behaviors of these named queries are configured as part of the Storage Service configuration files. There is an option to define a cache filter and/or SQL query. If there is an index specified for the Store, this index can be used by implementing the `IndexFilter` interface for the cache filter. The index is automatically used for SQL queries that can make use of these indexes.

- Store listener

  The Store API has support for registering `StoreListeners`. These listeners get notified if the Storage Service decides to automatically remove Store entries (based on configuration parameters). It will not be notified if the extension itself removes entries from the Store.

- Cluster locking

Cluster wide locking can be done using the Store interface. This should be used if the same entry in a Store may be modified on multiple servers at the same time, to avoid getting errors due to concurrent modification when a transaction commits.

A Communication Service extension uses the StorageService through an API. The API functionality is implemented by a storage provider. Oracle Communications Services Gatekeeper uses a write-through invalidating storage provider. Invalidating stores are backed by a database table. Other storage providers, supporting additional features, can be integrated but are not supported out-of-the box.

Extensions can use the `com.bea.wlcp.wlng.api.storage.Store` interface. This interface extends a `java.util.Map` interface and adds the following methods:

- `addListener`: Adds a listener for the store.
- `getQuery`: Gets a named query.
- `lock`: Takes a cluster-wide lock.
- `release`: Releases the current store instance.
- `removeListener`: Removes a registered listener.
- `unlock`: Unlocks a previously obtained cluster-wide lock.

The storage service uses configuration files that define the configuration for stores and the relationship between the cluster-wide store and the database table that backs the store. In each configuration file it is possible to define named queries towards the store. There is one configuration file per plug-in. Each configuration store configuration file shall, together its XSD and any complex data types stored, be created and packaged in a JAR file, in the directory `$DOMAIN_HOME/config/store_schema`. The configuration file must be named `wlng-cachestore-config-extensions.xml` and it must be present in the root of the JAR.

For details about the store configuration file, see the corresponding xsd: `com.bea.wlcp.wlng.storage_4.1.0.0.jar/wlng-cachestore-config.xsd` in `$OCSG_HOME/modules`.

A Store is retrieved from `com.bea.wlcp.wlng.api.storage.StoreFactory`, either by the name of the store or by the class names of the key/value names. How to retrieve the Store depends on how the store is configured.

The store interface needs to be released when it is no longer needed. The programming model is to retrieve the Store from the StoreFactory when the Store is used, and to release it once it has finished, using `try { .. } finally { store.release(); }`

Listing 7-5 Example: retrieve a store identified by key/value classes, operate on it, and release it.

```
Store<String, NotificationData> store =
StoreFactory.getInstance().getStore(String.class, NotificationData.class);

try {

   notificationData = store.put(address.toString(), notificationData);

} finally {

   store.release();

}
```

If it is a named store, it can also be retrieved by name as illustrated below.

Listing 7-6 Retrieving a store by name

```
Store<Serializable,Serializable> store =
StoreFactory.getInstance().getStore("A", this.getClass().getClassLoader());
```

Store configuration file

The configuration file `wlng-cachestore-config-extensions.xml` defines attributes of the store and relations between the store, the cache for the store, and the mapping to a database table. This part is used by extension developers.

In addition, the configuration file can contain a section with mapping information between a store, the provider it uses, and the factory for the storage provider. This section should not be used by extension developers.

The XSD for the configuration file is located in `com.bea.wlcp.wlng.storage_4.1.0.0.jar/wlng-cachestore-config.xsd` in `$OCSG_HOME/modules`.

There is one configuration file per plug-in. The file must be embedded in a JAR that contains the file itself and any complex data types used. The JAR must be stored in `$DOMAIN_HOME/config/store_schema`.

Below is an example of a store configuration file for extensions.

Listing 7-7 Example of a store configuration file for extensions

```
<store-config>
  <db_table name="example_store_notification">

    <key_column name="address" data_type="VARCHAR(255)"/>
    <!-- bucket_column using default BLOB type -->
    <bucket_column name="notification_data_value"/>

    <value_column name="correlator" data_type="VARCHAR(255)">
      <methods>
        <get_method name="getCorrelator"/>
        <set_method name="setCorrelator"/>
      </methods>
    </value_column>

  </db_table>

  <store type_id="wlng.db.wt.example_store_notification"
         db_table_name="example_store_notification">
    <identifier>
      <classes key-class="java.lang.String"
value-class="com.acompany.plugin.example.netex.notification.NotificationData"/
>
    </identifier>
    <index>
```

```
      <get_method name="getCorrelator"/>
    </index>
  </store>

  <query name="com.bea.wlcp.wlng.plugin.example.netex.Query">
    <sql>
      <![CDATA[
    SELECT * FROM example_store_notification WHERE correlator = ?
   ]]>
    </sql>
<filter-class>com.acompany.plugin.example.netex.store.FilterImpl</filter-class
>
  </query>
</store-config>
```

A store is defined between the elements <**store-config**> and </**store-config**>

Each Store has three sections:

- **store**: Defines the store.
- **db_table**: Defines the database table used to persist data in the store.
- **query**: Defines queries on the store. This is optional, only required if non-key queries are used with the store.

<store>

The **store** section defines the store itself. The attribute **type_id** defines the type of the store and a store type identifier. The ID must be mapped to a provider store mapping defined in `wlng-cachestore-config.xml`.

The name should always have the prefix `wlng.db.wt.` when using the storage provider in Oracle Communications Services Gatekeeper. The prefix which indicates that it is a write-through cache, that is, data put in the store is always written to database without any delay.

The attribute **db_table_name** identifies the database definition to use.

store contains the following elements:

- **identifier:** Holds one **classes** element. This element defines the classes for the key and the value that defines the store. The class for the key is defined in the attribute **key-class** and the class for the value part is defined in the attribute **value-class**. If a named store is used, the name is given in the element **name**.
- **index**: Defines an index on the cache and one or more get methods. The methods maps to an index on the corresponding columns in the table and potentially a cache index if supported by the provider in use.

<db_table>

The **db_table** section defines the database table used to persist data in store. The attribute **name** defines the table name to use. This name must be the same as the **db_table_name** specified in the **store** section. It contains the following elements:

- **key_column**: Has the attributes **name** and **data_type**. The attribute **name** specifies the column name for the key and the attribute **data_type** specifies the SQL data type for the key.
- **multi_key_column**: Has the attributes **name** and **data_type**. The attribute **name** specifies the column name for *one* part of a *multi key column* and the attribute **data_type** specifies the SQL data type for *the part* of the key. The difference between **multi_key_column** and **key_column** is that **multi_key_column** supports 2 or more columns to be parts of the key, so **multi_key_column** can occur two or more times in the configuration file.
- **bucket_column**: Has the attribute **name**. This attribute specifies the name of the column for the value part of the store. By default this is a BLOB. There is an optional attribute data_type, that can be used if other data types are used. This must be a Java to SQL supported data type mapping and corresponds to the data type in the value part of the store.
- **value_column**: Is used if attributes in the value part of the store should be stored in a separate column. The attribute **name** defines the name of the column and the data_type specifies the SQL data type for the column. **value_column** has the sub-element **methods**, which encloses the elements **get_method** and **set_method.** The sub-element **methods** defines the names of the set and get methods for the data stored in **value_column** and the set and get methods for the attribute of the object in the store.

Listing 7-8 Example of single key column configuration

```
...
<db_table name="single_key_store">
  <key_column name="sample_key_1" data_type="VARCHAR(30)">
    <methods>
      <get_method name="getSampleKey1"/>
      <set_method name="setSampleKey1"/>
    </methods>
  </key_column>
  <value_column name="sample_value" data_type="VARCHAR(30)">
    <methods>
      <get_method name="getSampleValue"/>
      <set_method name="setSampleValue"/>
    </methods>
  </value_column>
</db_table>
…
```

Listing 7-9 Example of multi key column configuration

```
...
<db_table name="combined_key_store">
  <multi_key_column name="sample_key_1" data_type="VARCHAR(30)">
    <methods>
      <get_method name="getSampleKey1"/>
      <set_method name="setSampleKey1"/>
    </methods>
  </multi_key_column>
  <multi_key_column name="sample_key_2" data_type="INT">
    <methods>
      <get_method name="getSampleKey2"/>
      <set_method name="setSampleKey2"/>
    </methods>
  </multi_key_column>
  <value_column name="sample_value" data_type="VARCHAR(30)">
    <methods>
      <get_method name="getSampleValue"/>
      <set_method name="setSampleValue"/>
    </methods>
  </value_column>
</db_table>
...
```

<query>

In addition to the standard `java.util.Map` interface, Stores have support for a `StoreQuery` interface. The behavior of these named queries are configured as part of the Storage Service configuration files.

The query section specifies a named query and a filter associated with the named query. The attribute name defines the name of the query. When using the storage service, the query is fetched using this name. The SQL query towards the database is defined in the element **sql**. The actual query is defined in the element <![CDATA[.....]]>.

The filter is a class that implements `com.bea.wlcp.wlng.api.storage.filter.Filter`, and the name of the class is defined in the element **filter-class**. The filter implements the method `setParameters`, and a `matches(...)` method.

The `setParameters` method maps the parameters to the filter class or a PreparedStatement `setObject` call ordered as the parameter array given. The filter class must implement the `matches` method in such a way that it will yield the same result as the SQL query specified.

Listing 7-10 Example of a named query

```
<query name="com.bea.wlcp.wlng.plugin.example.netex.Query">

    <sql>

      <![CDATA[

    SELECT * FROM example_store_notification WHERE correlator = ?

   ]]>

    </sql>

<filter-class>com.acompany.plugin.example.netex.store.FilterImpl</filter-class
>

  </query>
```

Listing 7-11 Example of using the named query using a filter

```
StoreQuery<String, NotificationData> storeQuery =
store.getQuery("com.bea.wlcp.wlng.plugin.example.netex.Query");

storeQuery.setParameters(correlator);

set = storeQuery.entrySet();
```

Listing 7-12 Example of a filter implementation

```
public class FilterImpl implements Filter {


  /**

   * The query parameters.
```

```
 */
private Serializable[] parameters;

/**
 * Default constructor.
 */
public FilterImpl() {

}

/**
 * Evaluate if a store entry matches the filter.
 *
 * @param value The store entry value to evaluate.
 */
public boolean matches(Object value) {

   if (parameters == null || value == null || parameters.length == 0) {

     return false;
  }

  if (value instanceof NotificationData) {
    String compareValue = ((NotificationData) value).getCorrelator();

    if (compareValue != null) {
      return compareValue.equals(parameters[0]);
      }
      return compareValue == parameters[0];
    }

    return false;
  }

  /**
   * Set query parameters. The parameters will be ordered as provided to the
   * StoreQuery and it it the responsibility of the implementation to handle
   * them in this order.
   *
   * @param parameters The query parameters to use.
   */
  public void setParameters(Serializable ... parameters)
    throws StorageException {

    this.parameters = parameters;
  }


}
```

<provider-mapping>

The **provider-mapping** section contains definitions of which storage provider a given type-id is mapped to. This section shall not be used unless a custom storage provider is used.

In the **type_id** attribute for **store_mapping type**, the same ID shall be used as when the store was defined. A best match (longest matching entry) is performed. A wildcard (*) can be used at the end of **type_id** to match the prefix.

The **<provider-name>** entry references the type of store being used, see "<providers>" on page 7-27.

The **type_id** for the storage provider mapping in use is wlng.db.wt.*. which references the write-through provider.

There is another set of **type_id** attributes defined for **store_mapping**:

- wlng.db.log.*, which is used for internal purposes only.
- wlng.db.wb.*, which is used if the storage provider supports write-behind operations. The invalidating storage provider does not support write-behind operations, write-through will be used.
- wlng.db.wt.*, which is used if the storage provider supports write-through operations.
- wlng.cache.*, which is used if the storage provider supports cache-only operations. The invalidating storage provider does not support cache-only operations, write-through will be used.
- wlng.local.*, which is used for internal purposes only.

These store mapping types are present for internal and future use. All store mapping types (except for the internal wlng.db.log.*) are by default mapped to the keyword `invalidating` which represents the invalidating storage provider. This should not be changed unless a custom storage provider is used.

<providers>

The **providers** section contains mappings between the **provider-name** defined in the **provider-mapping** section and the factory class for the storage provider. This section should not be changed used unless a custom storage provider is used.

Shared libraries

It is possible for multiple plug-ins to share common libraries, for example a third party library or custom code that can be shared.

If there are such parts, these should preferably not be packaged into the plug-in jar but instead be copied into the `APP-INF/lib` directory of the Communication Service network tier EAR. All classes in this directory are available for all of the plug-ins in the EAR.

Communication Service Description

This chapter provides a description of a Communication Service and its components:

- High-level Components
- Communication Service Common
- Plug-in
- Management
- SLA Enforcement
- Shared libraries

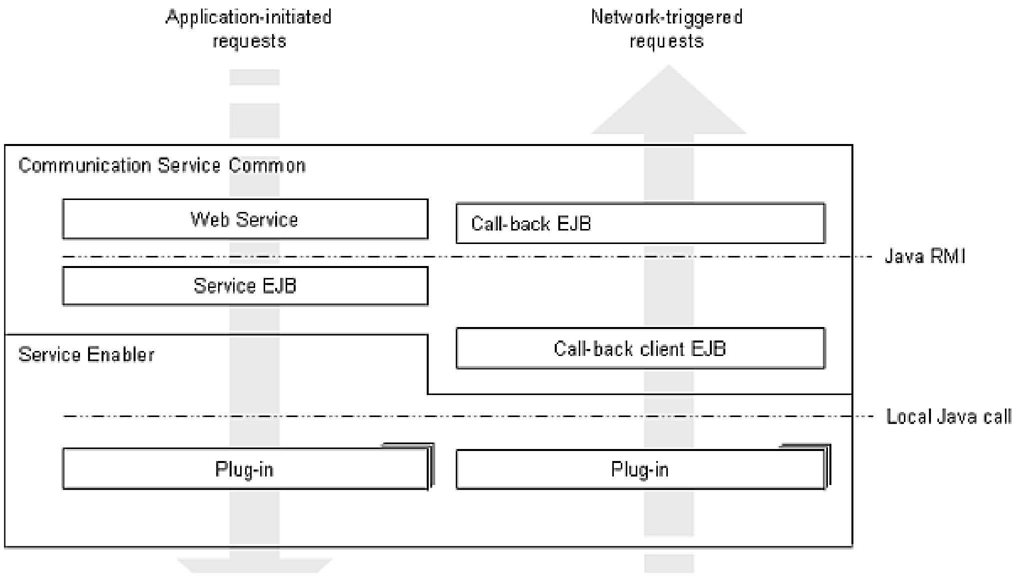
High-level Components

A Communication Service consists of a set of components:

- A Service Web Service (SOAP or RESTful)
- A Service EJB
- A Callback EJB
- A Call-back client EJB
- A set of network protocol plug-ins

One section of the set handles application-initiated requests, while another handles network-triggered requests, as illustrated in Figure 8-1. Some calls are remote since the modules may be deployed in separate clusters.

Figure 8-1 High-level component of a Communication Service



Communication Service Common

The Communication Service common parts are auto-generated based on one or more WSDLs. For application-initiated requests, these are referred to as service WSDLs, while for network triggered requests, they are referred to as callback WSDL files.

Based on the service WSDLs, the following common parts of a Communication Service are generated using the Eclipse wizard:

- Service Web Service (SOAP or RESTful)
- Service EJB
- Call-back EJB
- Call-back client EJB

The Service Web Service implements the interfaces defined in the set of WSDL files that define the Web Service for application-initiated requests.

The Web Service is packaged into a single WAR file. An example of this is the SOAP Parlay X 2.1 Short Messaging, which defines the following interfaces for application-initiated requests: SendSms, ReceiveSms, and SmsNotificationManager. The Service Web Service implements all the above interfaces and is packaged into one single WAR file for the Communication Service.

The Web Service makes a Java RMI call to the Service EJB which, using the Plug-in Manager, calls the appropriate plug-in instance. The operations defined between the Service Web Service and the Service EJB are Java realizations of the interfaces defined in the service WSDLs. The Service EJB is packaged into a single JAR file for the Communication Service.

The Callback EJB is a Web Services client that uses a Web Service implemented by an application. It uses the interfaces defined in the set of WSDL files that define the Web Service for network-triggered requests, the callback WSDL files. The Web Service client is packaged into a single JAR file for the Communication Service.

The Callback EJB client is a client library that abstracts the remote call between the plug-in POJO and the Callback EJB and provides an invalidating cache of references to the remote object in order to support in-production redeployment of the EAR file for the access tier. The Callback EJB client is packaged into a single JAR file for the Communication Service.

Module	Description	North interface	South interface
Service Web Service	Implements the interfaces defined in the set of WSDL files that define the Web Service for application-initiated requests. Passes on the requests to the Service EJB. Any Service EJB of the same type can be chosen, regardless of the server on which it is deployed. The requests are load-balanced across the different server instances. Packaged into a single WAR file. Deployed as a part of the access tier EAR for the Communication Service. The Service Web Service is transparent to an extension developer.	SOAP/HTTP representation of the Service WSDLs	Java RMI representation of the Service WSDLs
Service EJB	Accepts requests from the Service Web Service implementation and propagates them to the appropriate plug-in using the Plug-in Manager. The Service EJB is responsible for: • Constructing the RequestInfo object. • Converting any exception caught to an exception that is defined in the Service WSDLs. This functionality must be implemented in the PluginFactory class, which extends Class: RequestInfo. Packaged in a single JAR file. Deployed as a part of the network tier EAR.	Java RMI representation of the Service WSDLs	Local Java representation of the Service WSDLs.

Module	Description	North interface	South interface
Callback EJB	A Web Services client that uses a Web Service implemented by an application. Accepts requests from the Service callback client EJB and propagates them to an application. Packaged into a single JAR file for the Communication Service. Deployed as a part of the access tier EAR.	SOAP/HTTP representation of the Service callback WSDLs.	Java RMI representation of the Callback WSDLs
Callback EJB client	A client library that abstracts the remote call between the plug-in and the Callback EJB. Accepts requests from a plug-in and propagates them to the Callback EJB. It provides an invalidating cache of references to the remote object in order to support in-production redeployment of the EAR file for the access tier. Any Callback EJB of the same type can be chosen, regardless of the server on which it is deployed. The requests are load-balanced across the different server instances. See Class: CallbackFactory and Interface: Callback. Packaged into a single JAR file for the Communication Service. Deployed as a part of the network tier EAR.	Java RMI representation of the Service callback WSDLs.	Local Java representation of the Callback WSDLs.

Plug-in

The `com.bea.wlcp.wlng.api.plugin.*` packages contain a range of interfaces and classes for use by the extension developer.

The first of these is a set of interfaces that define the borders of a plug-in and related helper classes. These borders are used to apply aspects. See JavaDoc for `com.bea.wlcp.wlng.plugin`

Plug-in Service and Plug-in Instance

A plug-in service is a JEE application that implements com.bea.wlcp.wlng.api.plugin.ManagedPluginService. It has:

- A life-cycle, defined in com.bea.wlcp.wlng.api.plugin.PluginServiceLifecycle.
- A registry, defined in com.bea.wlcp.wlng.api.plugin.PluginService.
- A factory to create plug-in instances, defined in com.bea.wlcp.wlng.api.plugin.PluginInstanceFactory

The plug-in instance is a class that implements com.bea.wlcp.wlng.api.plugin.ManagedPluginInstance. It has:

- A life-cycle defined in com.bea.wlcp.wlng.api.plugin.PluginInstanceLifecycle.
- A set of PluginNorth and PluginSouth interfaces that it implements. These interfaces are defined by the application-facing interfaces and the network-facing interfaces.
- A registry, defined in com.bea.wlcp.wlng.api.plugin.PluginInstance. This registry holds the list of the registered interfaces.
- Logic that examines the data in a request and determines if the instance can handle it or not. The interface for this logic is defined in com.bea.wlcp.wlng.api.plugin.PluginInstance.
- Logic that maintains the state of a connection. The interface for this logic is defined in com.bea.wlcp.wlng.api.plugin.PluginInstance.

Plug-in routing and registration with the Plug-in Manager is done by the plug-in instance. It is the plug-in instance that is part of the traffic flow.

Life-cycle management is performed on the plug-in service.

States

A plug-in service is in one of a distinct set of states:

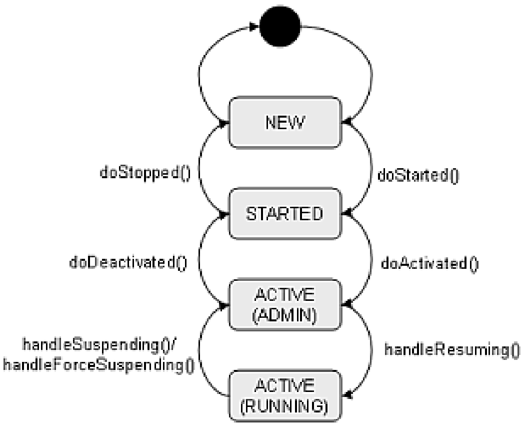
- NEW
- STARTED

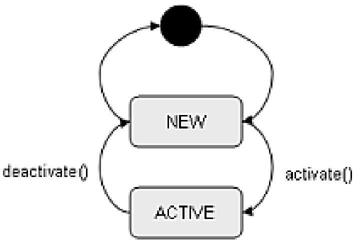
- ACTIVE (ADMIN)
- ACTIVE (RUNNING)

The plug-in instance is in one of the following states:

- NEW
- ACTIVE

Figure 8-2 States for a plug-in service (left) and a plug-in instance (right)





The state transitions in Table 8-1 are triggered by either the start-up sequence of the server in which the plug-in is deployed or by an explicit deployment of the plug-in using either the weblogic.Deployer, see *Oracle WebLogic Server Deploying Applications to WebLogic Server* at http://download.oracle.com/docs/cd/E12840_01/wls/docs103/deployment/, or the administration console.

Note: All deployments are made at the EAR level, which means that individual plug-ins are not targeted, but all plug-ins within the EAR are affected.

Table 8-1 Plug-in service state transitions

Transition	Triggered by	Descriptions
init	Deployment or startup.	The plug-in service has been created and initialized. The only method that will be called in this state is doStarted()
doStarted	Deployment or startup	The plug-in service should perform as much initialization as possible without being externally visible. Examples include: retrieve configuration data, create internal objects, and initialize stores.
doActivated	Deployment or startup	The plug-in service should continue activation and become visible, for example register MBeans, without accepting traffic.
handleResuming	Deployment or startup.	The plug-in service should order all plug-in instances to establish connections with the network node, if applicable, and accept traffic.
handleSuspending	Graceful undeployment/redeployment/stop That is, by invoking weblogic.Deployer with -graceful	The plug-in service should order the plug-in instance to reject new traffic, but to continue processing of in-flight work. A com.bea.wlcp.wlng.api.plugin.CompletionBarrier is provided in the request. When all in-flight work has been processed, the plug-in should get the com.bea.wlcp.wlng.api.plugin.CompletionBarrierCallback from the CompletionBarrier and call completed() on the CompletionBarrierCallback.
handleForceSuspending	Forced undeployment/redeployment/stop That is invoking weblogic.Deployer with -retiretimeout	The plug-in service should order the plug-in instance to reject new traffic and to discard in-flight work.
doDeactivated	Undeployment.	The plug-in service should deactivate itself, unregister any MBeans and become invisible.
doStopped	Undeployment.	The plug-in service should perform cleanup and be available for garbage collection.

