Oracle® Communications Services Gatekeeper

Extension Developer’s Guide

Release 6.1

E64626-01

September 2016

ORACLE®

Oracle Communications Services Gatekeeper Extension Developer's Guide, Release 6.1

E64626-01

Contents

Preface

This document describes the tools that Oracle Communications Services Gatekeeper includes to create and test new or customizes communication services.

Audience

This book is intended for system integrators and field engineers who need to extend the out-of-the-box functionality of Services Gatekeeper.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following documents in the Services Gatekeeper set:

- *Oracle Communications Services Gatekeeper Application Developer's Guide*
- *Oracle Communications Services Gatekeeper Communication Service Reference Guide*
- *Oracle Communications Services Gatekeeper OAuth Guide*
- *Oracle Communications Services Gatekeeper Portal Developer's Guide*
- *Oracle Communications Services Gatekeeper Platform Test Environment User's Guide*
- *Oracle Communications Services Gatekeeper Security Guide*

1

Overview of the Platform Development Studio

This chapter provides an overview of Oracle Communication Services Gatekeeper Platform Development Studio that you use to create or extend communication services.

About Platform Development Studio

Services Gatekeeper Platform Development Studio provides tools you can use to:

- Create communication services
- Customize existing communication services
- Integrate Services Gatekeeper with external systems

About Creating Communication Services and Plug-ins

You can expose your network's new features to partners using Platform Development Studio. Extension developers can then focus on only those parts of the system that correspond directly to their specific needs.

About Platform Development Studio Wizard

Platform Development Studio Wizard (PDS Wizard) is an Eclipse plug-in that streamlines the task of creating these Services Gatekeeper extensions:

- Communication Services
- Interceptor Modules
- OAuth2 Extension Handlers
- Platform Test Environment Custom Modules

For example, when you supply some basic naming information and the location of a Web Services Description Language (WSDL) file for each application-facing interface that the communication service supports, the wizard generates either a complete communication service project, or a network plug-in only project, as required. For more information about running the wizard, see "Creating Extensions with Platform Development Studio Wizard".

To get an understanding of the Services Gatekeeper features with which your communication service will interact, see these sections:

- Customizing SLA Behavior for a Service Provider or Application
- Understanding Aspects, Annotations, EDRs, Alarms, and CDRs

- Making Communication Services Manageable

Platform Development Studio Wizard offers XML-to-JSON mapping capability so you can map REST data to SOAP and back.

About the Example Communication Service

Platform Development Studio contains an example communication service that is ready to build and run. This example is based on a basic Web Service interface and a basic underlying network protocol. It demonstrates the entire range of tasks required to create your own communication service. For more information, see "Communication Service Example".

See "Service Enabler Example with SIP plug-in" for an example network protocol plug-in that uses the SIP Servlet container.

About Testing New Communication Services

You can test your communication service by using the Platform Test Environment (PTE) graphical user interface of the Platform Development Studio. PTE provides an extensible suite of tools for testing communication services and the Unit Test Framework, which includes the **WlngBaseTestCase** abstract base class which connects to the Platform Test Environment (PTE). There is a also complete set of sample tools created to interact with the example communication service. See *Services Gatekeeper Platform Test Environment User's Guide* for more information.

Integrating and Customizing Communication Services

You can customize the following services using Platform Development Studio:

- Intercepting Requests with Service Interceptors
- Creating Subscriber-centric Policies with SLAs
- Integrating Communication Services with External Systems

Intercepting Requests with Service Interceptors

Service interceptors offer an easy way to modify the request flow, simplify routing mechanisms for plug-ins, and centralize policy and SLA enforcement. Services Gatekeeper uses these modules as part of its internal functioning.

You can use service interceptors to intercept a service request and manipulate the request as it flows through a communication service. You can also choose to create new interceptors, or to rearrange the order in which the interceptors are used, in order to customize their functionality. See "Using Service Interceptors to Manipulate Requests" for more information.

Creating Subscriber-centric Policies with SLAs

Network operators can use the default Services Gatekeeper administration model to manage application service provider access to the network at increasingly granular levels. Network operators can also extend that model to encompass subscribers and offer them a highly personalized experience while protecting their privacy and keeping their subscriber data safe within your operator domain. See "Using SLA Policies to Manage Subscribers" for more information.

Integrating Communication Services with External Systems

You can use Platform Development Studio to integrate Services Gatekeeper with external network systems, including:

- EDR listeners. See "Creating EDR Listeners".
- Alarm monitoring using SNMP. See "Creating EDR Listeners".
- Short Messaging Peer to Peer Protocol (SMPP) API Java interface. See "Using the SMPP API".
- Universal Computer Protocol (UCP) API Java interface. See "Using the UCP API".