The state transitions in Table 8-2 are triggered by either the start-up sequence of the server on which the plug-in instance is created or an explicit creation of a new instance using the Plug-in manager: see Managing and Configuring the Plug-in Manager in the System Administrator's Guide.

Table 8-2 Plug-in instance state transitions

Transition	Triggered by	Descriptions
activate	Creation of the plug-in instance using the Plug-in Manager MBean.	The plug-in instance is created. Depending on the state of the plug-in service, the plug-in instance should take the appropriate action. If the plug-in service is in state: • ACTIVE (ADMIN), the plug-in instance shall: – instantiate and register the PluginNorth and call-back interfaces with the Plug-in Manager. – instantiate and register the PluginSouth interfaces with the Plug-in Manager. – instantiate any ConfigurationStore. – register the MBean for the instance. • ACTIVE (RUNNING), the plug-in shall: – connect to the network node, if a connection-oriented protocol is used. – register call-backs with the network node, if any.
deactivate	Destruction of the plug-in instance using the Plug-in Manager MBean.	The plug-in instance shall: • de-register any call-backs with the network node. • disconnect from the network node, if connected. • de-register the MBean for the instance. • cancel any timers.

The Plug-in Manager maintains a pool of plug-in instances. This pool is provided to the plug-in when init() is called. This pool can be used to iterate over all instances in order to propagate events related to state transitions in the plug-in service.

The Plug-in Manager has a registry of all PluginNorth and PluginSouth interfaces, and it is the responsibility of the plug-in instance to register these interfaces with the Plug-in Manager. The Plug-in Manager uses this list of registered interfaces when routing a request to an appropriate plug-in instance. The Plug-in Manager queries the plug-in instance for information in order to make a routing decision. A plug-in instance maintains:

- A list of PluginNort interfaces
- A list of PluginSouth interfaces
- Whether the plug-in instance has a connection to the network node.
- Custom pattern matching, where the plug-in examines the request and marks the plug-in instance as either a 1) mandatory, 2)optional, or 3) required target for the request.

The plug-in service maintains a:

- Service type, used by all plug-in instances to generate EDRs, CDRs, and Statistics.
- List of supported address schemes, used by the Plug-in Manager when taking a routing decision.

PluginPool

A collection of PluginInstances. The pool is populated when a plug-in instance is created using the PluginInstanceFactory. Using the pool, the plug-in service can:

- List plug-in instances
- Get a plug-in instance by its plug-in instance ID.

Interface: Plugin

Superinterface for Interface: PluginNorth, Interface: PluginNorthCallBack, and Interface: PluginSouth.

PluginNorth defines the entry-point for application-initiated requests and is one of the borders at which aspects are woven. This interface must be implemented by all classes that handle application-triggered requests from the service EJB to the plug-in. There must be one class per interface.

Plugin South defines the entry-point for network-triggered requests. PluginNorthCallback defines the limit between the plug-in and the service callback EJB and further on to an application.These interfaces must be implemented by any plug-in that handles network-triggered requests, either new requests or notifications.

Interface: PluginNorth

All interfaces in the plug-in that implement the traffic interfaces defined in the service WSDLs must implement this interface. A list of the implementations is maintained in the class that implements Interface: ManagedPluginInstance. Statistics aspects are applied for classes that implement this interface and counters for transaction units are increased. See *Oracle Communications Services Gatekeeper Licensing*, section for more information abut transaction units.

Interface: PluginNorthCallBack

All interfaces in the plug-in that implement the traffic interfaces defined in the service callback WSDLs must implement this interface. Statistics aspects are applied for classes that implement this interface and counters for transaction units are increased. See *Oracle Communications Services Gatekeeper Licensing*, section for more information abut transaction units.

Interface: PluginSouth

This interface must be implemented by the plug-in. Defines the south border of a plug-in, that is the network-facing border.

It contains methods used to rebuild the object defined by Interface: RequestContext for network-initiated requests, using information from the object defined by Interface: ContextMapperInfo, and methods for resolving which application instance the request belongs to.

When a network triggered request arrives at the plug-in, the usual pattern is to correlate the request with a previous subscription for notifications.

By extending PluginSouth in the class that implements the request, aspects that call the method

```
public String resolveAppInstanceGroupId(ContextMapperInfo)
```

are applied.

It is the responsibility of the plug-in instance to extract the information provided in the request and to resolve the application instance that matches this data as a part of the rebuilding of the RequestContext. This is done using the Context Aspect.

After resolving the application instance, the method

```
public void prepareRequestContext(RequestContext ctx, ContextMapperInfo
info)
```

is called. In the implementation of this method, the plug-in instance has the option to add additional data to the object defined by Interface: RequestContext.

Interface: ManagedPluginService

This is the interface a plug-in service must implement.

It extends the interfaces PluginService, PluginInstanceFactory and PluginServiceLifecycle.

Interface: PluginService

The interface that defines the plug-in service when registered in the Plug-in Manager.

It defines a set of attributes that must be defined by implementing the following methods:

- `getNetworkProtocol()`, returns a descriptive name for the supported network protocol. For example "SMPP v3.4"
- getServiceType(), returns a ServiceType. See Class: ServiceType.
- getSupportedSchemes(), returns a list of supported address schemes. This is a String array of URI schemes: for example “tel”, “mailto”, and “sip“.

Interface: PluginInstanceFactory

Factory that allows a plug-in service to create plug-in instances.

Defines the method:

```
ManagedPluginInstance createInstance(String pluginInstanceId)
```

The plug-in service is responsible for creating an instance of the class implementing Interface: ManagedPluginInstance when this method is invoked. The method is triggered by the method `createPluginInstance` on the Plug-in Manager MBean.

Interface: PluginServiceLifecycle

Defines the life-cycle for a plug-in service. See States.

Interface: ManagedPluginInstance

Must be implemented by a plug-in instance.

It extends the interfaces PluginInstance and PluginInstanceLifecycle.

Interface: PluginInstance

Defines the plug-in instance when registered in the Plug-in Manager.

The plug-in instance is responsible for:

- maintaining a list of north interfaces that the plug-in implements.
- maintaining a list of south interfaces that the plug-in implements.

Both of these lists are arrays of PluginInterfaceHolder.

The lists shall be returned when `getNorthInterfaces()` and `getSouthInterfaces()` are invoked, respectively.

The plug-in instance is also responsible for implementing `customMatch(RequestInfo requestInfo)`. In this request, the plug-in instance examines the RequestInfo object and decides if the plug-in instance can be used to serve the request. By returning:

- MATCH_OPTIONAL: Indicates that the request can be served by any plug-in instance. The request is completely stateless.
- MATCH_REMOVE: The request cannot be served. This situation can occur, for example, if a plug-in service does not implement the method being invoked or if the request relates to a previous request which is known only to a subset of the plug-in instances in the cluster.
- MATCH_REQUIRED: The request must be served by the plug-in instance. This situation can occur, for example, if the request relates to a previous request which is known only to a subset of the plug-in instances in the cluster.

Only these constants can be returned.

The plug-in instance is also responsible for maintaining information on the connection status with the network node it is connected to by returning True or False when `isConnected()` is invoked.

All methods in this interfaces are invoked by the Plug-in Manager when selecting a plug-in instance to route the request to.

Interface: PluginInstanceLifecycle

Defines the life-cycle for a plug-in service. See States.

Class: RequestFactory

The Request Factory is used to perform application-initiated request processing both before and after a request is processed in the plug-in. Each Communication Service must have one implementation of the RequestFactory per each application-facing interface, named according to the pattern: `<myinterfacename>.PluginFactory`. A skeleton for the factory is generated by the Eclipse plug-in.

The RequestFactory has two main functions:

- Packages routing information contained in the request into a `RequestInfo` object that the Plug-in Manager uses to select an appropriate plug-in to process the request. See below for more information on `RequestInfo` objects.

  Note: In order to support sendlists which target multiple plug-ins, the Request Factory implementation must support three methods that are not required for non-sendlist based plug-ins:

  - `createRequestInfos:` allows the creation of multiple `RequestInfo` objects. Each instance of a `RequestInfo` object is matched to a plugin. For example if an SMS message request is sent to 3 addresses, the factory should create an array of 3 `AddressRequestInfo` objects.
  - `createPartialRequest:` splits a request into multiple requests sent to different plug-ins
  - `mergeResults`: merges the results reported back by multiple plug-ins into a single result.

  For more information, see the RequestFactory JavaDoc

  Note: Plug-ins are invoked in sequence and if one of them fails the whole request is considered a failure. In this case, an exception is thrown and the transaction is rolled back.

- Translates any exceptions thrown in the plug-in (or the underlying network) into a form that can be sent back to the application.

Class: CallbackFactory

This class is used by a plug-in instance to get an implementation of Interface: Callback. There is one CallbackFactory per interface defined in the callback WSDLs.

The naming pattern is `com.acompany.example.callback.<interface name>CallbackFactory`

The implementation of the interface is fetched using the following pattern:

Listing 8-1

```
import com.acompany.example.callback.NotificationCallback;
import com.acompany.example.callback.NotificationCallbackFactory;
...
private NotificationCallback cachedNotificationCallback = null
....
private NotificationCallback getNotificationCallback() {
  if(cachedNotificationCallback == null) {
    cachedNotificationCallback =
    NotificationCallbackFactory.getInstance().create();
  }
  return cachedNotificationCallback;
}
```

Interface: Callback

This interface is used by a plug-in to propagate a network-triggered request from the plug-in to the callback EJB. The interface defines a Java representation of the methods defined in the callback WSDLs. There is one of these per interface defined in the callback WSDLs.

The naming pattern is `com.acompany.example.callback.<interface name>Callback`

Class: RequestInfo

The object created by the RequestFactory to hold information from the application-initiated request. There are four sub-classes of `RequestInfo` that can be used depending on the request:

- `AddressRequestInfo`, if the request contains an address.
- `CorrelatorRequestInfo`, if the request contains a correlator.

- `RegistrationIdentifierRequestInfo`, if the request contains a registration identifier.
- `RequestIdentifierRequestInfo`, if the request contains a request identifier.

Class: ServiceType

This is an abstract utility class that each plug-in must implement. An object of this type is passed to the Plug-in Manager when the plug-in registers itself, so that the Plug-in Manager can query for service type.

Aspects take care of making this service type available in the request thread of each plug-in. The service type is used by various services, including the `EdrService`.

Table 8-3 Existing ServiceTypes

ServiceType	Plugin
AccessServiceType	Access
ThirdPartyCallServiceType	Third-party Call
CallNotificationServiceType	Call Notification
SmsServiceType	Sms
MultimediaMessagingServiceType	Mms
TerminalLocationServiceType	Terminal Location
AudioCallServiceType	Audio Call
PresenceServiceType	Presence

Interface: ContextMapperInfo

This interface defines a `ContextMapperInfo` object. When network-initiated traffic enters the plug-in from the network-facing (south) side, aspects take any annotated arguments from the network call that will be needed by the plug-in for correlation purposes and places them in this very short-lived object. Arguments are stored by key, defined when the annotation is set, that make it possible to retrieve a particular value. So if an argument is annotated with `@MapperInfo(C)`, its value can be retrieved using the key “C”. Methods in the plug-in that need to retrieve these arguments in order to perform a mapping (for example, associating a notification with the session ID of the request that established it) can use this object. The `PluginSouth` interface includes one such method, `resolveAppInstanceGroupdId.`

Interface: RequestContext

Defines a `RequestContext` object. A `RequestContext` object is available in all Communication Services for both application-initiated and network-initiated requests. It contains contextual information about the request, including the service provider account ID, application account ID, and application instance of the application that initiated either the request or the notification, as well as the session ID.

Class: ManagedPlugin

Deprecated. Use ManagedPluginService instead.

Allows the plug-in to register itself in the Plug-in Manager. See Class: AbstractManagedPlugin.

Class: AbstractManagedPlugin

Deprecated. Use ManagedPluginService instead.

Extends `ManagedPlugin`, implements `ServiceDeployable`. It makes the plug-in deployable as a service in Oracle Communications Services Gatekeeper, and assists in registering the plug-in with the Manager. See the `com.bea.wlcp.wlng.api.plugin.common` package JavaDoc for details.

Management

These are base classes and annotations for giving the Oracle Communications Services Gatekeeper Management Console or other JMX tools management access to Communication Services. See Chapter 10, “Making Communication Services Manageable” for more information. Also see the JavaDoc for the packages: `com.bea.wlcp.wlng.api.management.*`

SLA Enforcement

SLA enforcement operates on methods identified by the Java representation of the interface, and the operation of the application-facing interface for the Communication Service

The content of the tag <scs> defined in the <serviceContract> tag in the SLA is the plug-in type for the plug-in.

An operation on the application-facing interface is represented in the rules according to the following scheme: <service name> and <operation name>.

Parameters in the operation are represented in the rules according to the following scheme:

arg<n>.<parameter name>

where <n> in arg<n> depends on the WSDL that defines the application-facing interface, normally this is arg0.

If the parameter in <parameter name > is

- a composed parameter, the notation is according to the Java Bean notation for that parameter.
- an enumeration, the notation is according to the Java-representation of that parameter, <parameter name >.<enumeration value>. The <enumeration value> is the String representation. See Using the Platform Test Environment for information about the SLA Editor.

Shared libraries

It is possible for multiple plug-ins to share common libraries: for example, a third party library or custom code that can be shared.

If there are such parts, these should preferably not be packaged into the plug-in jar but instead be copied into the `APP-INF/lib` directory of the Communication Service EARs that utilizes this shared library. All jars in this directory are available for each of the plug-ins in the EAR.

CHAPTER 9

Annotations, EDRs, Alarms, and CDRs

The following section describe aspects and generation of EDRs, alarms, CDRs, and statistics:

About aspects and annotations

Aspects allow developers to manage cross-cutting concerns in their code in a straight forward and coherent way. Aspects in Oracle Communications Services Gatekeeper (pointcuts, advice, etc.) are written in the AspectJ 1.5.3 annotation style. There is already support for editing annotations in many modern IDEs, and aspects are simply set up as annotated classes.

How aspects are applied

All aspects are applied at build time by weaving the byte code of previously complied Java packages. Minimal reflection is used at runtime to make aspect based decisions.

Different aspect types are applicable to different Oracle Communications Services Gatekeeper modules. In general there are two categories of aspects:

- Those restricted to the code for the traffic flow
- Those that can be applied to other packages.

Note: In this case, traffic flow is defined to include only plug-in implementations.

Traffic aspects are subdivided into two categories:

- Those that are always applied
- Those that are controlled using annotations.

Only statistics aspects are always applied because they are used to calculate usage costs. Traffic aspects are applied to North and South boundaries of a plug-in as well as to the internal processing of the plug-in.

Annotations are used to control the aspects that are not always applied for each plug-in. These annotations are defined as part of the functional areas that a given set of aspect implements. They allow the plug-in to communicate with the aspects as well as to customize their behavior.

Context Aspect

The Context aspect is woven at compile time, using PluginSouth as a marker.

While requests coming from the north interface have a valid context (with attributes like Service provider account ID, application Account ID, and so on) any events triggered by the network and entering a plug-in's south interface do not have a valid context.

The Context aspect solves this problem by rebuilding the context as soon as a south interface method is invoked: after this aspect is executed, a valid context will be available for any subsequent usages, such as the EDR aspect. All methods inside a class implementing the interface PluginSouth are woven by the Context aspect.

The Context aspect requires the following in order to correctly weave the south interface methods and be able to rebuild the context:

- Each Plugin must explicitly register its north and south interfaces.
- Each south interface must implement the `resolveAppInstanceGroupdId()` and `prepareRequestContext()` methods of the PluginSouth interface.
- North interfaces must implement PluginNorth and south interfaces must implement PluginSouth.

The following rules apply for methods in classes that implement PluginNorth:

- The default behavior is that EDRs are triggered only for exceptions and callbacks to EJBs in the access tier (Service Callback EJB)
- If a method is annotated with @NoEdr, no EDRs will be generated. It overrides the default behavior.
- If a method is annotated with @EDR, 2 EDRs will be generated:
  - When entering the method
  - When exiting the method.

The following rule applies for methods in classes that implement PluginSouth:

- Methods that perform requests to the network may have a parameter annotated with @MapperInfo in order to be able to rebuild the RequestContext when the response to the request arrives from the network. The annotated parameter must be used as a key to resolve the application instance ID using some plug-in specific lookup.

- Methods must implement `resolveAppInstanceGroupdId(ContextMapperInfo info)` in PluginSouth and return the application instance ID that corresponds to the original request to the network.

The ways of doing this are plug-in-specific, but normally a network triggered request is tied to an application instance in a store that is managed by the plug-in. The store used for context mapping may be a local cache or a cluster wide store, depending on whether responses are known to always arrive on the same plug-in instance, or if they can arrive at a plug-in on another server in the cluster.

Example:

1. An application sends a request to the network and an ID for this request is either supplied by the network or generated by the plug-in. At this point the originator of the requests, the application instance, is known since the request originated from an application.
2. The plug-in puts the application instance ID and the ID for the request into a store.
3. At a later stage, when a response to the original requests arrives at the plug-in, the method `resolveAppInstanceGroupId()` is called by aspects.
4. In this method, the plug-in must perform a lookup in the store of the application instance related to that request and return the application instance ID to the aspect.
5. The aspect authenticates the application instance with the container and puts the application instance ID in the RequestContext.
6. The method in the plug-in receives the request from the network and the RequestContext contains the application instance ID.

In the example below the method `deliver(...)` is a request from the underlying network. The destinationAddress is annotated to be available to the aspect that handles network-triggered requests associated with this request, represented by constant C.

NotificationHandler handles the store for notifications and supplies all necessary parameters to the store.

Listing 9-1 Application initiated request

```
protected static final String C = "destinationAddress";

@Edr

  public void deliver(String data,
                      @ContextKey(EdrConstants.FIELD_DESTINATION_ADDRESS)

                      @MapperInfo(C) String destinationAddress,

                    @ContextKey(EdrConstants.FIELD_ORIGINATING_ADDRESS) String
originatingAddress,

                      String nwTransactionId)

    throws Exception {


    notificationHandler.deliver(data, destinationAddress, originatingAddress,
nwTransactionId);


  }
```

When a network triggered event occurs, the aspect calls `resolveApplicationInstanceGroup(...)` in PluginSouth and the plug-in looks up the application instance using any argument available in ContextMapperInfo that can help the plug-in to resolve this ID from ContextMapperInfo, using `info.getArgument(C)`. The application instance ID is returned to the aspect and the execution flow continues in the plug-in, with a RequestContext that contains the application instance ID, session ID and so on.

Listing 9-2 Rebuilding RequestContext

```
protected static final String C = "destinationAddress";

public String resolveAppInstanceGroupdId(ContextMapperInfo info) {


    String destinationAddress = (String) info.getArgument(C);

    NotificationData notificationData = null;

    try {

      notificationData =
StoreHelper.getInstance().getNotificationData(destinationAddress);

    } catch (StorageException e) {
      return null;
    }

    if (notificationData == null) {
      return null;
    }

    return notificationData.getAppInstanceGroupId();
  }
```

Below are the steps you have to take to make your plug-in compliant with the Context aspect:

- Make sure to register all your PluginSouth objects before registering your plug-in in the Plug-in Manager.
- Make sure to implement the `resolveAppInstanceGroupdId()` method for each PluginSouth instance.
- Annotate each parameter in south object methods that you need to have when aspects call back the `resolveAppInstanceGroupId()` or the `prepareRequestContext()` methods. All the annotated parameters will be available in the ContextMapperInfo parameter. The aspects need to have them annotated to be able to store them into the ContextMapperInfo object.

EDR Generation

EDRs are generated in the two following ways:

- automatically using aspects at given points in the traffic execution flow in a plug-in.
- manually anywhere in the code using the EdrService.

EDRs should be generated at the plug-in boundaries (north and south), using the @Edr annotation to ensure that the boundaries are covered. Additional Edrs can be added elsewhere in the plug-in if needed: for example for CDRs.

For extensions, the EDR ID should be in the range 500 000 to 999 999.

EDRs are generated automatically by an aspect in the following locations in the plug-in:

- Before and after any method annotated with @Edr
- Before and after any callback to an EJB
- After any exception is thrown

Note: Note that aspects are not applied outside the plug-in.

Table 9-1 Manual annotation for EDRs

Trigger	When	Modifiers restrictions	What is woven
method	before executing	public method only	only in methods annotated with @Edr
method	after executing	public method only	only in methods annotated with @Edr
method-call	before calling	any method	only for method call to a class implementing the PluginNorthCallback interface (EJB callback)
method-call	after calling	any method	only for method call to a class implementing the PluginNorthCallback interface (EJB callback)
exception	after throwing	any method	any exception thrown except in methods annotated with @NoEdr

The following values are always available in an EDR when it is generated from an aspect:

- class name
- method name
- direction the request is going toward (south, north)
- position (before, after)
- interface (north, south, other, null)
- source (method, exception)

Exception scenarios

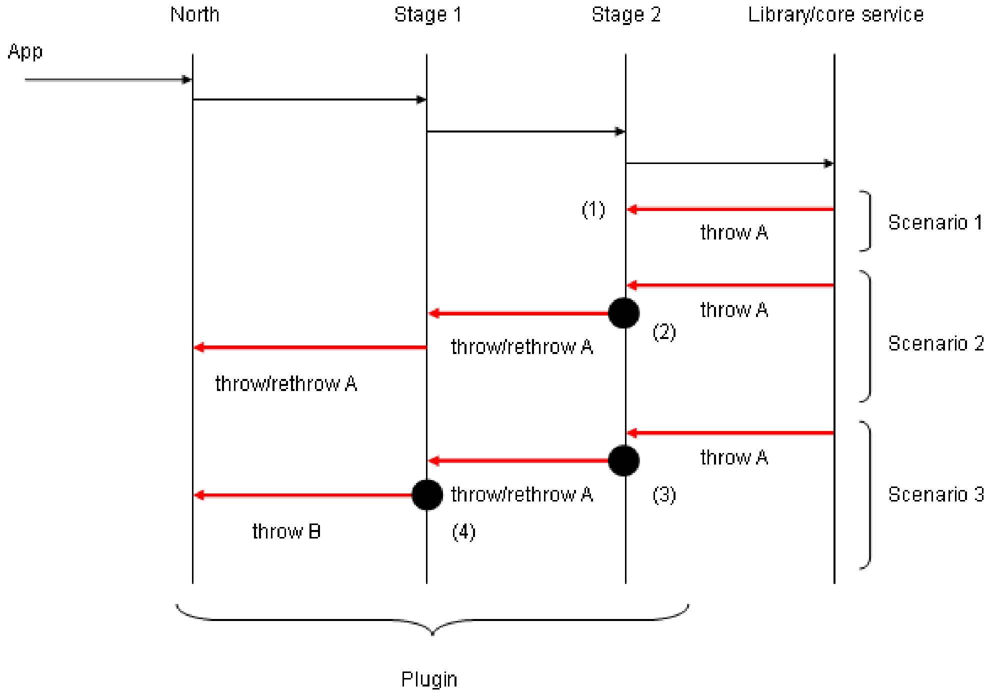
Exceptions are automatically woven by the aspect.