Additional integration points not covered in this document include:

- The Partner Relationship Management interfaces for creating Partner Management portals. See *Services Gatekeeper Portal Developer's Guide*.
- JMX for non-console based management, covered in these WebLogic Server documents:
  - *Oracle Fusion Middleware Developing Custom Management Utilities With JMX for Oracle WebLogic Server* at

    `http://docs.oracle.com/cd/E24329_01/web.1211/e24415/toc.htm`
  - *Oracle Fusion Middleware Designing Manageable Applications With JMX for Oracle WebLogic Server* at:

    `http://docs.oracle.com/cd/E24329_01/web.1211/e24416/designapp.htm`

2 Understanding Communication Service Components

This chapter describes the components, management, and use of the communication services used by Oracle Communications Services Gatekeeper. Also see "About Communication Services" in *Services Gatekeeper Communication Service Reference Guide* for a higher-level look at communication services.

Understanding Communication Service Components

A communication service consists of:

- A Service Web Service (SOAP or RESTful)
- A Service enterprise Java bean (EJB)
- A Callback EJB
- A Callback client EJB
- A set of network protocol plug-ins

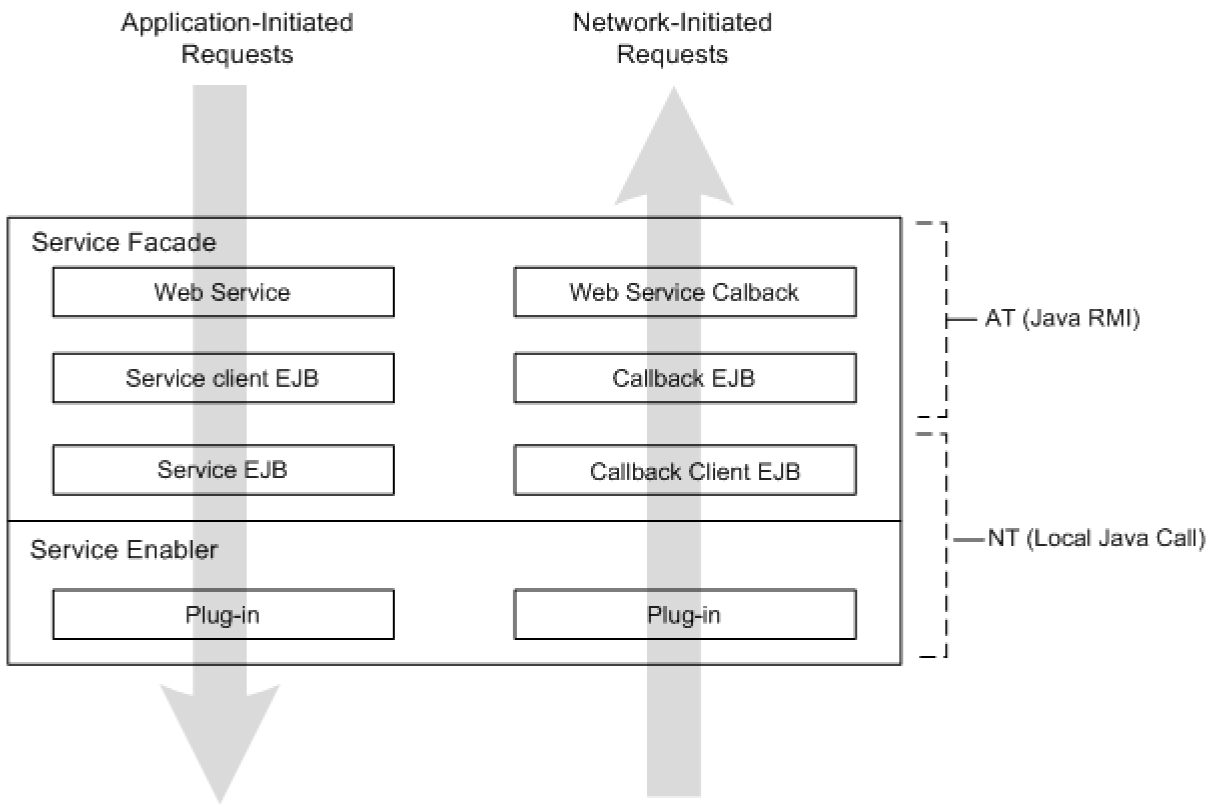
Service components, such as the Web service and service EJB, handle application-initiated requests.

Callback components, such as the callback EJB and callback client EJB, handle network-triggered requests.

When these components are deployed in separate clusters, remote calls are used to transport the request or response.

Figure 2–1 shows the high-level communication path used by application-triggered and network-triggered requests.

Figure 2–1 High-Level Components of a Communication Service



Communication service components are auto-generated and are based on one or more WSDL or WADL files. Application-initiated requests use service WSDL or WADL files. Network-triggered requests use callback WSDL or WADL files.