Some limitations apply:

- The aspect will catch only exceptions that are thrown by a plug-in method.
- The aspect will not catch an exception that is thrown by a library and caught by the plug-in.
- If the same exception is re-thrown several times, the aspect will only trigger an EDR once, for the first instance of the exception.

The diagram illustrates typical scenarios when a library (or core service) throws an exception in the plug-in.

Figure 9-1 Exception scenarios



Scenario 1:

The plug-in method in Stage 2 simply catches the exception but does not re-throw it or throw another exception. Since it just consume the exception, the aspect will not trigger an EDR.

Scenario 2:

The plug-in method in Stage 2 lets the exception A propagate (or re-throws exception A).

In this case, the aspect triggers an EDR after the method in stage 2. Since the same exception A (the same exception instance object) is propagated (or re-thrown), only the first method triggers an EDR.

Scenario 3:

This scenario is almost identical to scenario 2 except that the method in stage 1 is not throwing the exception A but another exception, named B. In this case, because B is not the same instance as A, the aspect will trigger another EDR after the method in stage 1.

Adding data to the RequestContext

In addition to the default values, an EDR also contains all the values put into the RequestContext using the `putEdr()` method.

Listing 9-3 Example to add values to and EDR using RequestContex

```
...
RequestContext ctx = RequestContextManager.getCurrent();
// this value will be part of any EDRs generated in the current request
ctx.putEdr("address", "tel:1234");
// this value will NOT be part of any EDRs since ctx.put(...) is used
ctx.put("foo", "bar");
...
```

Note: Common key names are defined in the class com.bea.wlcp.wlng.api.edr.EdrConstants.

Using translators

When a parameter is a more complex object, it is possible to specify a translator that will take care of extracting the relevant information from this parameter.

The annotation is @ContextTranslate.

For example, the following method declares:

- The first (and only) parameter should be translated using the specified translator ACContextTranslator
- The returned object should also be translated using the specified translator ACContextTranslator

Listing 9-4 Using a translator

```
...

  @Edr

  public @ContextTranslate(ACContextTranslator.class) PlayTextMessageResponse
playTextMessage(@ContextTranslate(ACContextTranslator.class) PlayTextMessage
parameters) {

    ...

    return response;

  }

  ...
```

The Translator is a class implementing the ContextTranslator interface.

Listing 9-5 Example of a Translator

```
  public class ACContextTranslator implements ContextTranslator {

    public void translate(Object param, ContextInfo info) {

      if(param instanceof PlayTextMessage) {

        PlayTextMessage msg = (PlayTextMessage) param;
        info.put("address", msg.getAddress().toString());
      } else if(param instanceof PlayTextMessageResponse) {
        PlayTextMessageResponse response = (PlayTextMessageResponse) param;
        info.put("correlator", response.getResult());
      } ...
    }
  }
```

The ContextTranslator class specified in the @ContextTranslate annotation is automatically instantiated by the aspect when needed. It is however possible to explicitly register it using the ContextTranslatorManager.

Listing 9-6 Example of registering a context translator

```
ContextTranslatorManager.register(ACContextTranslator.class.getName(), new
ACContextTranslator());
```

Below is a summary of annotations to use.

Table 9-2 Annotations

Name	Type	Description
@ContextKey	Annotation	Specifies that an argument must be put into the current RequestContext under the name provided in this annotation
@ContextTranslate	Annotation	Same as @ContextKey but for complex argument that need to be translated using a translator (implementing the ContextTranslator interface).
ContextTranslator	Interface	Interface used by static translators to translate complex object.

Trigger an EDR programmatically

Oracle Communications Services Gatekeeper triggers EDRs automatically in all plug-ins where aspects have been applied. It is also possible to trigger EDRs explicitly. In this case, you will have to manually create and trigger the EDR by following these steps:

1. Create an EdrData object
2. Trigger the EDR using the EdrService instance

Below is an example of triggering an EDR from inside a plug-in.

Listing 9-7 Trigger an EDR programmatically

```
public class SamplePlugin {

   // Get the EdrDataHelper like a logger

   private static final EdrDataHelper helper =
EdrDataHelper.getHelper(SamplePlugin.class);


   public void doSomething() {

      ...

      // Create a new EdrData using the EdrDataHelper class to allow

      // Services Gatekeeper to automatically populate some fields

      EdrData data = helper.createData();

      // Since we are creating the EdrData manually,

      // we have to provide the mandatory fields.

      // Note that the EdrDataHelper will provide most of them

      data.setValue(EdrConstants.FIELD_SOURCE,
EdrConstants.VALUE_SOURCE_METHOD);

      data.setValue(EdrConstants.FIELD_METHOD_NAME, "doSomething");

      // Log the EDR

      EdrServiceFactory.getService().logEdr(data);

      ...
```

```
    }
}
```

EDR Content

The following table describes the content of an EDR. It describes which values are mandatory, who is responsible for providing these values, and other information.

Legends:

- A: Automatically provided by Oracle Communications Services Gatekeeper
- H: Provided if the EdrDataHelper createData API is used to create the EdrData (which is the recommended way)
- M: Provided manually in the EdrData
- X: Provided in the EDR descriptor.
- C: Custom filter. Use the element <attribute> to specify a custom filter.

Note: EDRs triggered by aspects will have all the mandatory fields provided by the aspect.

Table 9-3 EDR content

Name	Description	Filter tag name
EdrId	To get the ID, use `getIdentifier()` in EdrConfigDescriptor. This value is provided in the EDR descriptor. Provider INSIDE plug-in: X Provider OUTSIDE plug-in: X Mandatory: Yes	C
ServiceName	The name (or type) of the service. Fields in EdrConstants: FIELD_SERVICE_NAME Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: Yes	C
ServerName	The name of the Oracle Communications Services Gatekeeper server. Fields in EdrConstants: FIELD_SERVER_NAME Provider INSIDE plug-in: H Provider OUTSIDE plug-in: H Mandatory: Yes	C

Table 9-3 EDR content

Name	Description	Filter tag name
Timestamp	The time at which the EDR was triggered (in ms since midnight, January 1, 1970 UTC) Fields in EdrConstants: FIELD_TIMESTAMP Provider INSIDE plug-in: A Provider OUTSIDE plug-in: A Mandatory: Yes	C
ContainerTransactionId	The WebLogic Server transaction ID, if available. Fields in EdrConstants: FIELD_CONTAINER_TRANSACTION_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: H Mandatory: No	C
Class	Name of the class that triggered the EDR. Fields in EdrConstants: FIELD_CLASS_NAME Provider INSIDE plug-in: H Provider OUTSIDE plug-in: H Mandatory: Yes	<class>
Method	Name of the method that triggered the EDR. Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: Yes	<name> inside <method> or <method> inside <exception>

Table 9-3 EDR content

Name	Description	Filter tag name
Source	Indicates the type of source that triggered the EDR. Fields in EdrConstants: FIELD_SOURCE Values in EdrConstants: VALUE_SOURCE_METHOD, VALUE_SOURCE_EXCEPTION Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: Yes	<method> or <exception>
Direction	Direction of the request. Fields in EdrConstants: FIELD_DIRECTION Values in EdrConstants:VALUE_DIRECTION_SOUTH, VALUE_DIRECTION_NORTH Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	<direction>
Position	Position of the EDR relative to the method that triggered the EDR. Fields in EdrConstants: FIELD_POSITION Values in EdrConstants: VALUE_POSITION_BEFORE, VALUE_POSITION_AFTER Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	<position>

Table 9-3 EDR content

Name	Description	Filter tag name
Interface	Interface where the EDR is triggered. Fields in EdrConstants: FIELD_INTERFACE Values in EdrConstants: VALUE_INTERFACE_NORTH, VALUE_INTERFACE_SOUTH, VALUE_INTERFACE_OTHER Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	<interface>
Exception	Name of the exception that triggered the EDR. Fields in EdrConstants: FIELD_EXCEPTION_NAME Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	<name> inside <exception>
SessionId	Session ID. Fields in EdrConstants: FIELD_SESSION_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No	C

Table 9-3 EDR content

Name	Description	Filter tag name
ServiceProviderId	Service provider account ID. Fields in EdrConstants: FIELD_SP_ACCOUNT_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No	C
ApplicationId	Application account ID. Fields in EdrConstants: FIELD_APP_ACCOUNT_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No	C
AppInstanceGroupId	Application instance ID. Fields in EdrConstants: FIELD_APP_INSTANCE_GROUP_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No.	C
OrigAddress	The originating address with scheme included (for example “tel:1234”). Fields in EdrConstants: FIELD_ORIGINATING_ADDRESS Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	C

Table 9-3 EDR content

Name	Description	Filter tag name
DestAddress	The destination address(es) with scheme included (For example “tel:1234”). See Using send lists. Fields in EdrConstants: FIELD_DESTINATION_ADDRESS Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	C
<custom>	Any additional information put into the current RequestContext using the putEdr() API will end up in the EDR. Fields in EdrConstants: - Provider INSIDE plug-in: - Provider OUTSIDE plug-in: - Mandatory: No	C

Using send lists

If more than one address needs to be stored in the DestAddress field, use the following pattern. Both patterns described below can be used.

Listing 9-8 Pattern to store one single or multiple addresses in field destination directly on EdrData.

```
EdrData data = ...;
// If there is only one address
data.setValue(EdrConstants.FIELD_DESTINATION_ADDRESS, address);
// If there are multiple addresses
data.setValues(EdrConstants.FIELD_DESTINATION_ADDRESS, addresses);
```

If you are using the current RequestContext object, simply store a List of addresses. The EdrDataHelper will automatically take care of converting this to a List of Strings in the EdrData.

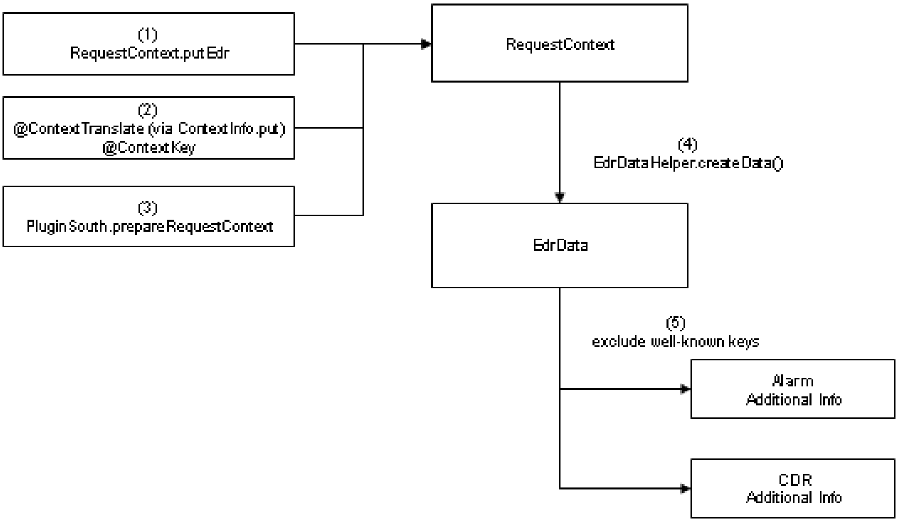
Listing 9-9 Pattern to store one single or multiple addresses in field destination using RequestContext.

```
RequestContext ctx = RequestContextManager.getCurrent();
// If there is only one address
ctx.putEdr(EdrConstants.FIELD_DESTINATION_ADDRESS, address);
// If there are multiple addresses
URI[] addresses = ...;
ctx.putEdr(EdrConstants.FIELD_DESTINATION_ADDRESS, Arrays.asList(addresses));
```

RequestContext and EDR

The following diagram shows how and where information for the EDR is added to the RequestContext and how it finally ends up in the additional info column of the alarm and CDR databases.

Figure 9-2 RequestContext and EDR



There are 3 ways of putting information in the RequestContext that will end up in the EDR (more precisely in the EdrData object):

- Using the `putEdr()` API of the RequestContext
- Using the @ContextKey or @ContextTranslate annotation. In the case of the @ContextTranslate annotation, the information that will end up in the RequestContext will be what is put into the ContextInfo object.
- Any information put in the RequestContext parameter of the PluginSouth.`prepareRequestContext()` method.

When an EDR is created, the EdrDataHelper (which is the recommended way to create the EDR) will populate the EdrData with all the key/value pairs found in the RequestContext.

When the EdrService writes the alarm or CDR additional information content into the database, it will use all the EdrData key/value pairs EXCEPT a set of well-known keys that are either not relevant or already included in other columns of the database, see "Alarm content" on page 9-39 and "CDR content" on page 9-42.

Categorizing EDRs

Only one type of EDR exists: alarms and CDRs are subsets of this EDR type. In order to categorize the flow of EDRs as either pure EDRS, alarms or CDRs, the EDR service uses 3 descriptors:

- The EDR descriptor contains descriptors that describe pure EDRs.
- The alarm descriptor contains descriptors that describe EDRs that should be considered alarms.
- The CDR descriptor contains descriptors that describe EDRs that should be considered CDRs.

These XML descriptors can be manipulated using the **EDR Configuration Pane** as described in Managing and Configuring EDRs, CDRs and Alarms in the *System Administrator's Guide*. File representations of these must be included in edrjmslistener.jar if you are using external EDR listeners.

The EDR descriptor

Each descriptor contains a list of EDR descriptors that define an EDR as a pure-EDR, as an alarm or as a CDR.

Table 9-4 EDR descriptors.

Descriptor	Descriptor	Description
EDR	<edr...>	Defines which EDRs are pure EDRs
Alarm	<alarm...>	Defines which EDRs are alarms
CDR	<cdr...>	Defines which EDRs are CDRs

The descriptor is composed of two parts:

- The <filter> element: this is the filter
- The <data> element: this part is used to attach additional data with the EDR if it is matched by the <filter> part

Table 9-5 describes the elements allowed in the <filter> part:

Table 9-5 Elements allowed in <filter> part of an EDR descriptor.

Source	Filter	Min occurs	Max occurs	Description
<method>		0	unbounded	Filter EDR triggered by a method
	<name>	0	unbounded	Name of the method that triggered the EDR
	<class>	0	unbounded	Name of the class that triggered the EDR
	<direction>	0	2	Direction of the request
	<interface>	0	3	Interface where the EDR has been triggered
	<position>	0	2	Position relative to the method that triggered the EDR
<exception>		0	unbounded	Filter EDR triggered by an exception
	<name>	0	unbounded	Name of the exception that triggered the EDR
	<class>	0	unbounded	Name of the class where the exception was thrown
	<method>	0	unbounded	Name of the method where the exception was thrown
	<direction>	0	2	Direction of the request
	<interface>	0	3	Interface where the EDR has been triggered
	<position>	0	2	Position relative to the method that triggered the EDR
<attribute>		0	unbounded	Filter EDR by looking at custom attribute

Table 9-5 Elements allowed in <filter> part of an EDR descriptor.

Source	Filter	Min occurs	Max occurs	Description
	<key>	1	1	Name of the key
	<value>	1	1	Value

Table 9-6 describes the values allowed for each element of the <filter> part:

Table 9-6 Values allowed in each element of the <filter> part.

Source	Filter	Allowed values	Comment
<method>	<name>	"returntype nameofmethod([args])"	Method name. The arguments can be omitted with the parenthesis. See Special characters below.
	<class>	"fullnameofclass"	Fully qualified class name. See Special characters below.
	<direction>	"south", "north"	
	<interface>	"north", "south", "other"	
	<position>	"before", "after"	
<exception >	<name>	"fullnameofexceptionclass"	Fully qualified exception class name. See Special characters below.
	<class>	"fullnameofclass"	Fully qualified class name where the exception was triggered. See Special characters below.
	<method>	"returntype nameofmethod([args])"	Method name. The arguments can be omitted with the parenthesis See Special characters below.
	<direction>	"south", "north"	

Table 9-6 Values allowed in each element of the <filter> part.

Source	Filter	Allowed values	Comment
	<interface>	“north”, “south”, “other”	
	<position>	“before”, “after”	
<attribute>	<key>	“astring”	
	<value>	“astring”	

Special characters

The filter uses special characters to indicate more precisely how to match certain values.

Using * at the end of a method, class or exception name matches all names that match the string specified prior to the * (that is, what the string starts with).

Note: The usage of any of these characters disables the caching of the filter containing them. To avoid a performance hit, using the other way of matching is strongly encouraged.

Table 9-7 Example filters

To match on	Use the filter

All sendInfoRes methods with one argument of type int.	<method>
	  <name>void sendInfoRes(int)</name>
	  ...
	 </method>

All methods starting with sendInfoRes regardless of the arguments.	<method>
	  <name>void sendInfoRes</name>
	  ...
	 </method>

Table 9-7 Example filters

To match on	Use the filter

All methods starting with void sendInfo.	<method>
	  <name>void sendInfo*</name>
	  ...
	 </method>

All class names beginning with com.bea.wlcp.wlng.plugin	<method>
	  <class>com.bea.wlcp.wlng.plugin*</class>
	  ...
	 </method>

Values provided

The exact value in these fields depends on who triggered the EDR. If the aspect triggered the EDR, then the name of the method (with return type and parameters) or the fully qualified name of the class/exception is indicated. If the EDR is manually triggered from the code, it is up to the implementer to decide what name to use. Here are some examples of fully qualified method/class names as specified by the aspect:

Example methods:

```
SendSmsResponse sendSms(SendSms)
```

```
void receivedMobileOriginatedSMS(NotificationInfo, boolean,
SmsMessageState, String, SmsNotificationRemote)
```

```
TpAppMultiPartyCallBack reportNotification(TpMultiPartyCallIdentifier,
TpCallLegIdentifier[], TpCallNotificationInfo, int)
```

Example Class:

```
com.bea.wlcp.wlng.plugin.sms.smpp.SMPPManagedPluginImpl
```

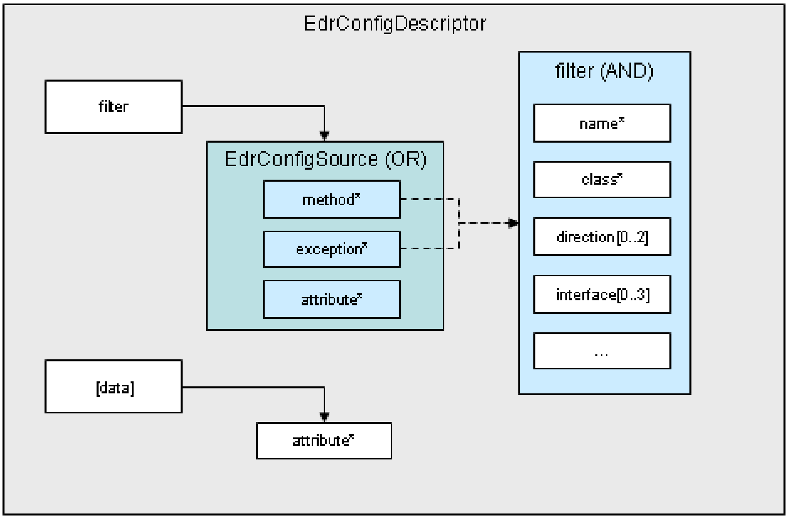
Boolean semantic of the filters

The following diagram shows briefly how the filter works:

- The EdrConfigSource elements are the following: <method>, <exception> or <attribute>. They are combined using OR.

- The filter elements of each EdrConfigSource are combined using AND. However, if the same filter is available more than once (e.g. multiple class names), they are combined with OR.

Figure 9-3 Filter mechanism



Example filters

Example 1: filter

Listing 9-10 categorizes EDRs as pure EDRs with an id of 1000 when the following conditions are met:

- The class where the method triggered the EDR is com.bea.wlcp.wlng.plugin.AudioCallPlugin or any subclass of it.
- AND the request is southbound (direction = south)
- AND the interface where the EDR was trigger is north
- AND the EDR has been triggered after the method has been executed (position = after)

Listing 9-10 Example 1: filter

```
<edr id="1000" description="...">
    <filter>
      <method>
        <class>com.bea.wlcp.wlng.plugin.AudioCallPlugin</class>
        <direction>south</direction>
        <interface>north</interface>
        <position>after</position>
      </method>
    </filter>
  </edr>
```

Example 2: Alarm filter

Listing 9-11categorizes EDRs as alarms when the following conditions are met:

- The exception is the class com.bea.wlcp.wlng.plugin.PluginException or a subclass of it.
- OR the name of the exception starts with org.csapi.*. Since "'*" is used, the matching will not be performed using the class hierarchy but only using a pure string matching.

The alarms descriptor has a <alarm-group> element that is used to group alarms by service/source: this group id and each individual alarm id is used to generate the OID of SNMP traps.

Listing 9-11 Example 2: filter

```
<alarm-group id="104" name="parlayX" description="Parlay X alarms">>
<alarm id="1000" severity="minor" description="Parlay X exception">
    <filter>
      <exception>
        <name>com.bea.wlcp.wlng.plugin.PluginException</name>
```

```
          <name>org.csapi*</name>
        </exception>
      </filter>
    </alarm>
  </alarm-group>
```

Example 3: Alarm filter

Listing 9-12 categorizes EDRs as alarms when the following conditions are met:

- The exception is the class com.bea.wlcp.wlng.plugin.PluginException or a subclass of it
- OR the name of the exception starts with “org.csapi”. String matching in used.
- AND the exception was triggered in a class whose name starts with com.bea.wlcp.wlng.plugin
- AND the request is northbound (direction = north) when the exception was triggered

If the filter determines that the EDR is an alarm, the following attributes are available to the alarm listener (they are defined in the <data> part):

- identifier = 123
- source = wlng_nt1

Listing 9-12 Example 3: filter

```
<alarm id="1000" severity="minor" description="Parlay X exception">
    <filter>
      <exception>
        <name>com.bea.wlcp.wlng.plugin.PluginException</name>
        <name>org.csapi*</name>
        <class>com.bea.wlcp.wlng.plugin*</class>
        <direction>north</direction>
      </exception>
    </filter>
    <data>
      <attribute key="identifier" value="123"/>
      <attribute key="source" value="wlng_nt1"/>
    </data>
  </alarm>
```

Example 4: filter

Listing 9-13 (for example purposes only) categorizes EDRs as pure EDRs with the id 1002 when the following conditions are met:

- The name of the method that triggered the EDR starts with “void play” AND the class is com.bea.wlcp.wlng.plugin.AudioCallPluginNorth or a subclass of it AND the EDR was triggered after executing this method.
- OR the name of the method that triggered the EDR is “String getMessageStatus” AND the class is 'com.bea.wlcp.wlng.plugin.AudioCallPluginNorth' or a subclass of it AND the EDR was triggered before executing this method.
- OR the name of the exception that triggered the EDR starts with com.bea.wlcp.wlng.bar AND the exception was triggered in a plug-in north interface
- OR the name of the exception that triggered the EDR starts with com.bea.wlcp.wlng.plugin.exceptionA AND the exception was triggered in a class whose name starts with com.bea.wlcp.wlng.plugin.classD AND the exception was triggered in a method whose name starts with void com.bea.wlcp.wlng.plugin.methodA AND the exception was triggered in a plugin north interface
- OR the EDR contains an attribute with key attribute_a and value value_a
- OR the EDR contains an attribute with key attribute_b and value value_b

Listing 9-13 Example 4: filter

```
<edr id="1002">
```

```
  <filter>
    <method>
      <name>void play*</name>
      <class>com.bea.wlcp.wlng.plugin.AudioCallPluginNorth</class>
      <position>after</position>
    </method>
    <method>
      <name>String getMessageStatus</name>
      <class>com.bea.wlcp.wlng.plugin.AudioCallPluginNorth</class>
      <position>before</position>
    </method>
    <exception>
      <name>com.bea.wlcp.wlng.bar*</name>
      <interface>north</interface>
    </exception>
    <exception>
      <name>com.bea.wlcp.wlng.plugin.exceptionA</name>
      <class>com.bea.wlcp.wlng.plugin.classD</class>
      <method>void com.bea.wlcp.wlng.plugin.methodA</method>
      <interface>north</interface>
    </exception>
    <attribute key="attribute_a" value="value_a"/>
    <attribute key="attribute_b" value="value_b"/>
  </filter>
</edr>
```

Example 5: filter with corresponding code for manually triggering a matching EDR

Listing 9-14 shows a manually triggered EDR with its corresponding filter. The EDR is triggered using these lines.

Listing 9-14 Example 5: Trigger the EDR

```
  // Declare the EdrDataHelper for each class

  private static final EdrDataHelper helper =
EdrDataHelper.getHelper(MyClass.class);


  public void myMethodName() {

    ...

    // Create a new EdrData. Use the EdrDataHelper class to allow Services
Gatekeeper to automatically populate some fields

    EdrData data = helper.createData();


    // Because we are creating the EdrData manually, we have to provide the
mandatory fields

    data.setValue(EdrConstants.FIELD_SOURCE, EdrConstants.VALUE_SOURCE_METHOD);

    data.setValue(EdrConstants.FIELD_METHOD_NAME, "myMethodName");

    data.setValue("myKey", "myValue");


    // Log the EDR

    EdrServiceFactory.getService().logEdr(data);

    ...

  }
```

This EDR can be filtered using Listing 9-15 (note the various ways of identifying this EDR):

Listing 9-15 Example: Filter 5

```
  <edr id="1003">
    <filter>
      <!-- Match both method name and class name -->
      <method>
        <name>myMethodName</name>
        <class>com.bea.wlcp.wlng.myClassName</class>
      </method>
      <!-- OR match only the method name (looser than matching also the class
name) -->
      <method>
        <name>myMethodName</name>
      </method>
      <!-- OR match only the classname (looser than matching also the method
name) -->
      <method>
        <class>com.bea.wlcp.wlng.myClassName</class>
      </method>
      <!-- OR match only the custom attribute -->
      <attribute key="myKey" value="myValue"/>
    </filter>
  </edr>
```

Check-list for EDR generation

Below is a list of steps to take to make your plug-in able to use aspect EDRs:

- Make sure to register all your PluginNorth (and south) objects within the ManagedPlugin before registering in the PluginManager.

- Annotate all the methods you want to be woven using the @Edr annotation.
- Annotate the specific arguments you want to see in the EDR for each annotated methods. Use either @ContextKey or @ContextTranslate depending on the kind of argument.
- Add to the EDR descriptor all the EDRs you are triggering, either manually or with the @Edr annotation. This is the only way to customize alarms and CDRs.
- If external EDR listeners, CDR, and alarms are used, the file `edrjmslistener.jar` needs to be updated on all the listeners. Add the contents of the EDR descriptors to edr.xml, CDR descriptor to cdr.xml, and alarm descriptor to alarm.xml. The xml files resides in the directory `edr` in `edrjmslistener.jar`.

Frequently Asked Questions about EDRs and EDR filters

Question: Is it possible to specify both exception and method name in the filter section?

Listing 9-16 Example: method name and exception in a filter.

```
<filter>

      <method>

        <name>internalSendSms</name>

      </method>

      <exception>


<name>com.bea.wlcp.wlng.plugin.sms.smpp.TooManyAddressesException</name>

      </exception>

    </filter>
```

Answer

Yes, make sure that the <method> element is before the <exception> element. Otherwise the XSD will complain.

Q: Is it possible to specify multiple method names?

Answer

Yes.

Q: In some places I have methods re-throwing an exception. Is it possible to have only one of the methods generate the EDR and map that edr to an alarm?

Listing 9-17 Re-throwing an exception

```
myMethodA()throws MyException{
  myMethodB();
}

myMethodB()throws MyException{
  myMethodC();
}

myMethodC()throws MyException{
  ...
  //on error
 throw new MyException(“Exception text..”);
}
```

Answer

In this case, only the first exception will be caught by aspects. Or more precisely, they will all be caught by aspects but will only trigger an EDR for the first one, but not for the re-thrown ones (if they are the same, of course). So you don’t need to use the @NoEdr annotation for myMethodA and myMethodB.

Q: Will aspects detect the following exception?

Listing 9-18 Example exception

```
try{
  throw new ReceiverConnectionFailureException(message);
}catch(ReceiverConnectionFailureException connfail){
  //EDR-ALARM-MAPPING
}
```

Answer

This exception will NOT be detected by aspects. If you need to generate an EDR you will have to either manually create an EDR or call a method throwing an exception.

Q: Will EDRs for exceptions also work for private methods?

Answer

Yes, EDRs can work for any method.

Q: Will exceptions be disabled with the @NoEdr annotation?

Answer

Yes, with the @NoEdr annotation you will not get any EDRs, not even for exceptions.

Q: How can data from the current context be included in an alarm?

For example, can an alarm be generated in a request with more than 12 destination addresses? How can information about how many addresses were included in the request be added to the alarm

It is possible to specify some info in the alarm descriptor with something like

```
<data>
      <attribute key="source" value="thesource"/>
</data>
```

Can something be put in the RequestContext using the putEdr method and then get it into the alarm in some way?

Answer

Yes, add custom information by putting this information into the current RequestContext, as show below.

```
RequestContext ctx = RequestContextManager.getCurrent();

ctx.putEdr("address", "tel:1234");
```

This value is part of any EDRs generated in the current request.

The information will be available in the database in the additional_info column. Make sure you are putting in only relevant information.

Q: Is it possible to specify classname in the filtering section?

Answer

Yes, use the <class> tag inside <method> or <exception> in the filter.

```
 <filter>

      <exception>

        <class>com.y.y.z.MyClass</class>

        <name>com.x.y.z.MyException</name>

      </exception>

</filter>
```

Alarm generation

An alarm is a subset of an EDR. To generate an alarm, generate an EDR, either using one generated in aspects or programmatically, and define the ID and the descriptor of the alarm in the alarm descriptor.

The alarm ID, severity, description and other kind of attributes are defined in the alarm descriptor, see "The EDR descriptor" on page 9-22. For extensions, the alarm ID shall be in the range 500 000 to 999 999.

Note: The alarm filter that provides the *first* match in the alarm descriptor is used for triggering the alarm.

There are two ways to trigger an alarm:

- Use an existing EDR that is generated in the plug-in and add its descriptor to the alarm descriptor.
- Programmatically trigger an EDR and add its descriptor in both the alarm descriptor file and the EDR descriptor. Make sure the ID of the alarm is unique and that the description is the same as in the EDR descriptor.T

Trigger an alarm programmatically

Trigger an EDR as described in "EDR Content" on page 9-13. Then specify in the alarm descriptor the corresponding alarms.

Listing 9-19 Example code to trigger an alarm

```
private static final EdrDataHelper helper =
EdrDataHelper.getHelper(MyClass.class);

...

EdrData data = helper.createData();

data.setValue(EdrConstants.FIELD_SOURCE, EdrConstants.VALUE_SOURCE_METHOD);

data.setValue(EdrConstants.FIELD_METHOD_NAME, "com.bea.wlcp.wlng.myMethod");

data.setValue("myAdditionalInformation", ...);

EdrServiceFactory.getService().logEdr(data);

...
```

The corresponding entry in the alarm descriptor that matches this EDR is shown below.

Listing 9-20 Alarm descriptor

```
  <alarm id="2006"

         severity="major"

         description="Sample alarm">

    <filter>
      <method>
        <name>com.bea.wlcp.wlng.myMethod</name>
        <class>com.bea.wlcp.wlng.myClass</class>
      </method>
    </filter>
  </alarm>
```

Alarm content

Below is a list of the information provided in alarms.

Table 9-8 Alarm information for alarm listeners, also stored in DB

Field	Comment
alarm_id	Unique ID for the alarm. Automatically provided by the EdrService.
source	Service name emitting the alarm. Automatically provided by the EdrService.
timestamp	Timestamp in milliseconds since midnight, January 1, 1970 UTC. Automatically provided by the EdrService.
severity	Severity level. Defined in the alarm. descriptor.
identifier	The alarm identifier. Defined in the alarm descriptor. The column in the database will always contain the identifier defined in the alarm descriptor.

Table 9-8 Alarm information for alarm listeners, also stored in DB

Field	Comment
alarm_info	The alarm information or description. Defined in the alarm descriptor.
additional_info	Automatically provided by the EdrService. Not valid for backwards compatible alarm listeners. Each entry is formatted as: key=value\n Similar to the Java properties file. All the custom key/value pairs found in the EdrData except these are present (EdrConstants if not specified): • FIELD_TIMESTAMP • FIELD_SERVICE_NAME • FIELD_CLASS_NAME • FIELD_METHOD_NAME • FIELD_SOURCE • FIELD_DIRECTION • FIELD_POSITION • FIELD_INTERFACE • FIELD_EXCEPTION_NAME • FIELD_ORIGINATING_ADDRESS • FIELD_DESTINATION_ADDRESS • FIELD_CONTAINER_TRANSACTION_ID • FIELD_CORRELATOR • FIELD_SESSION_ID • FIELD_SERVER_NAME • ExternalInvocatorFactory.SERVICE_CORRELATION_ID • FIELD_BC_EDR_ID • FIELD_BC_EDR_ID_3 • FIELD_BC_ALARM_IDENTIFIER • FIELD_BC_ALARM_INFO

CDR generation

A CDR is a subset of an EDR. To generate a CDR, generate an EDR and define the ID of the EDR in the CDR descriptor.

Triggering a CDR

There are two ways to trigger a CDR:

- Use an existing EDR that is generated in the plug-in and add its description to the CDR descriptor.
- Programmatically trigger an EDR and add its description to the CDR descriptor.

Trigger a CDR programmatically

If none of the existing EDRs is appropriate for a CDR, you can programmatically trigger an EDR that will become a CDR. See the section, “Trigger an EDR programmatically” on page 9-12 for information on how to create and trigger an EDR. Specify in the CDR descriptor the description necessary for this EDR to be considered a CDR.

Listing 9-21 Example, triggering a CDR

```
private static final EdrDataHelper helper =
EdrDataHelper.getHelper(MyClass.class);

...

EdrData data = helper .createData();

data.setValue(EdrConstants.FIELD_SOURCE, EdrConstants.VALUE_SOURCE_METHOD);

data.setValue(EdrConstants.FIELD_METHOD_NAME,
"com.bea.wlcp.wlng.myEndOfRequestMethod");

// Fill the required fields for a CDR

data.setValue(EdrConstants.FIELD_CDR_START_OF_USAGE, ...);

...

EdrServiceFactory.getService().logEdr(data);

...
```

The description, in the CDR descriptor, that matches this EDR is shown in Listing 9-22.

Listing 9-22 Filter to match the EDR

```
<cdr>
    <filter>
      <method>
        <name>com.bea.wlcp.wlng.myEndOfRequestMethod</name>
        <class>com.bea.wlcp.wlng.myClass</class>
      </method>
    </filter>
</cdr>
```

CDR content

In addition to the EDR fields, there are specific fields used only for CDRs. They are listed in Table 9-9.

Table 9-9 Fields in EdrConstants specific for CDRs.

Field in EdrConstants	Comment
FIELD_CDR_SESSION_ID	
FIELD_CDR_START_OF_USAGE	
FIELD_CDR_CONNECT_TIME	
FIELD_CDR_END_OF_USAGE	
FIELD_CDR_DURATION_OF_USAGE	
FIELD_CDR_AMOUNT_OF_USAGE	

Table 9-9 Fields in EdrConstants specific for CDRs.

Field in EdrConstants	Comment
FIELD_CDR_ORIGINATING_PARTY	
FIELD_CDR_DESTINATION_PARTY	Same pattern applies as for send lists, see "Using send lists" on page 9-19.
FIELD_CDR_CHARGING_INFO	

The structure of the CDR content is aligned toward the 3GPP Charging Applications specifications. As a result the database schema has been changed to accommodate these ends and to facilitate future extensions.

Legends:

- NU: Not used
- NC: New column in DB
- RC: Renamed column in DB

Table 9-10 Content in database

Field	Comment	DB
transaction_id	Unique id for the CDR. Provided automatically by the EDR service.	x
service_name	name of the service Provided automatically by the EDR service.	x
service_provider	the service provider account ID Provided automatically by the EDR service.	x
application_id	the application account ID (was user_id in 2.2)	RC
application_instance_grp_id	the application instance ID.	NC
container_transaction_id	id of the current user transaction Provided automatically by the EDR service.	NC

Table 9-10 Content in database

Field	Comment	DB
server_name	name of the server that generated the CDR. Provided automatically by the EDR service.	NC
timestamp	in ms since midnight, January 1, 1970 UTC	NC
service_correlation_id	Service Correlation ID. Provided automatically by the EDR service.	NC
charging_session_id	Id that correlates requests that belong to one charging session as defined by the plug-in. Was 'session_id' in 2.2. Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
start_of_usage	The date and time the service capability module started to use services in the network (in ms since midnight, January 1, 1970 UTC) Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
connect_time	The date and time the destination party responded (in ms since midnight, January 1, 1970 UTC). Used for call control only. Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
end_of_usage	The date and time the service capability module stopped using services in the network (in ms since midnight, January 1, 1970 UTC). Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR	x

Table 9-10 Content in database

Field	Comment	DB
duration_of_usage	The total time the service capability module used the network services (in ms) Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR	x
amount_of_usage	Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
originating_party	The originating party address with scheme included (e.g. “tel:1234”) Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
destination_party	the originating party address with scheme included (e.g. “tel:1234”). Additional addresses are stored in the additional_info field.	x
charging_info	The charging service code from the application. Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.	x
additional_info	Additional information provided by the plug-in	x
revenue_share_percentage	Not used.	NU
party_to_charge	Not used.	NU
slee_instance	Not used.	NU
network_transaction_id	Not used.	NU
network_plugin_id	Not used.	NU
transaction_part_number	Not used.	NU
completion_status	Not used.	NU

Additional_info column

The EDR populates the additional_info column of the DB with all the custom key/value pairs found in the EdrData except the ones listed below.

Excluded keys (EdrConstants if not specified):

- FIELD_SERVICE_NAME
- FIELD_APP_INSTANCE_GROUP_ID
- FIELD_SP_ACCOUNT_ID
- FIELD_CONTAINER_TRANSACTION_ID
- FIELD_SERVER_NAME
- FIELD_TIMESTAMP
- ExternalInvocatorFactory.SERVICE_CORRELATION_ID
- FIELD_CDR_SESSION_ID
- FIELD_CDR_START_OF_USAGE
- FIELD_CDR_CONNECT_TIME
- FIELD_CDR_END_OF_USAGE
- FIELD_CDR_DURATION_OF_USAGE
- FIELD_CDR_AMOUNT_OF_USAGE
- FIELD_CDR_ORIGINATING_PARTY
- FIELD_CDR_DESTINATION_PARTY
- FIELD_CDR_CHARGING_INFO
- FIELD_CLASS_NAME
- FIELD_METHOD_NAME
- FIELD_SOURCE
- FIELD_DIRECTION
- FIELD_POSITION

- FIELD_INTERFACE
- FIELD_EXCEPTION_NAME
- FIELD_ORIGINATING_ADDRESS
- FIELD_DESTINATION_ADDRESS
- FIELD_CORRELATOR
- FIELD_APP_ACCOUNT_ID
- FIELD_SESSION_ID
- FIELD_BC_EDR_ID
- FIELD_BC_EDR_ID_3
- FIELD_BC_ALARM_IDENTIFIER
- FIELD_BC_ALARM_INFO

Two keys not present in the EdrData are added to additional_info.

Table 9-11 Keys not present in EdrData, but added in additional_info

Key	Description
destinationParty	If a send list is specified as the destination party, the first address will be written in the destination_party field of the DB and the remainder of the list will be written under this key name
oldInfo	Any backwards compatible additional info is available

The format of the additional_info field is formatted as:

```
key=value\n
```

similar to the Java properties file.

Out-of-the box (OOTB) CDR support

It is difficult to come up with a CDR generation scheme that fulfills the requirements of all customers. Oracle Communications Services Gatekeeper generates a default set of CDRs which can be customized by re-configuring the CDR descriptor.

The guiding principle for deciding when to generate CDRs is:

- Generate a CDR when you are 100% sure that you have completely handled the service request

In other words, after the last method, in a potential sequence of method calls, returns.

For network-triggered requests this means that you should a trigger a CDR at the south interface after the method has returned back to the network. For application-triggered requests generate a CDR at the north interface after the method has returned to the Network Tier SLSB.

CHAPTER 10

Making Communication Services Manageable

Once you have created your extension Communication Service, any OAM functions that you have designed - read/write attributes and/or operations - must be exposed in a way that allows them to be accessed and manipulated, either through the Oracle Communications Services Gatekeeper Administration Console extension, or through other management tools. The following chapter provides a description of the mechanism that Oracle Communications Services Gatekeeper uses to accomplish this.

Overview

Oracle Communications Services Gatekeeper uses the Java Management Extensions (JMX) 1.2 standard, as it is implemented in JDK 1.6. The JMX model consists of three layers, Instrumentation, Agent, and Distributed Services. As a Communication Service developer, you work in the Instrumentation layer. You create managed beans (MBeans) that expose your Communication Service management functionality as a management interface. These MBeans are then registered with the Agent, the Runtime MBean Server in the WebLogic Server instance, which makes the functionality available to the Distributed Services layer, management tools like the Oracle Communications Services Gatekeeper Administration Console. Finally, because configuration information needs to be persisted, you store the values you set using Oracle Communications Services Gatekeeper's Configuration Store, which provides a write-through database cache. In addition to persisting the configuration information, the cache also provides cluster-wide access to the data, updating a cluster-wide store whenever there is a change in globally relevant configuration data.

For more information on the JMX model in general in relation to WebLogic Server, see *Oracle WebLogic Server Developing Manageable Applications with JMX* at *http://download.oracle.com/docs/cd/E12840_01/wls/docs103/jmxinst/*.

Create Standard JMX MBeans

Creating standard MBeans is a three step process.

1. Create an MBean Interface
2. Implement the MBean
3. Register the MBean with the Runtime MBean Server

Configuration settings should be persisted, see Use the Configuration Store to Persist Values.

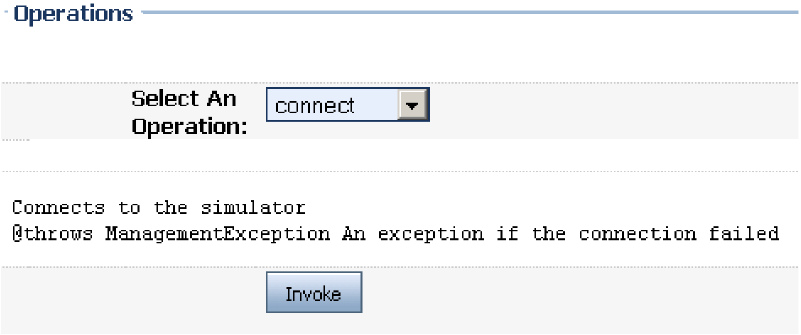
Create an MBean Interface

The first thing you need to do is to create an interface file that describes getter and setter methods for each class attribute that is to be exposed through JMX (getter only for read-only attributes; setter only for write-only) and a wrapper operation for each class method to be exposed. The attribute names should be the case-sensitive names that you wish to see displayed in the UI of the Console extension.

- For each read-write attribute define a *get* and *set* method that follows this naming pattern: get<Attribute name>, set<Attribute name> where <Attribute name> is a case-sensitive name that you want to expose to JMX clients.
- For each read-only attribute define only an *is* or a *get* method. For each write-only attribute, define only a *set* method.
- The JavaDoc will be rendered in the console as a description of an attribute or operation. It will render exactly as in the JavaDoc. For example:

```
/**
   * Connects to the simulator
   * @throws ManagementException An exception if the connection failed
   */
  public void connect() throws ManagementException;
```

  Will render as:



```
Connects to the simulator
@throws ManagementException An exception if the connection failed
```

- Any internal operation or attribute should be annotated with @Internal annotation. This attribute or method will not be shown in the console. Example:

  ```
  @Internal
  public String resetStatistics();
  ```

- Indicate optional parameters for the operation by using the @OptionalParam annotation. In the JavaDoc for the operation, explicitly specify which parameters are optional. Example:

  ```
  /**
      * Gets the alarms matching the specified criteria from the database
      * @param Identifier EDR Identifier
      * @Param Source server name (optional)
      * @Param Severity 0 - Critical, 1- Major, 2 -Minor
      * @Param maxEntries max number of entries
      */
      AlarmData[] getAlarms(long identifier,
                            @OptionalParam('source')String source,
                            int severity,
                            int maxEntries) throws ManagementException;
  ```

The interface should be named `<ServiceName>MBean.java`. The interface for the example Communication Service provided with the Platform Development Studio is named `ExampleMBean.java`.

Implement the MBean

Once you have defined the interface, it must be implemented.

You must name your class `<ServiceName>MBeanImpl.java,` based on the interface name. The implementation must extend `WLNGMBeanDelegate.` This class takes care of setting up notifications and MBean descriptions and all MBean implementation classes must extend it. All MBean implementations must also be public, non-abstract classes and have at least one public constructor. The MBean implementation for the example Communication Service provided with the Platform Development Studio is named `ExampleMBeanImpl.java`.

- The MBean implementation must be a public, non abstract class
- The MBean must have at least one public constructor
- The MBean must implement its corresponding MBean interface and extend WLNGMBeanDelegate

Register the MBean with the Runtime MBean Server

The MBean must be registered with the Runtime MBean Server in the local WebLogic Server instance. Oracle Communications Services Gatekeeper provides a proxy class for MBean registration:

```
com.bea.wlcp.wlng.api.management.MBeanManager
```

The MBean implementation is registered using an ObjectName, and a DisplayName:

```
registerMBean(Object mBeanImpl, ObjectName objectName, String displayName)
```

Construct the ObjectName using:

```
constructObjectName(String type, String instanceName, HashMap properties)
```

There should be no spaces in the InstanceName or Type. Object names are case-sensitive

If the MBean is a regular MBean, use the conventions as illustrated in Table 10-1

Table 10-1 MBean ObjectName

The ObjectName convention for extensions	
type	Fully qualified MBean Name. <MBeanObj>.class.getName()
instanceName	Unique name that identifies the instance of the MBean. For example, it can be obtained from serviceContext.getName() The unique name of the MBean. If this is a plug-in that potentially is used on the same server with multiple plug-in instances this should be unique per plug-in instance. It is recommended to use managedPlugin.getId().
properties	HashMap that contains objectName key and value pairs ObjectNameConstants class has set of constants that can be used as keys. Null for non-hierarchical MBeans.

Example:

com.bea.wlcp.wlng:AppName= wlng_nt_sms_px21#4.1,InstanceName=
Plugin_px21_short_messaging_smpp,
Type=com.bea.wlcp.wlng.plugin.sms.smpp.management.SmsMBean

If the MBean is an MBean that should be the child of a regular MBean, use the conventions as illustrated in Table 10-2

Table 10-2 MBean ObjectName with hierarchy

The ObjectName convention for extensions	
type	Fully qualified MBean Name of the parent MBean. <Parent MBeanObj>.class.getName()
instanceName	Unique name that identifies the instance of the parent MBean.
properties.key=ObjectNameConstants.LEVEL1_INSTANCE_NAME	properties.value is a unique name that identifies the instance of the MBean

Table 10-2 MBean ObjectName with hierarchy

The ObjectName convention for extensions	
properties.key=ObjectNameConstants.LEVEL1_TYPE	Fully qualified MBean Name: <MBeanObj>.class.getName()
properties.key=ObjectNameConstants.LEVEL2_INSTANCE_NAME	properties.value is a unique name that identifies the instance of the MBean
properties.key=ObjectNameConstants.LEVEL2_TYPE	Fully qualified MBean Name: <MBeanObj>.class.getName()

Example:

A child MBean, for example HeartBeatConfiguration, can register with the same Level1InstanceName for all instances of the Plug-in (since it is a child, its MBean name depends on the parent’s instance:

com.bea.wlcp.wlng:AppName= wlng_nt_sms_px21#4.1,InstanceName= Plugin_px21_short_messaging_smpp, Type=com.bea.wlcp.wlng.plugin.sms.smpp.management.SmsMBean,Level1InstanceName=HeartBeatManager,Level1Type=com.bea.wlcp.wlng.heartbeat.management.HeartbeatMBean

com.bea.wlcp.wlng:AppName= wlng_nt_multimedia_messaging_px21#4.1,InstanceName Plugin_px21_multimedia_messaging_mm7, Type= com.bea.wlcp.wlng.plugin.multimediamessaging.mm7.management.MessagingManagementMBean,Level1InstanceName=HeartBeatManager,Level1Type=com.bea.wlcp.wlng.heartbeat.management.HeartbeatMBean

Use the Configuration Store to Persist Values

The Oracle Communications Services Gatekeeper Configuration Store API provides a cluster-aware write-through database cache. Parameters stored in the Configuration Store are both cached in memory and written to the database. The store works in two modes: Local and Global. Values stored in the Local store are of interest only to a single server instance, whereas values stored in the Global store are of interest to all servers cluster-wide. Updates to a value in the Global store update all cluster nodes. The example Communication Service provides a handler class, `ConfigurationStoreHandler`, that gives an example of both usages of the Configuration Store API.

Note: The configuration store supports only Boolean, Integer, Long, and String values.

CHAPTER 11

Service Interceptors

The following sections give a high-level overview of service interceptors and describe both the out-of-the-box interceptors that ship with Oracle Communications Services Gatekeeper and the process of developing your own custom interceptors:

Overview

Interceptors are used to:

- Provide a mechanism to intercept and manipulate a request flowing through any arbitrary Communication Service in Oracle Communications Services Gatekeeper

- Supply an easy way to modify the request flow
- Simplify the routing mechanism for plug-ins
- Centralize policy and SLA enforcement

Some typical use cases for interceptors are to:

- Deny a request if the user does not subscribe to a particular service in the application layer.
- Deny a request if a PIN is not valid
- Verify that a request's parameters are valid
- Perform argument manipulation like aliasing

A set of standard interceptors are provided out-of-the-box. Some are required, while others provide extra functionality. In addition, custom interceptors can be developed.

Interceptor Decisions and Request Flow

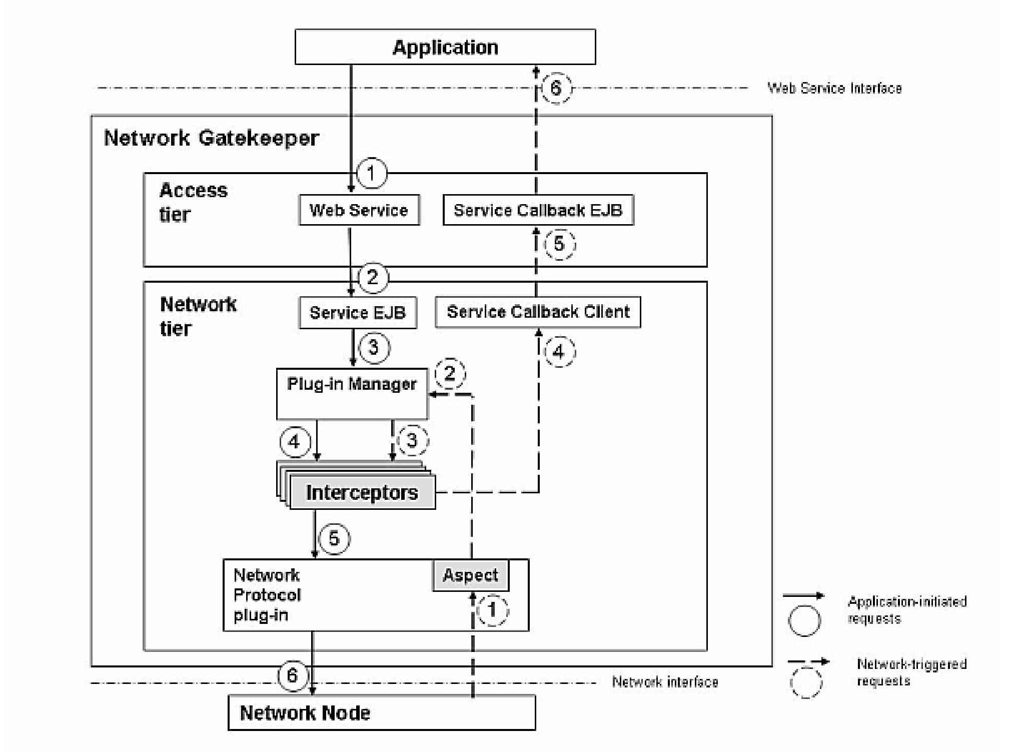
An interceptor makes a decision whether to permit, deny, or stay neutral to a particular request: see Decisions. The Plug-in Manager is responsible for calling the first interceptor in the chain of interceptors as defined in the interceptor configuration file: see Flow Control. When changing the chain of interceptors, the interceptor module normally needs to be redeployed: see Changing the invocation order.

Decisions

For application-initiated requests, the Plug-in Manager is called automatically by the service EJB for the application-facing interface. For network-triggered requests, the Plug-in Manager is called by an aspect that is woven prior to calling the service callback EJB for the application-facing interface.

Figure 11-1 illustrates where interceptors are triggered, both for application-initiated requests and for network-triggered.

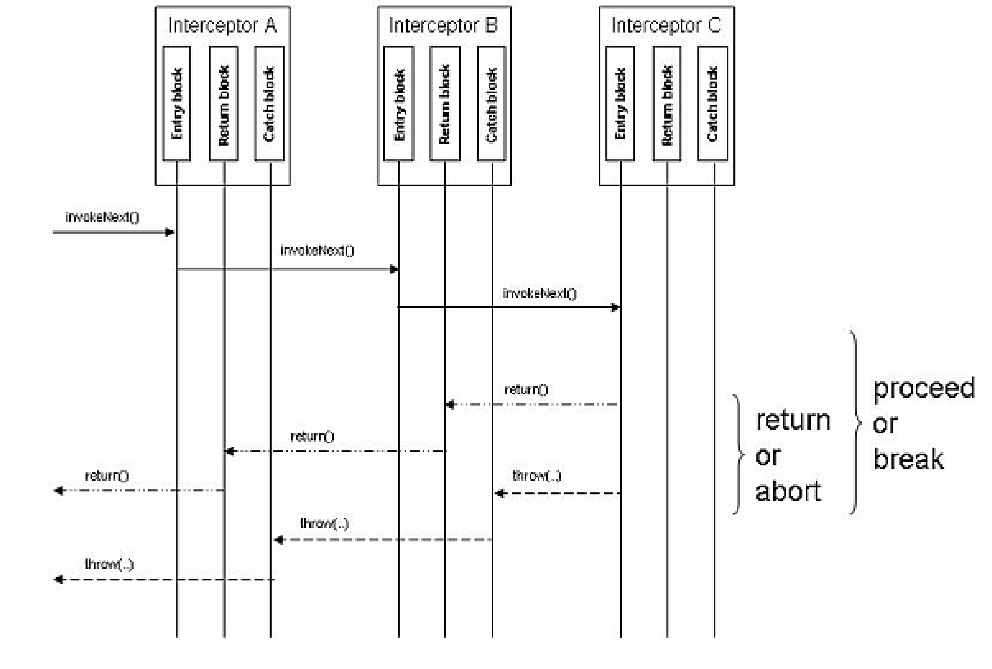
Figure 11-1 Interceptors and the request flow



The interceptor chain is invoked at the point-cut that is a Java representation of the application-facing interface. Note that some application-initiated requests are not necessarily propagated to the network, and some network-triggered requests are not necessarily forwarded to the service callback client.

Each interceptor is responsible for deciding whether to continue to *proceed* down the chain of interceptors or to break it. The interceptor has two ways to break the chain, either to *return* or to *abort*.

Figure 11-2 Proceeding or breaking the interceptor chain



When the decision is to:

- *Proceed*, the request is passed on to the next interceptor in the chain and ultimately to the network protocol plug-in or to the application. When the request is returned from either one of these, the return path traverses the interceptors that were used in the calling path, making in possible to manipulate the request in the return path and ultimately return to the originator of the request, the application or the network node.
- *Return*, the request is rolled back through the previous interceptors using a regular return statement, making it possible for the previous interceptors to manipulate the request in the rollback path and ultimately return to the originator of the request, the application or the network node.
- *Abort* the request, is rolled back through each interceptors' exception catch-block rather than returning in a regular mode.

  - For application-initiated requests the exception is reported back to the application. It is possible to reuse the exception catalogue to map the exception thrown by the interceptor to an exception defined by the application-facing interface. `com.bea.wlcp.wlng.api.plugin.DenyPluginException` should be used by the interceptors for this scenario.
  - For network-triggered requests, it is the responsibility of the plug-in to act on the thrown exception.

The interceptors have access to context data for the request. The actual data that is available depends on the context of the request. In general, the data that is available is the data that is defined by the application-facing interface, and includes the following items:

- The RequestContext for the request, including:
  - Service provider account ID.
  - Application account ID.
  - Application User ID.
  - Transaction ID.
  - Session ID.
  - A Java Map containing arbitrary request-specific data.
- The type of plug-in targeted by the request for (application-initiated requests).
- The type of object targeted by the request (network-triggered requests).
- The method targeted by the request.
- The arguments that will be used in the method targeted by the request.
- The set of RequestInfo available to the request, including:
  - method name.
  - arguments to the method.
  - plug-in type.
- A list of plug-ins that matches the specified RequestInfo.
- The interception point: is the request is network-triggered or is it application-initiated.

The following data can be set by the interceptor:

- In the RequestContext:

  - Session ID.
  - Transaction ID.
  - Java Map.
- A list of plug-ins that matches the specified RequestInfo.
- Arguments to the method.

Flow Control

The invocation order of interceptors is defined in an XML-based configuration file that contains the interceptors: see Standard Interceptors.

Each interceptor is identified by the class name of the entry point of the interceptor, that is, the class that implements the Service Provider Interface (SPI) `Interceptor`.

The configuration file, which is expressed in XML, contains the tags described in Table 11-1.

Table 11-1 Description of interceptor configuration file

Tag	Description
<interceptor-config>	Main tag. Contains zero or more <position> tags.
<position>	Contains one or more <interceptor> tags. Has an attribute name which is either: • MT_NORTH, which indicates that all <interceptor> tags encapsulated by this tag are valid for application-initiated (mobile terminated) requests. • MO_NORTH, which indicates that all <interceptor> tags encapsulated by this tag are valid for network-triggered (mobile originated) requests. An interceptor may be present in both.
<interceptor>	Has the following attributes: • class, which identifies the class for the interceptor implementation, see above. • index, which indicates the invocation order relative to other interceptors within the same <position> tag. The order is ascending. Must be unique.

Listing 11-1 Example of an interceptor configuration file

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

<interceptor-config xmlns="http://www.bea.com/ns/wlng/30"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://www.bea.com/ns/wlng/30 config.xsd">

  <position name="MT_NORTH">

    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceApplicationState"
index="100"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceSpAppBudget"
index="200"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.ValidateRequestUsingRequestFactory"
index="300"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePluginList"
index="400"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveInactivePlugin"
index="500"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.FilterPluginListUsingCustomMatch"
index="700"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.RoundRobinPluginList"
index="800"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceNodeBudget"
index="900"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.InvokeServiceCorrelation" index="1000"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.FindAndValidateSLAContract"
index="1100"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePolicyData"
index="1200"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.CheckMethodParametersFromSLA"
index="1300"/>
```

```
    <interceptor
class="com.bea.wlcp.wlng.interceptor.EnforceBlacklistedMethodFromSLA"
index="1400"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.InjectValuesInRequestContextFromSLA"
index="1500"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.EvaluateILOGPolicy"
index="1600"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.InvokePlugin"
index="1700"/>

  </position>

  <position name="MO_NORTH">

    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceApplicationState"
index="100"/>

    <interceptor
class="com.bea.wlcp.wlng.interceptor.InvokeServiceCorrelation" index="200"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePolicyData"
index="300"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.EvaluateILOGPolicy"
index="400"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.InvokeApplication"
index="500"/>

  </position>

</interceptor-config>
```

Each interceptor is responsible for calling the next interceptor in the chain, as opposed to being invoked by a delegator. This means that:

- For an application-initiated request, the interceptors can change and add request-specific data. This data is then propagated to the next interceptor and ultimately to the network protocol plug-in. When the request returns from the plug-in, the data can be changed as the request is returning through the invocation chain.
- For network-triggered request, the interceptors can change and add request-specific data. This data is then propagated to the next interceptor and ultimately to the application. When

the request returns from the application, the data can be changed as the request is returning through the invocation chain.

This is useful for aliasing of data, where the interceptor anonymizes request data such as telephone numbers so that an application is not aware of the true subscriber telephone number.

For application-initiated requests, the last interceptor in the chain is responsible for calling the plug-in. The out-of-the-box interceptor InvokePlugin does this.

For network-triggered requests, the last interceptor in the chain is responsible for calling the callback service EJB, which calls the application. The out-of-the-box interceptor InvokeApplication does this.

In either scenario, the first interceptor is called by the Plug-in Manager. The Plug-in Manager is, for application-triggered requests, invoked by the service EJB. For network triggered requests, the Plug-in Manager is invoked by an aspect applied to the north interface of the plug-in.

Changing the invocation order

As described in Flow Control, the invocation order of the interceptors is defined in the interceptor configuration file, see Table 11-3.

To rearrange the invocation chain, explode the ear file, edit the config.xml file and change the attribute `index` in the tag `<interceptor>`. Repackage the ear file and deploy it.

To exclude an interceptor chain, explode the ear file and delete or comment out the `<interceptor>` tag for it. Repackage the ear file and deploy it.

Always use the `interceptors.ear` deployed on the Administration server as the master and use standard WebLogic procedures to redeploy the application `interceptor.ear` to all servers in the network tier cluster from the Administration server.

Standard Interceptors

Below is a description of the interceptors that are available out of the box as a part of Oracle Communications Services Gatekeeper. The name of the interceptor in the configuration file is the fully qualified class name. That is, it is prefixed with `com.bea.wlcp.wlng.interceptor`.

Table 11-2 Out-of-the-box interceptors

Interceptor	Description
EnforceApplicationState	
	Enforces the application state. Verifies that the application with which the request is related has established a session with Oracle Communications Services Gatekeeper.
EnforceSpAppBudget	
	Enforces the budget defined in the service provider group SLA and application group SLA. Is related to the SLA tag <rate> in <methodRestrictions>: see Defining Service Provider Level and Application Level Service Agreements.
ValidateRequestUsingRequestFactory	
	Validates the request using the RequestFactory corresponding to the type of plug-in the request is intended for. See description of the class RequestFactory.
CreatePluginList	
	Creates a list of plug-ins that are capable of handling the given request. Populates the RequestInfo object.
RemoveInactivePlugin	
	Removes any plug-in that is not active from the current plug-in list. CreatePluginList must have been invoked prior to this.
FilterPluginListUsingCustomMatch	
	Invokes the custom match method of each plug-in in the current plug-in list. The custom match method either removes the plug-in from the current plug-in list or marks it as required. CreatePluginList must have been invoked prior to this.
RemoveOptional	

Table 11-2 Out-of-the-box interceptors

Interceptor	Description
	Removes any plug-in that is marked as optional if there is a at least one marked as required in the current plug-in list. CreatePluginList must have been invoked prior to this.
RoundRobinPluginList	
	Performs a round-robin of the list of available plug-ins. This is not a strict round-robin, but a function of the number of plug-ins that match the request and the number of destination or target addresses in the request. If these parameters are consistent, a true round-robin is performed. CreatePluginList must have been invoked prior to this.
EnforceNodeBugdet	
	Enforces all settings in the service provider node SLA and global node SLA, including validity of the dates and the budgets. See Writing Node SLAs. EnforceSpAppBudget must have been invoked prior to this.
InvokeServiceCorrelation	
	Invokes the service correlation feature, see Service Correlation.
FindAndValidateSLAContract	
	Enforces the existence of application level and service provider level SLAs for the given request. It also verifies that the dates given in the SLA are current. See Defining Service Provider Level and Application Level Service Agreements.
CreatePolicyData	
	Creates the policy request data object needed by other interceptors.
RemoveInvalidRoute	
	Enforces the plug-in routing logic. CreatePluginList and CreatePolicyData must have been invoked prior to this.
CheckMethodParametersFromSLA	

Table 11-2 Out-of-the-box interceptors

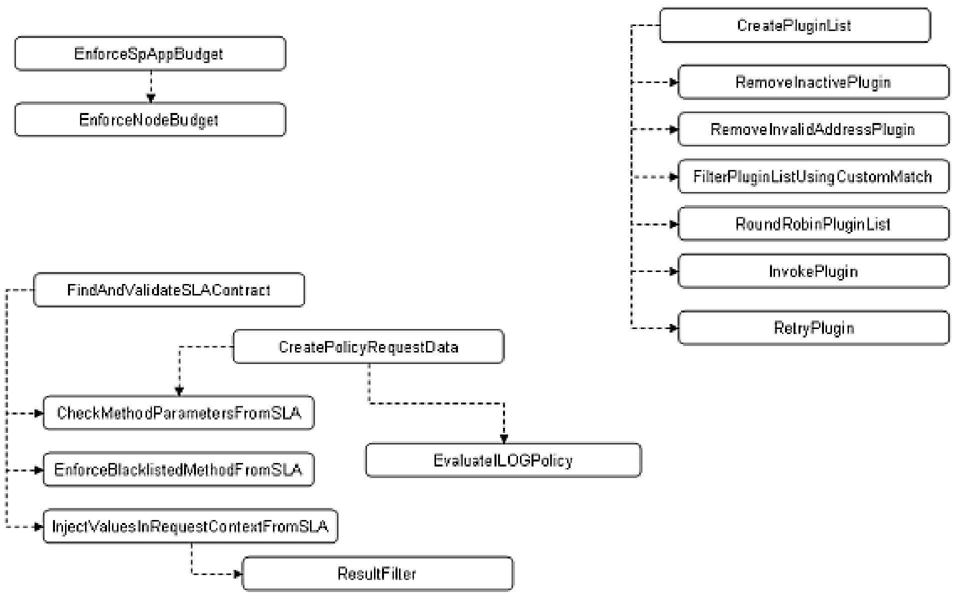
Interceptor	Description
	Checks and enforces that the request parameters are allowed as specified in the service provider group and application group SLAs. Is related to the SLA tags <parameterName> and <parameterValue> in <methodParameters>, see Defining Service Provider Level and Application Level Service Agreements. FindAndValidateSLAContract and CreatePolicyRequest must have been invoked prior to this.
EnforceBlacklistedMethodFromSLA	
	Enforces the method blacklist as specified in the service provider group and application group SLAs. Is related to the SLA tag <blacklistedMethod> in <methodAccess>. See Defining Service Provider Level and Application Level Service Agreements. FindAndValidateSLAContract must have been invoked prior to this.
InjectValuesInRequestContextFromSLA	
	Adds any optional request context attribute as specified in the service provider group and application group SLAs. Is related to the SLA tags <attributeName>, <attributeValue>, and <contextAttribute> in <requestContext>. See Defining Service Provider Level and Application Level Service Agreements. FindAndValidateSLAContract must have been invoked prior to this.
EvaluateILOGPolicy	
	Evaluates any custom ILOG policy rules. CreatePolicyData must have been invoked prior to this.
InjectXParamtersFromRequestContext	
	Takes tunnelled parameters from the RequestContext and puts them in the the SOAP header of either a request to an application or in response to a request from an application.
InvokePlugin	

Table 11-2 Out-of-the-box interceptors

Interceptor	Description
	Invokes the plug-in(s). This should be the last interceptor for an application-initiated (mobile terminated) request. CreatePluginList must have been invoked prior to this.
InvokeApplication	
	Invokes the Application via the service callback EJB. This should be the last interceptor for an network-triggered (mobile originated) request.
RetryPlugin	
	Performs retries of request. See Retry functionality for plug-ins. CreatePluginList must have been invoked prior to this.
EnforceNodeBudget	
	Enforces budgets related to the global and service provider node SLAs. CreatePluginList must have been invoked prior to this.
EnforceSubscriberBudget	
	Enforces budgets related to the Subscriber SLAs. CreatePluginList must have been invoked prior to this.
ResultFilter	
	Applies result filters as specified in the service provider group and application group SLAs. Relates to the SLA tag <resultRestriction>. See Defining Service Provider Level and Application Level Service Agreements. InjectValuesInRequestContextFromSLA must have been invoked prior to this.

Note: Some interceptors must be invoked before others can be invoked. A quick overview of the necessary sequences is seen in Figure 11-3

Figure 11-3 Required Interceptor Sequences


EnforceSpAppBudget
EnforceNodeBudget
FindAndValidateSLAContract
CreatePolicyRequestData
CheckMethodParametersFromSLA
EnforceBlacklistedMethodFromSLA
InjectValuesInRequestContextFromSLA
ResultFilter
EvaluateILOGPolicy
CreatePluginList
RemoveInactivePlugin
RemoveInvalidAddressPlugin
FilterPluginListUsingCustomMatch
RoundRobinPluginList
InvokePlugin
RetryPlugin

All out-of-the-box interceptors are classes packaged in `$OCSG_HOME/applications/interceptors.ear`.

Table 11-3 provides a description of the contents of this ear:

Table 11-3 Contents of interceptor.ear

Path	Content
`/`	
	`dummy.war` Empty war file. Present in order to deploy the interceptors. Do not remove or change.
`/APP-INF/classes/`	
	`config.xml` Interceptor configuration file. See Flow Control.

Table 11-3 Contents of interceptor.ear

Path	Content
	`config.xsd` Schema for config.xml
`/APP-INF/classes/com/bea/wlcp/wlng/interceptor/`	
	Classes for the out-of-the-box interceptors, see Table 11-2. Do not change the content of this directory.
`/APP-INF/classes/com/bea/wlcp/wlng/interceptor/deploy/`	
	Infrastructure for the interceptor functionality. Do not change the content of this directory.
`META-INF/`	
	`MANIFEST.MF` Manifest file for the interceptor infrastructure.
	`application.xml` Deployment descriptor. Do not edit or remove.
	`weblogic-application.xml` WebLogic extensions to `application.xml`. Do not edit or remove.
`WEB-INF/`	
	No content.

Retry functionality for plug-ins

The RetryPlugin interceptor handles retry functionality for plug-ins. The retry is attempted among the plug-ins that were chosen based on the data provided in the request. Retries are only performed among the plug-ins in the same Oracle Communications Services Gatekeeper instance.

The RetryPlugin is triggered when a plug-in throws a RetryPluginExeption. This exception is captured by the RetryPlugin interceptor, which removes the plug-in that threw the exception from the list of chosen plug-ins and calls the next interceptor in the chain.

The different decision scenarios are described below.

If the RequestInfo objects in the RequestContext are associated with:	The RetryPlugin interceptor:
PluginHolder objects that are marked as optional	removes the failed RequestInfo from the RequestContext and the next interceptor in the chain is invoked.
PluginHolder objects that are marked as required	treats the request itself as failed. No retry is performed, and an exception is thrown.
some PluginHolder objects that are marked as optional, and some that are marked as required	removes the RequestInfo objects that are associated with the PluginHolder objects that are marked as optional from the RequestContext and the next interceptor in the chain is invoked.

The following out-of-the-box plug-in throws the RetryPluginException:

- Subscriber Profile/LDAPv3.

Custom plug-ins can use the infrastructure for retries as provided by the RetryPlugin interceptor. This exception should be thrown if the communication with the underlying network node fails, or if an unexpected error is reported back from the plug-in.

Custom Interceptors

Developing Custom Interceptors

An interceptor implements the interface `com.bea.wlcp.wlng.api.interceptor.Interceptor`.

This interface defines the method:

```
Object invoke(com.bea.wlcp.wlng.api.interceptor.Context context) throws
Exception;
```

The interceptor is responsible for invoking the next interceptor in the invocation chain using the method:

`Object com.bea.wlcp.wlng.api.interceptor.Context.invokeNext(Interceptor current) throws Exception;`

Since the interceptors call each other, the normal case would be just to return the object that was returned by the called interceptor. But in some cases, the returned object may be changed in order, for example, to do aliasing.

The decisions within the interceptor are expressed in these ways:

- To *proceed*, continue down the invocation chain by calling the next interceptor.
- To break the chain due to a violation: for example a parameter in the request is out-of bounds, or that usage policies are violated. This *aborts* the request throwing a `PluginDenyException`.
- To break the chain because the request has been fulfilled (for example because there is no need to call the plug-in or the application in order to fulfill the needs of the request), simply *return* the request.

See Interceptor Decisions and Request Flow.

Note: The interceptor must be thread safe.

Listing 11-2 illustrates a very basic interceptor.

Listing 11-2 Example interceptor

```
import com.bea.wlcp.wlng.api.interceptor.Interceptor;


public class SampleInterceptor implements Interceptor {
  private final int ABORT = 0;
  private final int RETURN = 1;


  public Object invoke(Context ctx) throws Exception {
    int decision = // Logic that evaluates the request and makes a decision.
    if (decision == ABORT) {
      throw new Exception();
    } else if (decision == RETURN) {
      Object returnValue = // Define a returnValue here if desired.
      return returnValue;
    } else {
      Object returnValue = ctx.invokeNext(this);
      // Define a new returnValue here if desired, for example for aliasing.
        return returnValue;
    }
  }
}
```

All necessary classes are available in the package:

`com.bea.wlcp.wlng.api.interceptor` located in `$PDS_HOME/lib/api/wlng.jar`

As an alternative to embedding the interceptor in `interceptors.ear` and defining the invocation order in `/APP-INF/classes/config.xml` it is possible to put the new interceptor in a separate ear file. Using this alternative, the interceptor must register the interceptor using the `InterceptorManager`, which is retrieved using the `InterceptorManagerFactory`.

When registering the interceptor manually, data corresponding to the data set in `/APP-INF/classes/config.xml` in `interceptors.ear` is supplied as parameters to the method:

```
void register(Interceptor interceptor, InterceptionPoint ip, int index);
```

in the `InterceptorManager` interface.

The attribute `name` in the tag `<position>` corresponds to the argument `ip`, the attribute `index` in the tag `<interceptor>` corresponds to the argument `index`.

Listing 11-3 shows an example of how to register an interceptor manually.

Listing 11-3 Manually registering an interceptor

```
InterceptorManager im = InterceptorManagerFactory.getInstance(); // Get manager
im.register(myInterceptor,
            InterceptionPoint.MT_NORTH.MT_NORTH,
            myIndex); // Register
im.update(); // Changes do not take effect until update() is called
```

Deploying Custom Interceptors

To deploy the interceptor in the common interceptor ear file, explode the `interceptors.ear` file and put the class files for the interceptor in /APP-INF/classes. Add a new <interceptor> tag with the attribute `class` referring to the entry point of the interceptor and a numeric value in the attribute `index` that corresponds to the location in the interceptor invocation chain.

For example:

If the interceptor main class is `com.acompany.interceptor.DoStuff`, the class `DoStuff` should be inserted into `interceptors.ear` in `/APP-INF/classes/com/acompany/interceptor`, and the corresponding entry in `/APP-INF/classes/config.xml` shall be

```
<interceptor class="com.acompany.interceptor.DoStuff" index="1150"/>
```

See Flow Control for more information about `/APP-INF/classes/config.xml`. See Standard Interceptors to get information about where in the invocation chain to insert the new interceptor.

If deploying the interceptor in a separate ear, always deploy it using the Administration server and use standard WebLogic procedures to deploy the application to all servers in the cluster from the Administration server.

CHAPTER 12

Custom Service Level Agreements

This section describes how to implement enforcement of custom service level agreements (SLAs) and the relationship between the custom SLAs, their XSDs, and the enforcement logic:

Introduction

Custom service level agreements (SLAs), offer a mechanism to add custom SLA enforcement in addition to the SLA enforcement provided out-of-the-box with Oracle Communications Services Gatekeeper. In contrast to the system SLA types that have static XSDs and enforcement logic, the custom SLAs offer configuration time loading of SLA XSDs and runtime deployment of the enforcement logic. It is a framework for definition and enforcement of custom SLAs.

The entities involved include:

- Custom SLA XSDs
- Custom SLAs
- Enforcement logic for the custom SLAs

The custom SLA XSDs are loaded and assigned an SLA type using the management interfaces. Then SLAs are loaded, and associated with a service provider group, application group, or globally. After this is done, the SLA type is used and the custom SLAs are validated against the XSDs.

At run-time, when the custom SLAs are enforced, the enforcement logic is responsible for fetching the enforcement logic relevant for the custom SLA type.

Custom SLAs and XSDs

The SLAs must be expressed in XML and be formatted according to their SLA XSDs. There are no other requirements on the SLAs.

At load time, the custom SLA XSD is validated and associated with an SLA type. This type is used when loading the custom SLA, and the SLA is validated against the XSD.

The XSD and SLA are loaded using the management interfaces. See section Managing SLAs in *Oracle Communications Services Gatekeeper Managing Accounts and SLAs*.

Custom SLA Enforcement

The custom SLA enforcement is implemented as one or more service interceptors. Thus gives the operator the ability to deploy and undeploy the enforcement logic in runtime. It also gives the enforcement logic access to all data about a request from the context object through the class com.bea.wlcp.wlng.api.interceptor.Context.

The service interceptor is responsible for:

- Resolving the request data it needs from the Context object.
- Loading the representation of the custom SLA
- Fetching any other data needed for the enforcement logic
- Manipulating the Context with new data, if necessary
- Allowing or denying the request, if necessary

For information on how to access the data from the Context object, see Service Interceptors.

The Java representation of the custom SLA is fetched from `com.bea.wlcp.wlng.api.sla.CustomSlaManager`.

This class exposes the following methods:

```
Document getApplicationGroupCustomSla(String slaType)

Document getServiceProviderGroupCustomSla(String slaType)

Document getGlobalCustomSla(String slaType)

Object getApplicationGroupCustomSla(String slaType, String parserId)

Object getServiceProviderGroupCustomSla(String slaType, String parserId)

Object getGlobalCustomSla(String slaType, String parserId)

void registerSlaParserCallback(String slaType, String parserId,
SlaParserCallback parser)

void unregisterSlaParserCallback(String slaType, String parserId)
```

There are two ways to get the Java representation of the SLA, through a DOM object or from a custom XML parser:

- Get an SLA using a DOM Object
- Get an SLA using a Custom Parser

Note: A custom SLA parser can produce a more efficient Java representation of the SLA than the more general DOM representation.

The `CustomSlaManager` automatically resolves which custom SLA should be fetched, so there is no need to resolve which group the originator of the request belongs to. In the case of a global SLA, only the custom SLA type is of significance since this scope does not take into account the originator of the request, but is relevant for all requests.

If the combination of SLA data and enforcement logic is intended to add or replace data about the request, the service interceptor must manipulate the `Context` object accordingly.

If the combination of SLA data and enforcement logic is intended to function to deny or allow the request , the service interceptor must throw an exception and break the chain of interceptors or pass on the request to the next interceptor as described in Service Interceptors.

Get an SLA using a DOM Object

When using get methods that return the SLA as an `org.w3c.dom.Document`, a standard DOM parser is used to construct the Java representation of the SLA:

```
Document getApplicationGroupCustomSla(String slaType)
```

```
Document getServiceProviderGroupCustomSla(String slaType)
```

```
Document getGlobalCustomSla(String slaType)
```

The `slaType` identifies the XSDs and returns the custom SLA for the service provider group, application group, or global, respectively. Depending on the scope of the enforcement logic, the corresponding method is used. In this case there is no need to implement and register any parser.

Get an SLA using a Custom Parser

When using get methods to return the SLA as an `Object,` the custom parser parses the SLA and returns an object in a known format:

```
Object getApplicationGroupCustomSla(String slaType, String parserId)
```

```
Object getServiceProviderGroupCustomSla(String slaType, String parserId)
```

```
Object getGlobalCustomSla(String slaType, String parserId)
```

All of the above methods require the ID of parser to use for creating the `Object`. The parser must be registered using:

```
void registerSlaParserCallback(String slaType, String parserId,
SlaParserCallback parser)
```

It can be unregistered using:

```
unregisterSlaParserCallback(String slaType, String parserId
```

The custom SLA parser must implement the interface `com.bea.wlcp.wlng.api.sla.SlaParserCallback`, which defines the method:

```
Object parse(String sla)
```

The parameter sla contains a text-representation of the SLA, and originates from the SLA as loaded using the Account Service. Oracle Communications Services Gatekeeper is responsible for caching and keeping the SLA in sync with the loaded SLA. The implementation of `parse(String sla)` returns the object that is returned by the get methods.

The two methods are equivalent in every aspect except the custom SLA implementation and the parser ID.

Example

Below is an example of how a custom SLA that combines data from an application’s request, the contents of a custom SLA and data from an external source can be implemented. A DOM parser for the SLA is used.

The use case assumes that service provider groups are used to differentiate between different content providers. For example, service provider groups are created for content providers of entertainment, sports, and weather. End-users of the services can opt in to get content of a certain category, and this data is accessible by Service Gatekeeper.

A simple custom SLA schema with entries for allowed content types is created. See Custom SLA Schema and Example SLA. The custom SLA XSD is loaded in Oracle Communications Services Gatekeeper using the management interfaces. Custom SLAs are created that list the content types from these service provider groups. Service provider groups are created for different content types. Each SLA is associated with the corresponding service provider group using the management interfaces.

The enforcement logic for the SLA is created. The logic is deployed as a service interceptor.

When an application uses Service Gatekeeper to deliver content, the request travels through the communication service until the custom service interceptor is reached. The interceptor gets the custom SLA XSD, and - depending on the originator of the request - fetches the appropriate SLA and matches the addressee’s preferences. Based on that information, it allows or blocks the request. For detailed information, see Enforcement Logic.

Custom SLA Schema and Example SLA

Listing 12-1 is an example of a SLA schema that allows a set of content types to be defined.

Listing 12-1 Example SLA Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.example.com"
           xmlns="http://www.example.com"
           elementFormDefault="qualified">
  <xs:element name="contentFilter">
```

```
    <xs:complexType>
     <xs:sequence>
      <xs:element name="allowContents">
       <xs:complexType>
        <xs:sequence>
          <xs:element name="allowContentType"
                      type="xs:string"
                      maxOccurs="unbounded"
                      minOccurs="1"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 12-2 is an SLA that adheres to the schema in Listing 12-1. It allows the content type Entertainment.

Listing 12-2 ContentFilterSla.xml

```
<?xml version="1.0"?>
 <contentFilter xmlns="http://www.example.com"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.example.com contentFilter.xsd">
  <allowContents>
   <allowContentType>Entertainment</allowContentType>
  </allowContents>
 </contentFilter>
```

Enforcement Logic

The enforcement logic of the SLA is implemented as a service interceptor, so it must register itself and de-register itself using the `InterceptorManagerFactory`, see Service Interceptors.

Below are the main steps involved in implementing the enforcement logic:

1. A request enters the interceptor. The destination address of the request is retrieved from `com.bea.wlcp.wlng.api.interceptor.Context` by iterating over the `RequestInfo` objects until the `AddressRequestInfo` is found.

```
for (RequestInfo requestInfo : context.getRequestInfos()) {
  if (requestInfo instanceof AddressRequestInfo) {
   URI uri = ((AddressRequestInfo) requestInfo).getAddress()
   ...
```

2. A lookup of which content types are allowed by the subscriber identified by the destination address is done. This lookup could be done on a subscriber database.
3. The custom SLA for the service provider group is fetched from the `CustomSlaManager`. The SLA is fetched by name and the SLA type given when the XSD for the custom SLA was loaded using the management interfaces. The SLA for the service provider group that is associated with the originating application is resolved automatically by the `CustomSlaManager`. Different methods are used to fetch the custom SLA on service provider group, application group, and global level.

```
Document spSla = slaManager.getServiceProviderGroupCustomSla(CONTENT_FILTER);
```

4. The custom SLA is returned as a org.w3c.dom.Document, and the Document is parsed to get the data, in this case the content of the `<allowContentType>` elements.
5. The content of the SLA is compared to the list of allowed contents for the destination address. If there is a mismatch, an exception is thrown to stop the service interceptor chain. If the request is allowed, it is passed on to the next service interceptor.

Subscriber-centric Policy

Making subscriber personalization easy and offering superior subscriber data protection is key to growing and maintaining a loyal subscriber base. The Platform Development Studio offers a straightforward way to extend the power of Oracle Communications Services Gatekeeper's flexible policy-based control to the operator's subscriber base. The mechanism can be divided into three parts:

- Service Classes and the Subscriber SLA
- The Profile Provider SPI and Subscriber Contracts
- Subscriber Policy Enforcement

Note: There is an example Profile Provider in `$PDS_HOME/example`

Service Classes and the Subscriber SLA

The first step in adding subscriber-centric policy to Oracle Communications Services Gatekeeper is to create a Subscriber SLA. This is an XML file based on the `sub_sla_file.xsd` schema.

Note: The schema file can be found in the `wlng.jar` file located in the `$PDS_HOME/lib/wlng` directory.

The SLA is used to define classes of service in the context of existing Service Provider and Application Groups. (For more information on Service Provider and Application Groups, see "Managing Application Service Providers" in *Concepts and Architectural Overview*, a separate document in this set.) These *service classes* can then be associated with subscribers, based on their preferences and permissions, defining individualized relationships between subscribers and Service Provider and Application Group functionality.

The <reference> tag

The `<reference>` tag specifies the operator's already-established Application and Service Provider Groups that are to be associated with this service class. There are two reference types that define the groups: the `ApplicationGroupReference and` the `ServiceProviderGroupReference.` In addition there are two additional reference types, the `ServiceReference` and the `MethodReference` that indicate specific service interfaces and methods, respectively, covered by those groups. In the Listing 13-1 snippet, the service class `news_subscription` is defined. Evaluation of matches in the class occurs using the following rules:

- If no reference type is specified, everything of that type is a match
- Two or more entries of the same reference type creates an OR relationship
- The default relationship is AND

So, in the case of Listing 13-1, the class covers any request that matches:

- Any of the service interfaces of the `silver_app_group`

  (No `ServiceReference` type is specified, so everything is a match)

- **OR** the `gold_app_group`

  (Two `ApplicationGroupReference` entries creates an **OR)**

  - **AND** the `SendSMS` service interface of the `gold_app_group`

    (The default relationship)

  - **AND** the `content_sp_group`

    (The default relationship)

  - **AND** the `SendSMS` service interface of the `content_sp_group`

    (The default relationship)

  - **AND** either the `sendSms` **OR** the `getSmsDeliveryStatus` methods

    (Two `MethodReference` entries creates an OR)

Listing 13-1 The <reference> element

```
<ServiceClass name="news_subscription">
        <references>
            <ApplicationGroupReference id="silver_app_group"/>
            <ApplicationGroupReference id="gold_app_group">
                <ServiceReference
serviceInterface="com.bea.wlcp.wlng.px21.plugin.SendSmsPlugin"/>
            </ApplicationGroupReference>
            <ServiceProviderGroupReference id="content_sp_group">
                <ServiceReference
serviceInterface="com.bea.wlcp.wlng.px21.plugin.SendSmsPlugin">
                    <MethodReference methodName="sendSms" />
                    <MethodReference methodName="getSmsDeliveryStatus" />
                </ServiceReference>
            </ServiceProviderGroupReference>
        </references>
```

Use of the empty tag, `<references/>`, matches everything.

The <restriction> tag

In addition to the `<reference>` tag, service classes may have a `<restriction>` tag. This tag is used to attach default rates and quotas that are used to create budgets for the classes. These rates and quotas can be replaced in specific contracts.

Note: The XSD requires you either to specify a rate/quota restriction or to use the `<restrictAllType/>` tag.

Listing 13-2 The <restriction> tag

```
<restriction>
```

```
            <rate>
              <reqLimit>5</reqLimit>
              <timePeriod>1000</timePeriod>
            </rate>
            <quota>
              <qtaLimit>600</qtaLimit>
              <days>3</days>
              <limitExceedOK>true</limitExceedOK>
            </quota>
</restriction>
```

These tags function exactly as they do in the other SLAs in Oracle Communications Services Gatekeeper. For more information on these tags, see the **Contract structure** section of the "Defining Service Provider Group and Application Group SLAs" chapter of *Managing Accounts and SLAs*, a separate document in this set. If the `<limitExceedOK>` tag is set to true, the request is allowed even when quota has been exceeded, but an alarm (Alarm id `200000`) is fired

There is also a `<restrictAllType/>` tag. This tag, as its name implies, denies access to all requests.

Managing the Subscriber SLA

There are three management methods in the Service Level Agreement MBean for managing a Subscriber SLA. They are covered in detail in the "Managing SLAs" chapter of *Managing Accounts and SLAs*, a separate document in this set. The methods allow you to load a Subscriber SLA as a string, to load a Subscriber SLA from a URL, and to retrieve a loaded Subscriber SLA.

The Profile Provider SPI and Subscriber Contracts

Once the Subscriber SLA is established, the various service classes it defines must be associated with individual subscribers. The combination of a subscriber (identified by URI) and a service class is called a *subscriber contract*. A subscriber (a URI) can have multiple subscriber contracts associated with it.

The subscriber contract object contains a URI designating the subscriber and the service class type with which it is associated. It also contains an expiration time, represented as a `java.util.Date`.

Note: The subscriber contract constructor will throw an exception if the URI, service class type, and expiration time are not specified.

The subscriber contract may also replace the default rate and/or quota settings in the service class, or set this subscriber to RestrictAll, that is, to deny access for all requests.

The operator or integrator is responsible for creating the mechanism, a Profile Provider, that supplies these subscriber contracts.

Note: All class files related to creating Profile Providers are in the `com.bea.wlcp.wlng.spi.subscriberdata` package, and can be found in the `wlng.jar` file in the `$PDS_HOME/lib/wlng` directory. The JavaDoc for the files can be found in the `$PDS_HOME/doc/javadoc` directory. An example implementation can be found in the `$PDS_HOME/example/profile_providers/src` directory. This sample implementation assumes the use of a properties file to assign subscriber URIs to particular service classes. An example properties file, `exampleSubscriberContractMappingFile.properties`, can be found in the `$PDS_HOME/example/profile_providers/resource` directory.

The Profile Provider must implement the Profile Provider SPI. The SPI defines three methods;

- `init`: Oracle Communications Services Gatekeeper initializes the Profile Provider by passing in a list of the service classes that are defined in the Subscriber SLA and a list of any previously defined subscriber contracts. The Provider returns a list of updated subscriber contracts.
- `contractExpired`: Oracle Communications Services Gatekeeper sends the Provider a list of service classes and a list of expired contracts. The Provider returns an updated list of contracts for those that have expired. The Provider can remove or add contracts to the returned list.
- `serviceClassesUpdated`: Whenever the Subscriber SLA is updated, and the service classes are thus modified, Oracle Communications Services Gatekeeper sends the Provider a list of the updated service classes and a list of all current contracts. The Provider returns an updated list of contracts. The Provider can make any necessary updates to the subscriber contracts.

The Profile Provider implementation must have a public constructor with no parameters or a static method which returns `ProfileProvider`.

Deploying the Custom Profile Provider

Once the `ProfileProviderImpl` has been created, the `.jar` file containing it must be added to the `app-inf/lib` directory of the `profile_providers.ear` file, which can be found in the `$BEA_HOME/ocsg_4.1/applications` directory. You must also modify the `app-inf/classes/ProfileProviders.prop` file, adding a line containing the package and implementation file name of each of your providers (multiple providers are possible). For example:

```
com.mycompany.mypackage.MyProfileProviderImpl
```

Once the EAR is modified, it can be deployed in the normal manner. For more information on deploying EAR files in Oracle Communications Services Gatekeeper, see the "Deployment model for Communication Services and Container Services" chapter in the *System Administrator's Guide*, a separate document in this set.

Subscriber Policy Enforcement

Once the `providers.ear` is deployed, the singleton `SubscriberProfileService` initializes the Profile Provider(s) and receives the relevant subscriber contracts. It uses the Budget Service to create budgets for the contracts, based on the specified rates and quotas, and also creates and schedules a timer based on the expiration times in the contracts. Both the Subscriber SLA and the subscriber contracts are persisted using the Storage Service.

Note: For more information on budgets in Oracle Communications Services Gatekeeper, see the "Managing and Configuring Budgets" chapter in the *System Administrator's Guide*, a separate document in this set.

When a request from an application arrives at Oracle Communications Services Gatekeeper, it passes through the Interceptor Stack for policy evaluation. The `EnforceSubscriberBudget` interceptor manages policy enforcement for subscriber contracts. The process within the interceptor has two phases:

- Do Relevant Subscriber Contracts Exist
- Is There Adequate Budget for the Contracts

Do Relevant Subscriber Contracts Exist

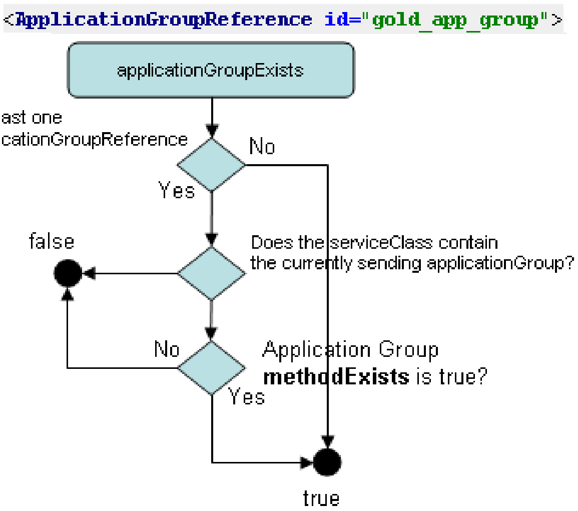
The first thing the interceptor must determine is whether one or more contracts exist that are relevant to the particular request that is being evaluated. The interceptor iterates through all the target URIs in the application request, and evaluates whether or not there are contracts in effect that it should enforce.

- If there are no contracts at all associated with a particular URI, the request is simply passed on to the next interceptor in the sequence.
- If there are contracts associated with a particular URI, a set of evaluations must be carried out. The figures below show the decision flow for the evaluations. All three sections must evaluate to `true` for there to be an enforceable contract.

  Note: The XML snippets correspond to the relevant sections of Listing 13-1:

– Is there an `ApplicationGroupReference` and is it relevant? See Figure 13-1

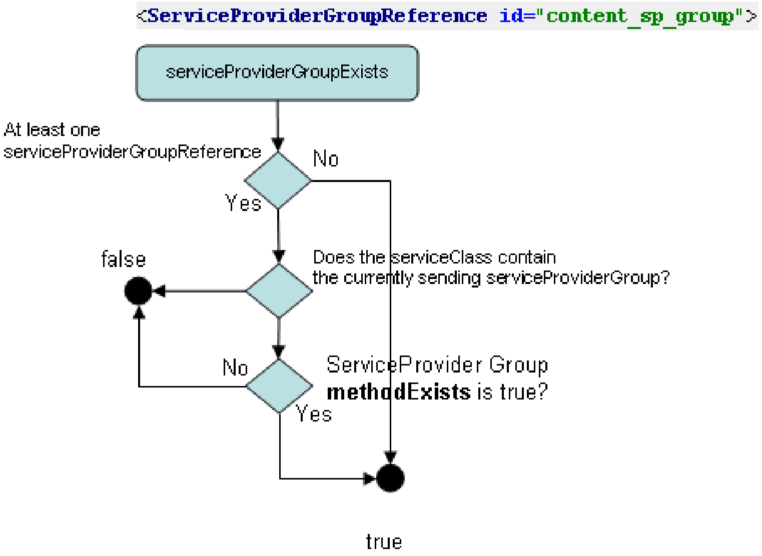
Figure 13-1 Application Group Reference Evaluation


<ApplicationGroupReference id="gold_app_group">
applicationGroupExists
At least one applicationGroupReference
No
Yes
false
Does the serviceClass contain the currently sending applicationGroup?
No
Application Group **methodExists** is true?
Yes
true

Note: The evaluation for **methodExists** is covered in Figure 13-3

– Is there a `ServiceProviderGroupReference` and is it relevant? See Figure 13-2.

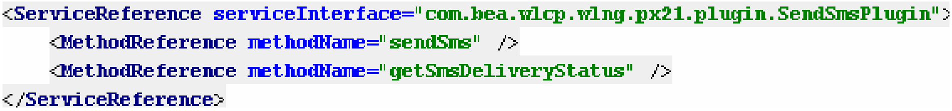
Figure 13-2 Service Provider Group Reference Evaluation


<ServiceProviderGroupReference id="content_sp_group">
serviceProviderGroupExists
At least one serviceProviderGroupReference
No
Yes
false
Does the serviceClass contain the currently sending serviceProviderGroup?
No
ServiceProvider Group methodExists is true?
Yes
true

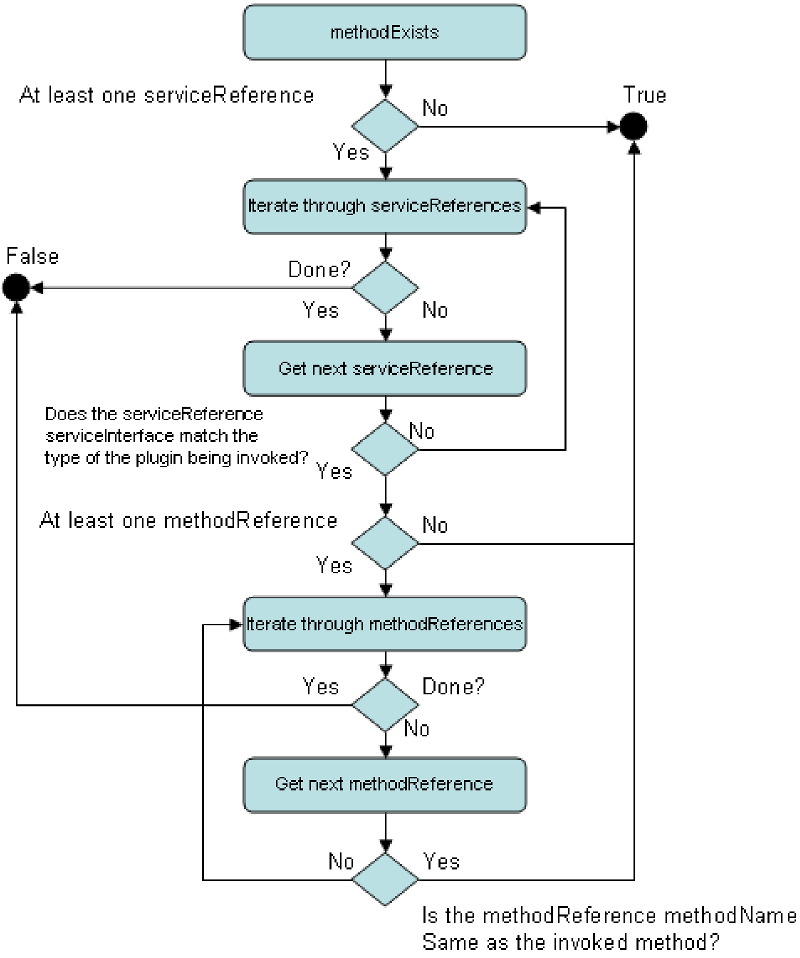
Note: The evaluation for **methodExists** is covered in Figure 13-3

– Is there a `Service Reference` (and possibly a `MethodReference`) and are they relevant? See Figure 13-3

Figure 13-3 Service and Method Reference Evaluation



```
<ServiceReference serviceInterface="com.bea.wlcp.wlng.px21.plugin.SendSmsPlugin">
    <MethodReference methodName="sendSms" />
    <MethodReference methodName="getSmsDeliveryStatus" />
</ServiceReference>
```


methodExists
At least one serviceReference
No
True
Yes
Iterate through serviceReferences
False
Done?
Yes
No
Get next serviceReference
Does the serviceReference serviceInterface match the type of the plugin being invoked?
No
Yes
At least one methodReference
No
Yes
Iterate through methodReferences
Yes
Done?
No
Get next methodReference
No
Yes
Is the methodReference methodName Same as the invoked method?

Is There Adequate Budget for the Contracts

Once the interceptor determines that an enforceable contract exists, it first determines whether the contract includes a `<restriction>` tag set to `<restrictAll/>`. If so, the request is immediately denied, and processing on the request ceases.

If the `<restriction>` tag is not set to `<restrictAll/>,` the decision flow here is identical to the other budget evaluations that take place in Oracle Communications Services Gatekeeper.

If there are no relevant contracts, or there are relevant contracts and there is adequate budget to cover them, budgets are adjusted as necessary and the request passes on to the next interceptor. If there are relevant contracts and there is not adequate budget to cover them, the request is denied.

Creating an EDR Listener and Generating SNMP MIBs

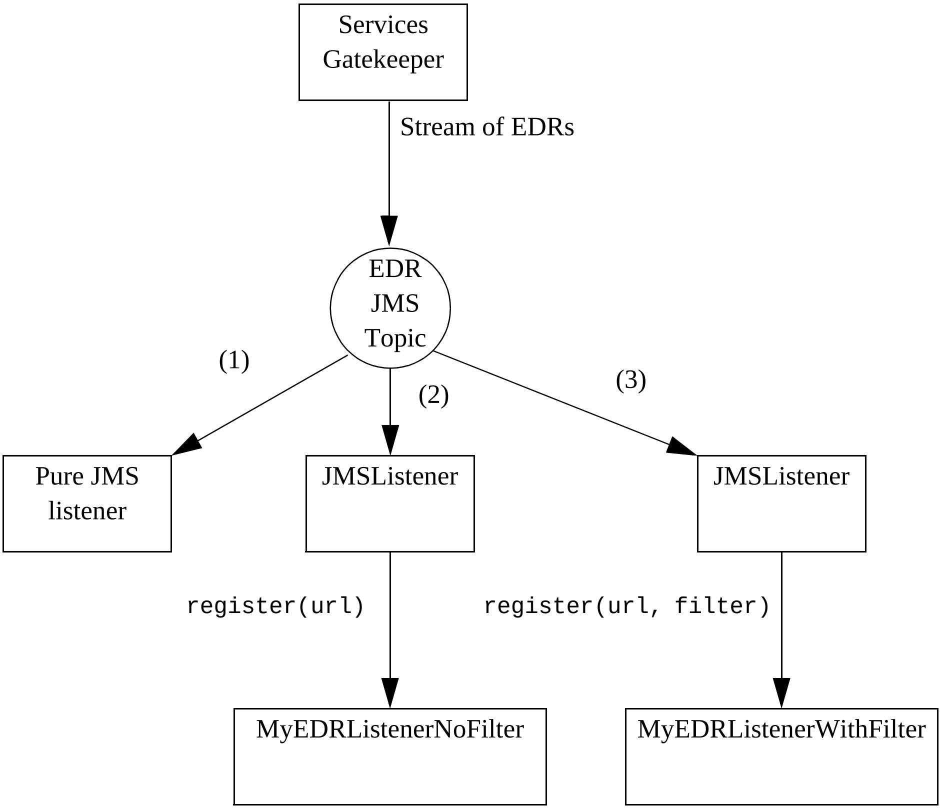
The following section describes how to create an external EDR listener.

Overview of External EDR listeners

External EDR listeners are JMS topic subscribers.

The diagram below illustrates three different ways of listening for EDRs as a JMS listener.

Figure 14-1 Flow for external EDR, alarm, and CDR listeners



EDRs are published externally using a JMS topic. This makes it possible to implement language-independent listeners anywhere on the network in a standard way. It is possible to implement an EDR listener in several ways:

- Alternative 1: Using a pure JMS listener. Implement the javax.jms.MessageListener interface. It is up to the implementation class to implement any filtering mechanism needed.

- Alternative 2: Using a subclass of JMSListener with no filter specified. In that case, the JMSListener class will use a tag, if available in the EDR, to filter the EDR into a specific category: EDR, alarm or CDR.
- Alternative 3: Using a subclass of JMSListener with a specified filter. This filter is used to perform the filtering. If a default filter is used to perform the same filtering as Oracle Communications Services Gatekeeper, all classes used in the xml configuration files must be present in the current class loader. Otherwise, some EDRs will not be correctly filtered.

Example using a pure JMS listener

Listing 14-1 Example using a pure JMS listener

```
public class ClientJMSListener implements MessageListener {
  public void onMessage(Message msg) {
    // Extract the EdrData object or array
    if(o instanceof EdrData[]) {
      for(EdrData edr : (EdrData[])o) {
        //do something with each EDR
      }
    }
  }
}
```

Example using JMSListener utility with no filter

Listing 14-2 Example using a subclass of JMSListener with no filter specified

```
public class SampleEdrJMSListener extends JMSListener {
  public SampleEdrJMSListener(String url) throws Exception {
    // Register in the JMS topic. No filter is specified so
    // the "tag" filtering mechanism will be used.
    register(url);
  }
  @Override
  public void onEdr(EdrData edr, ConfigDescriptor descriptor) {
    // The "tag" mechanism will filter the stream of EDRs according
    // to the internal filtering. To know which type of EDR is
    // actually provided in this method, we have to determine the
    // instance of the ConfigDescriptor as follow:
    if(descriptor instanceof EdrConfigDescriptor) {
      // do something with this EDR
    } else if(descriptor instanceof AlarmConfigDescriptor) {
      // do something with this alarm
    } else if(descriptor instanceof CdrConfigDescriptor) {
      // do something with this CDR
    }
  }
}
```

Using JMSListener utility with a filter

Listing 14-3 Using a subclass of JMSListener with a specified filter

```
public class SampleEdrJMSListener extends JMSListener {


  public SampleEdrJMSListener(String url) throws Exception {
    // Register in the JMS topic. Use the default alarm filter.
    // Note that in this case all classes needed by the alarm.xml file
    // must be in the current class loader in order for the filtering
    // to work correctly.
    register(url, EdrFilterFactory.createDefaultFilterForAlarm());
  }
  @Override
  public void onEdr(EdrData edr, ConfigDescriptor descriptor) {
    // Only AlarmConfigDescriptor should be received here.
    // Just check before casting.
    if(descriptor instanceof AlarmConfigDescriptor) {
      ... do something with this alarm
    }
  }
}
```

Note: When using the JMSListener class, make sure that any modification to an EDR, CDR. or alarms descriptor in Oracle Communications Services Gatekeeper is also updated in the edrjmslistener.jar file.

Description of EDR listener utility

The EDR listener utility contains a set of classes to use when creating an external JMS listener using the JMSListener.

The helper classes are found in the domain home directory in Oracle Communications Services Gatekeeper, in:

```
$ET_Home/lib/edrjmslistener.jar
```

Class JMSListener

Table 14-1 JMSListener

Method	Description
public void register(String url)	Registers the JMS listener to the EDR topic using no filter. The filtering will be done using the tagging mechanism. The parameter url specifies the URL of a Network Tier server.
public void register(String url, EdrFilter filter)	Registers the JMS listener to the EDR topic using the specified filter.
public void onEdr(EdrData edr, ConfigDescriptor descriptor)	Method that the subclass can override to get notified each time an EDR is received. The descriptor will be a subclass of ConfigDescriptor that will identify the type of EDR: either EdrConfigDescriptor, AlarmConfigDescriptor or CdrConfigDescriptor.

Class EdrFilterFactory

Table 14-2 EdrFilterFactory

Method	Description
public static EdrFilter createDefaultFilterForEdr()	Creates the default filter using in Oracle Communications Services Gatekeeper to filter the EDRs using the edr.xml file embedded in the edrjmslistener.jar file.
public static EdrFilter createDefaultFilterForAlarm()	Creates the default filter using in Oracle Communications Services Gatekeeper to filter the alarms using the alarm.xml file embedded in the edrjmslistener.jar file.
public static EdrFilter createDefaultFilterForCdr()	Creates the default filter using in Oracle Communications Services Gatekeeper to filter the CDRs using the cdr.xml file embedded in the edrjmslistener.jar file.

Class EdrData

This class contains all the values that an EDR (alarm and CDR) have.

Table 14-3 EdrData

Method	Description
public String getValue(String key)	Gets the value associated with the specified key.
public List<String> getValues(String key)	Gets the values associated with the specified key.

Class ConfigDescriptor

This class is the parent class of EdrConfigDescriptor, AlarmConfigDescriptor and CdrConfigDescriptor.

Class EdrConfigDescriptor

This class contains the data that is specified in the descriptors in the edr.xml configuration file: the identifier and the description.

Table 14-4 EdrConfigDescriptor

Method	Description
public long getIdentifier()	Returns the identifier of the EDR.
public String getDescription()	Returns the description of the EDR.

Class AlarmConfigDescriptor

This class contains the data that is specified in the descriptors in the alarm.xml configuration file: the identifier, the severity and the description.

Table 14-5 AlarmConfigDescriptor

Method	Description
public long getIdentifier()	Returns the identifier of the alarm.
public String getSeverity()	Returns the severity of the alarm.
public String getDescription()	Returns the description of the alarm.

Class CdrConfigDescriptor

This class identifies a CDR. This descriptor does not contain any additional data.

Updating EDR configuration files

If you are using external EDR listeners, and the alarm, CDR, or EDR descriptors have been updated in Oracle Communications Services Gatekeeper, the corresponding files need to be updated in `edrjmslistener.jar`. Update the corresponding xml file with the updated entries in the edr directory in `edrjmslistener.jar`.

Generating SNMP MIBs

Alarms can be forwarded as SNMP traps, see Managing and Configuring the SNMP service in *System Administrator’s Guide*.

The MIB file that corresponds to the alarms can be generated using the ant task `mibgenerator` defined in `com.bea.wlcp.wlng.ant.MIBGeneratorTask`.

The ant task is packaged in `$PDS_HOME/wlng/lib/ant-mib-generator.jar`

There is an example build file that uses the an task in `$PDS_HOME/integration`

When the alarms descriptor is changed, a new MIB should be generated and distributed to the SNMP clients. Copy the contents of the alarm descriptor and paste it into an xml file. Use this xml file when generating the MIB file.

CHAPTER 15

Converting Traffic Paths and Plug-ins to Communication Services

Traffic paths and network protocol plug-ins developed as extensions to Oracle Communications Services Gatekeeper 3.0 can be converted to communication services and deployed in this release using the procedure described in this section.

Plug-ins and traffic paths developed for Network Gatekeeper 2.2 and earlier should be re-engineered in order to take full advantage of the improvements of the platforms.

A pre-requisite is to have the source code for the traffic path and plug-in that is to be converted.

- Converting Network Protocol Plug-ins
- Converting Traffic Paths
- Checklist

Converting Network Protocol Plug-ins

The procedure for converting a plug-in for an existing communication service is:

1. Generate a a new plug-in using the Platform Development Studio Eclipse Wizard, see Using the Eclipse Wizard.
2. Copy the `src` directory of the plug-in to be converted to the `src` directory of the new plug-in.
3. If no MBean class is declared, remove the `javadoc2annotation` target from the `build.xml` file for the new plug-in.

Converting Traffic Paths

The procedure for converting a traffic path to a communication service is:

1. Generate a new common service with the same settings as the traffic path to be converted using the Platform Development Studio Eclipse Wizard, see Using the Eclipse Wizard.
2. Copy any customized part from the `traffic_path` directory for the traffic path to be converted to `common` directory for new communication service.
3. Copy the `src` directory of the plug-in to be converted to the `src` directory of the new plug-in.
4. If no MBean class is declared, remove the `javadoc2annotation` target from the `build.xml` file for the new plug-in.

Checklist

Below are a few items that should be verified during the conversion process:

- Make sure that the version used in the deploy targets in the main build.xml matches the one specified in the `common.xml` file.
- Make sure that the class specified in the property `plugin.class` defined in the build.xml for the plug-in is correct.
- Remove all references to `com.incomit.policy.DenyException` since it is not supported anymore.

CHAPTER 16

Policy

For most installations of Oracle Communications Services Gatekeeper, the ability rapidly and accurately to evaluate the status of requests in terms of Policy, or rules governing a variety of service characteristics, is one of the most important features that the system offers.

Note: Some evaluations, such as enforcement of SLAs, are performed by the Interceptor Stack. See Chapter 11, "Service Interceptors" for more details. The Policy system described in this chapter allows you to add additional types of evaluation to the request flow, including adding rules to be used for the Callable Policy Web Service.

If you extend Oracle Communications Services Gatekeeper, particularly if you add a new Communication Service, you may also need to make changes in the Policy system to cover new functionality that you have added. This chapter provides a very high level description of the process by which policy requests are processed and though which new rules can be added. It covers:

- Overview
- Policy Request Data
- Adding a New Rule
- Using RequestContext Parameters Defined in Service Level Agreements

Overview

When an application service request arrives at the service interceptor CreatePolicyRequestData, its parameters are put in a PolicyRequest object. The service interceptor EvaluateILOGPolicy evaluates these custom policy rules The rules themselves are written in the ILOG IRL language.

Policy Request Data

A `PolicyRequest` object has a standard form. The values in the object must be mapped to the variables in the Policy Rule that will be used to evaluate them. The Policy Request object can contain subsets of this standard data:

- `applicationID`: The Application ID of the requesting party
- `serviceProviderID`: The Service Provider ID of the requesting party
- `nodeID`: Used internally by Oracle Communications Services Gatekeeper - ignore
- `serviceName`: The name of the software module in which the policy request originates. Used in the rules to match to service contracts in the SLAs and to look-up any rules specific to the service.
- `methodName`: The name of the method which the request wishes to have executed. Access to this method is what is being evaluated.
- `serviceCode`: The service code provided by the application, which is written to CDRs for tracking purposes
- `requesterID`: An additional ID that may be provided by the application for tracking purposes. (dependent on the northbound interface being used)

All of the above are Strings.

- `transactionID`: Used internally by Oracle Communications Services Gatekeeper - ignore
- `noOfActiveSessions`: Used internally by Oracle Communications Services Gatekeeper - ignore
- `timeStamp`: The time the request was fed to the Policy Engine.
- `reqCounter`: The number of target addresses in the request. If only one target address is used in the request this value is set to 1. If using multiple target addresses in the request, it is the number of target addresses.

All of these are Longs.

In addition to these standard values, Policy Request objects contain all the parameters passed in from the application in its initial request, as `AdditionalParameters`, an array of `AdditionalDataValue`. An `AdditionalDataValue` consist of a name-value pair. The following data types can be defined in an `AdditionalDataValue` object.

- `intValue(int val):` Integer values
- `longValue(long val):` Long values
- `stringValue(String val)`: Strings.
- `stringArrayValue(String[] val)`: Arrays of String values.
- `booleanValue(boolean val)`: Boolean values.
- `shortValue(short val)`: Short values.
- `charValue(char val)`: Char values.
- `floatValue(float val)`: Float values.
- `doubleValue(double val)`: Double values.
- `intArrayValue(int[] val)`: Arrays of int values.

The name of the name-value pair is defined in the `dataName` member variable in the `AdditionalData` object. See Listing 16-1

Listing 16-1 Defining AdditionalData

```
AdditionalData adArray[] = new AdditionalData[1];
AdditionalDataValue targetAddressValue = new AdditionalDataValue();
AdditionalData adTargetAddressString = new AdditionalData();
targetAddressValue.stringValue(address);
adTargetAddressString.dataName = "targetAddress";
adTargetAddressString.dataValue = targetAddressValue;
adArray[0] = adTargetAddressString;
policyRequest.additionalParameters = adArray;
```

If any of the incoming parameters from the application are complex types, the objects are automatically examined and broken down into simple Java types. So, for example, the Parlay X 2.1 complex type `ChargingInformation` can contain a description, which is a string, a currency kind, which is also a string, an amount, which is a decimal number, and a code, which is a string. When the data is sent to the Policy Engine, it is broken down into a string value called `parameters.charging.currency`, another string value called `parameters.charging.code`, and so forth.

Adding a New Rule

New rules can be added to the Policy Service. The rule must have a `name` and a `priority.`

High priority rules are evaluated before low priority rules.There are a set of pre-defined priority levels, which are mapped to a numerical value:

- minimum, where the value is $-1*10^9$
- low, where the value is $-1*10^6$
- high, where the value is $1*10^6$
- maximum, where the value is $1*10^9$

Listing 16-2 shows the basic structure of a rule:

Listing 16-2 Skeleton of a rule

```
rule DenySubscriberNotExists
{
priority = high;
  when
  {
   // fetch the policy request data and perform evaluations.
  }
then
  {
      // Take action on
  }
};
```

Mapping PolicyRequest Data

In order to perform an evaluation, the data in the `PolicyRequest` object must be fetched by the rule in the Policy Engine and mapped to the equivalent variable name in the rule. The standard types of request data in the Policy Request are associated with variables of the same name in the rules. Below is an example of a rule assigning the `PolicyRequest` member variable `serviceName` to the rule variable `sname` via the Policy Request object. The rule object `pr` is assigned to the PolicyRequest object.

Listing 16-3 Policy Request data is fetched

```
?pr: event PolicyRequest(?sname: serviceName);
```

If the Policy Engine has evaluated the request and made the decision to deny it, the Policy Engine’s representation of the PolicyRequest object (`pr`) must be *retracted*. Retracting the PolicyRequest object aborts further rule enforcement.

Listing 16-4 Retract a request

```
retract (?pr);
```

If the Policy Engine has evaluated the request and made the decision to allow it, the Policy Engine’s representation of the request (`pr`) must still be retracted, but in the last rule of the execution flow. For example, this could be achieved by adding a general finalizing allow rule that retracts the request. This rule should have priority `minimum`.

Listing 16-5 General finalizing allow rule that retracts a request

```
rule AllowServiceRequest
{
  priority = minimum;
  when
  {
       ?pr: event PolicyRequest();

  }
  then
  {
       retract (?pr);
       ?pr.allow();

  }
};
```

Data that is defined as `AdditionalValues` must fetched as shown in Listing 16-6. The Additional Value named `targetAddress` is stored in the variable `addDataValue`. The PolicyRequest object is `pr`.

Listing 16-6 Fetching AdditionalValue data

```
bind ?addDataValue = ?pr.getAdditionalDataStringValue("targetAddress");
```

The particular signature of the fetching method depends on the type of data:

- `getAdditionalDataIntValue(...)`, for int values

- getAdditionalDataLongValue(...), for long value.
- getAdditionalDataStringValue(...), for String values
- getAdditionalDataStringArrayValue(...), for arrays of String values
- getAdditionalDataBooleanValue(...), for boolean values
- getAdditionalDataShortValue(...), for short values
- getAdditionalDataCharValue(...), for char values
- getAdditionalDataFloatValue(...), for float values
- getAdditionalDataDoubleValue(...), for double values
- getAdditionalDataIntArrayValue(...) for arrays of int values.

If the data type is unknown, it can be determined by invoking the discriminator method on the AdditionalDataValue object.

Listing 16-7 Determine the type of an AdditionalDatavalue

```
bind ?type = ?pr.getAdditionalData.dataValue.discriminator().value();
```

Where type is one of the following:

- AdditionalDataType._P_ADDITIONAL_INT
- AdditionalDataType._P_ADDITIONAL_LONG
- AdditionalDataType._P_ADDITIONAL_STRING
- AdditionalDataType._P_ADDITIONAL_STRING_ARRAY
- AdditionalDataType._P_ADDITIONAL_BOOLEAN
- AdditionalDataType._P_ADDITIONAL_SHORT
- AdditionalDataType._P_ADDITIONAL_CHAR
- AdditionalDataType._P_ADDITIONAL_FLOAT
- AdditionalDataType._P_ADDITIONAL_DOUBLE

- `AdditionalDataType._P_ADDITIONAL_INT_ARRAY`

Creating a New Rule File by Extending an Existing File: an Example

The following shows an example of extending an existing rule file:

1. List the Current Services' Rule Files
2. Select the Service Whose Rule File You Wish to Extend
3. Add a New Extended Rule
4. Load the New Rule File.

Use the operations in the PolicyService to manage the rule files, see Managing the PolicyService in the *System Administrator's Guide*.

Using RequestContext Parameters Defined in Service Level Agreements

It is possible to use generic data specified in service provider and application-level SLAs in a plug-in. This is useful when the choice of the action or behavior a plug-in should make is based on which service provider or application originates the request originates. For example, this can be used for information about parameters that corresponds to a certain group of applications. For instance a certain group might get the priority on their SMS set to LOW because they pay less. The priority might be a parameter that is sent down to the network which handles this.

In an SLA, a <contextAttribute> is defined as a name/value pair, where the name is defined in the tag <attributeName> and the value is specified in <attributeValue>.

A plug-in can retrieve the value specified in <attributeValue> using the name specified in <attributeName>. The value is retrieved using the RequestContext for the request:

```
String attributeValue =
(String)RequestContextManager.getCurrent().get("<attributeName>");
```

For example, the value associated with the contextAttribute with the attributeName com.bea.wlcp.wlng.plugin.sms.testName1 is retrieved using:

```
String value1 =
(String)RequestContextManager.getCurrent().get("com.bea.wlcp.wlng.plugin.s
ms.testName1");
```

Callable Policy Web Service

The following section describes how to use the callable policy interface exposed by Oracle Communications Services Gatekeeper.

Introduction

The callable policy service in Oracle Communications Services Gatekeeper exposes two Web Services interfaces related to callable policy:

- Policy evaluation
- Policy management

The callable policy service is intended to allow applications and network nodes that have no policy evaluation capabilities themselves to use the policy evaluation capabilities in Oracle Communications Services Gatekeeper. The service is not designed to expose the service to external service providers. Rather it is to be used internally as a way of exposing generic policy capabilities to network nodes within the telecom network where Oracle Communications Services Gatekeeper is deployed. Communication Services deployed in Oracle Communications Services Gatekeeper do not use the interfaces exposed by the callable policy Web Service.

For example, a node in the network might need to enforce a set of rules for requests flowing through it, to allow or deny requests based on time of day and originator of the request. In this case, the node might determine the originator of the request and use the callable policy evaluation Web Service to evaluate that request. The rule that is being evaluated uses the data provided in the web services call and makes its decision based on them. Modifications to the rules can be done using the policy management Web Service.

A user of the policy evaluation and policy management Web Services interfaces is registered using the same service provider and application model that is used for users of the Communication Services. If the system requires sessions, the user must be logged in using the same session manager interface exposed to these service provider applications.

Note: If there is no specific rule file associated with a ServiceName loaded in the rule engine, it uses the default rule file in its evaluation. If you are using Callable Policy, you must make sure that an appropriate rule file is loaded into the rule engine. For more information. see the "Managing the PolicyService" chapter in the *System Administration Guide*.

It necessary to have service provider group and application group Service Level Agreements defined for the user of the callable policy service. To use the policy evaluation interface, the tag `<scs>` must contain the value `com.bea.wlcp.wlng.px21.plugin.PolicyPlugin`.

To use the policy management interface, the tag `<scs>` must contain the value `com.bea.wlcp.wlng.px21.plugin.PolicyManagementPlugin`.

Listing 17-1 Example of SLA that allows the use of both the policy evaluation and policy management interfaces

```
<serviceContract>
  <scs>com.bea.wlcp.wlng.px21.plugin.PolicyPlugin</scs>
</serviceContract>
<serviceContract>
   <scs>com.bea.wlcp.wlng.px21.plugin.PolicyManagementPlugin</scs>
</serviceContract>
```

Callable Policy Web Service interface definition

Endpoints

The endpoint for the Policy evaluation interface is:

`http://<host:port>/callable_policy/Policy`

The endpoint for the Policy management interface is:

`http://<host:port>/callable_policy/PolicyManagement`

Detailed service description

Policy Evaluation

The policy evaluation interface makes it possible for an external application to evaluate a request containing a set of parameters. The parameters in the request include authentication information, information on the type of service the request should be evaluated against, the method name of the method that should be evaluated, and arbitrary additional data provided as name-value pairs.

All request parameters are evaluated according to a policy rule.

When evaluated, a copy of the data provided in the evaluation process is returned together with information on the outcome of the requests, that is, if the request was allowed or denied. If the request was allowed, the application calling the Web Service must use the returned copy of the parameters for further processing, because the returned parameters in the request may have been changed by the policy rule processing.

Policy management

The policy management web service interface makes it possible to load and delete policy rules.

XML Schema data type definition

AdditionalDataValue structure

Defines the AdditionalDataValue structure.

Element Name	Element type	Optional	Description
name	xsd:string	N	Name part of the additional data name-value pair.
value	xsd:string	N	Value part of the additional data name-value pair.
type	callable_policy_local_xsd:AdditionalDataValueType	N	Identifies the data type. See AdditionalDataValueType enumeration.

AdditionalDataValueType enumeration

Describes a data type.

Enumeration value	Description
STRING_TYPE	Data type is String.
INTEGER_TYPE	Data type is Integer.
FLOAT_TYPE	Data type is float.
DOUBLE_TYPE	Data type is double.
CHAR_TYPE	Data type is Char.
BOOLEAN_TYPE	Data type is boolean.

Enumeration value	Description
INT_ARRAY_TYPE	Data type is int array.
STRINGARRAY_TYPE	Data type is String array.

Interface: Policy

Operations to evaluate a request.

Operation: evaluate

The policy evaluation interface makes it possible for an external application to evaluate a request containing a set of parameters. All of the request parameters are evaluated according to a Policy rule.

Input message: evaluateRequest

Part name	Part type	Optional	Description
type	xsd:string	N	Service type to be evaluated.
serviceName	xsd:string	N	ServiceName associated with the rule file.
methodName	xsd:string	N	Name of method to be evaluated.
requesterID	xsd:string	N	The application ID as given by the operator.
timeStamp	xsd:dateTime	N	Defines the date and time of the request.
additionalData	callable_policy_local_xsd:additionalDataValue	Y	Specifies any other data, specified as name-value pairs. See AdditionalDataValue structure.

Output message: evaluateResponse

Part name	Part type	Optional	Description
modifiedRequest	callable_policy_local_xsd:evaluateRequest	N	The response that Oracle Communications Services Gatekeeper returns after being evaluated by policy rules. Same data structure as evaluateRequest, but data may have been changed by the policy evaluation.
returnValue	xsd:string	N	Return value the policy rules passed back.
thrownException	xsd:string	N	Name of the exception thrown during evaluation.
thrownPolicyException	xsd:string	N	Name of the policy rejection exception thrown during evaluation.
denyReasonDescription	xsd:string	N	Description of the reason of denying the request.
denyCode	xsd:string	N	Code identifying the reason of denying the request.

Referenced faults

ServiceException:

If there is an internal error during evaluation process, a ServiceException is thrown.

PolicyException:

If the policy evaluation request is rejected, a PolicyException is thrown.

Interface: PolicyManagement

Operations to manage policy rules.

Operation: viewRuleFile

Fetches a policy rule file of a given type and service from the rules engine.

Input message: viewRuleFile

Part name	Part type	Optional	Description
type	xsd:string	N	Type of SLA, either: • Application • Serviceprovider
serviceName	xsd:String	N	ServiceName associated with the rule file.

Output message: viewRuleFileResponse

Part name	Part type	Optional	Description
return	xsd:String	N	The rule file.

Referenced faults

ServiceException:

If there is an internal error during evaluation process, a ServiceException is thrown.

PolicyException:

If the policy evaluation request is rejected, a PolicyException is thrown.

Operation: deleteRuleFile

Deletes a policy rule file of a given type and service from the rules engine.

Input message: deleteRuleFile

Part name	Part type	Optional	Description
type	xsd:string	N	Type of rule file, either: • Application • Serviceprovider
serviceName	xsd:String	N	ServiceName associated with the rule file.

Output message: deleteRuleFileResponse

Part name	Part type	Optional	Description
-	-	-	-

Referenced faults

ServiceException:

If there is an internal error during evaluation process, a ServiceException is thrown.

PolicyException:

If the policy evaluation request is rejected, a PolicyException is thrown.

Operation: loadRules

Loads a a policy rule file of a given type and service into the rules engine.

Input message: loadRules

Part name	Part type	Optional	Description
type	xsd:string	N	Type of rule file, either: • Application • Serviceprovider
irlUrl	xsd:string	N	URL to rule file to be loaded.
serviceName	xsd:string	N	ServiceName associated with the rule file.

Output message: loadRulesResponse

Part name	Part type	Optional	Description
-	-	-	-

Referenced faults

ServiceException:

If there is an internal error during evaluation process, a ServiceException is thrown.

PolicyException:

If the policy evaluation request is rejected, a PolicyException is thrown.

Operation: listRuleFiles

Lists the rule files of a given type that are loaded into the rules engine.

Input message: listRuleFiles

Part name	Part type	Optional	Description
type	xsd:string	N	Type of rule file, either: • Application • Serviceprovider

Output message: listRuleFilesResponse

Part name	Part type	Optional	Description
ruleFile	Array of xsd:string	Y	A list of rule files matching the given criteria.

Referenced faults

ServiceException:

If there is an internal error during evaluation process, a ServiceException is thrown.

PolicyException:

If the policy evaluation request is rejected, a PolicyException is thrown.

Rule files

The rule files are written in IRL, ILog Rule Language.

When writing rules in the context of Oracle Communications Services Gatekeeper policy rules, the following apply:

The rule is associated with a service name when loaded into Oracle Communications Services Gatekeeper policy service, Input message: loadRules.

Which rule to be triggered by Input message: evaluateRequest is correlated with the parameter `serviceName` given in the Web Service request.

When the evaluate request triggers the rule, a set of general parameters can be accessed by the policy rule:

- String `applicationID`: Application ID associated with the request.
- String `serviceProviderID`: Service provider ID associated with the request.
- String `serviceName`: Service name from which the request originates or is destined for.
- String `methodName`: Method that triggered the request.
- String `serviceCode`.
- String `requesterID`.
- long `transactionID`.
- int `noOfActiveSessions`.
- `long timeStamp`: Time the request was sent to the rules engine for processing. Milliseconds from start of UNIX epoch.
- long `reqCounter`: Defines the increase rate for related counters.

A rule must have a name and a priority. High priority rules are evaluated before low priority rules.There are a set of pre-defined priority levels, which are mapped to a numerical value:

- minimum, where the value is -1*109
- low, where the value is -1*106
- high, where the value is 1*106
- maximum, where the value is 1*109

Listing 16-2 shows the basic structure of a rule:

Listing 17-2 Skeleton of a rule

```
rule DenySubscriberNotExists

{

        priority = high;

        when
        {
        // fetch the policy request data and perform evaluations.
        }
        then
        {
                // Take action on
        }
};
```

In order to perform an evaluation, the data in the `PolicyRequest` object must be fetched by the rule and mapped to the equivalent variable names in the rule. The standard types of request data in the Policy Request are associated with variables of the same name in the rules. Below is an example of a rule assigning the `PolicyRequest` member variable `serviceName` to the rule variable `sname` via the Policy Request object. The rule object `pr` is assigned to the `PolicyRequest` object.

Listing 17-3 Policy Request data is fetched

```
?pr: event PolicyRequest(?sname: serviceName);
```

If the Policy Engine has evaluated the request and made the decision to deny it, the Policy Engine's representation of the `PolicyRequest` object (pr) must be retracted. Retracting the PolicyRequest object aborts further rule enforcement.

Listing 17-4 Retract a request

```
retract (?pr);
```

If the Policy Engine has evaluated the request and made the decision to allow it, the Policy Engine's representation of the request (pr) must still be retracted, but in the last rule of the execution flow. For example, this could be achieved by adding a general finalizing allow rule that retracts the request. This rule should have priority minimum.

Listing 17-5 General finalizing allow rule that retracts a request

```
rule AllowServiceRequest
{
       priority = minimum;
       when
       {
              ?pr: event PolicyRequest();
       }
       then
       {
              retract (?pr);
              ?pr.allow();
       }
}
```

Data that is defined as AdditionalValues must fetched as shown below. The Additional Value named targetAddress is stored in the variable addDataValue. The PolicyRequest object is pr.

Listing 17-6 Fetching AdditionalValue data

```
bind ?addDataValue = ?pr.getAdditionalDataStringValue("targetAddress");
```

The particular signature of the fetching method depends on the type of data:

- `getAdditionalDataIntValue`(...), for int values
- `getAdditionalDataLongValue`(...), for long value.
- `getAdditionalDataStringValue`(...), for String values
- `getAdditionalDataStringArrayValue`(...), for arrays of String values
- `getAdditionalDataBooleanValue`(...), for boolean values
- `getAdditionalDataShortValue`(...), for short values
- getAdditionalDataCharValue(...), for char values
- `getAdditionalDataFloatValue`(...), for float values
- `getAdditionalDataDoubleValue`(...), for double values
- `getAdditionalDataIntArrayValue`(...) for arrays of int values.

If the data type is unknown, it can be determined by invoking the discriminator method on the `AdditionalDataValue` object.

Listing 17-7 Determine the type of an AdditionalDatavalue

```
bind ?type = ?pr.getAdditionalData.dataValue.discriminator().value();
```

`Where type is one of the following:`

- `AdditionalDataType._P_ADDITIONAL_INT`
- `AdditionalDataType._P_ADDITIONAL_LONG`
- `AdditionalDataType._P_ADDITIONAL_STRING`
- `AdditionalDataType._P_ADDITIONAL_STRING_ARRAY`
- `AdditionalDataType._P_ADDITIONAL_BOOLEAN`
- `AdditionalDataType._P_ADDITIONAL_SHORT`
- `AdditionalDataType._P_ADDITIONAL_CHAR`
- `AdditionalDataType._P_ADDITIONAL_FLOAT`

- `AdditionalDataType._P_ADDITIONAL_DOUBLE`
- `AdditionalDataType._P_ADDITIONAL_INT_ARRAY`

Checklist

This section contains a short summary checklist to use when creating extensions to Oracle Communications Services Gatekeeper:

- When creating the management interface, consider if the management operations and attributes should be cluster-wide or local.
- Make sure to follow the plug-in naming convention: `Plugin_<web service interface part>_<network protocol>`.
- Make sure to implement `customMatch` of the `PluginInstance` (or `ManagedPluginInstance`) to be sure that requests end up in the correct plug-in. This is important when there are multiple plug-ins for the same communication service.
- Create exception types that are very specific to various error scenarios. This will allow fine grain control of the alarms that are generated.
- Have a clean separation between the north and the south side of the plug-in.
- Make sure to return all north interfaces (callback included) and souths interfaces when implementing the `getNorthInterfaces()` and `getSouthInterfaces()` of `PluginInstance`.
- Make sure to implement the `resolveAppInstanceGroupdId()` method for each `PluginSouth` instance (if applicable).
- Annotate each parameter in the south object methods that you need to have when aspect calls back the `resolveAppInstanceGroupId()` or the `prepareRequestContext()` methods.

- Consider what additional EDR fields you need to add. Annotate all the methods you want to be woven using the `@Edr` annotation.
- Annotate the specific arguments you want to see in the EDR for each annotated methods. Use either `@ContextKey` or `@ContextTranslate` depending on the kind of argument.
- Add all the EDRs you are triggering to the EDR descriptor.