You use Platform Developer Studio Wizard to generate the communication service components listed in Table 2–1.

Table 2–1 Common Components of a Communication Service

Module	Description	North interface	South interface
Service Web Service	For application-initiated requests, the service Web service implements the (SOAP or RESTful) interfaces defined in the set of WSDL or WADL files for the specific service. The definition for a Web service is packaged into a single WAR file. For example, the SOAP Parlay X 2.1 Short Messaging service defines the **SendSms**, **ReceiveSms**, and **SmsNotificationManager** interfaces for application-initiated requests. The service Web service implements all the above interfaces and is packaged into one single WAR file for this communication service. 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 WSDL files	Java RMI representation of the service WSDL files

Table 2–1 (Cont.) Common Components of a Communication Service

Module	Description	North interface	South interface
Service EJB	The service EJB is packaged into a single JAR file for the communication service. When the Web service makes a Java RMI call to a service EJB, the service EJB uses the Plug-in Manager to locate and 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 WSDL files. The service EJB is responsible for: ▪ Constructing the **RequestInfo** object. ▪ Converting any exception caught to an exception that is defined in the service WSDL files. 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 file.	Java RMI representation of the service WSDL files	Local Java representation of the service WSDL files
Callback EJB	A Web services client that uses a Web service implemented by an application. It uses the interfaces defined in the set of callback WSDL or WADL files that define the Web service for network-triggered requests. 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 file.	SOAP/HTTP representation of the service callback WSDL file	Java RMI representation of the callback WSDL file
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 for cache invalidation 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 file.	Java RMI representation of the service callback WSDL file	Local Java representation of the callback WSDL file

Understanding Communication Service Plug-ins

As an extension developer, you can use the range of interfaces and classes provided in the **com.bea.wlcp.wlng.api.plugin.*** packages.

The first of these is a set of interfaces that define the borders of a plug-in and related helper classes. You can use these borders to apply aspects, the special constructs that contain several entities unavailable to standard classes in the definition of the plug-in and thereby provide flexibility in handling the request in transit. See "Understanding Aspects, Annotations, EDRs, Alarms, and CDRs" for information on aspects and the *Services Gatekeeper Java API Reference* for information on the **com.bea.wlcp.wlng.plugin** package.

Plug-in Service and Plug-in Instance

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

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

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

A plug-in instance is a class that implements the **com.bea.wlcp.wlng.api.plugin.ManagedPluginInstance** interface. Plug-in instances are part of the traffic flow. Each plug-in instance registers with the Plug-in Manager and manages the routing. It has:

- A life cycle defined in the **com.bea.wlcp.wlng.api.plugin.PluginInstanceLifecycle** interface.
- 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 the **com.bea.wlcp.wlng.api.plugin.PluginInstance** interface. 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**.

Understanding the Plug-in States

Plug-in services have the following states:

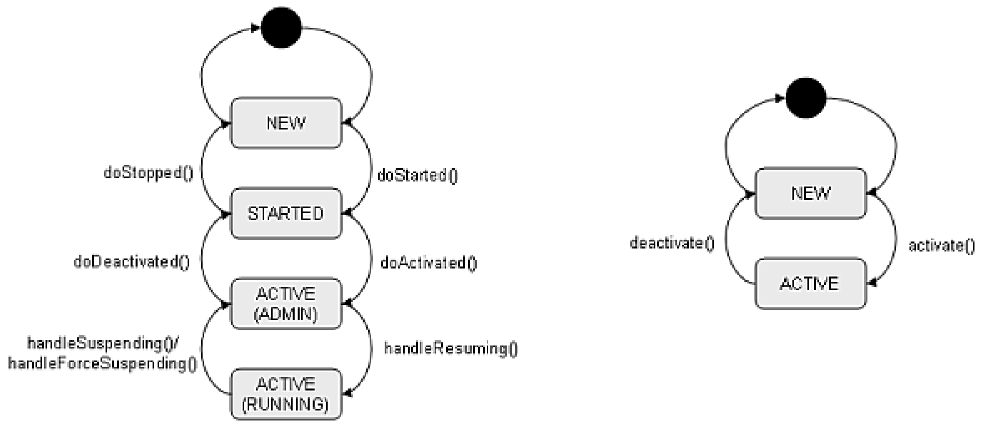
- NEW
- STARTED
- ACTIVE (ADMIN)
- ACTIVE (RUNNING)

Plug-in instances have the following states:

- NEW
- ACTIVE

Figure 2–2 shows the states used by the plug-in service and plug-in instance.

Figure 2–2 States for a Plug-in Service (left) and a Plug-in instance (right)



The state transitions in Figure 2–1 are triggered by:

- The start-up sequence of the server in which the plug-in is deployed.
- An explicit deployment of the plug-in using the **weblogic.Deployer**, a Java-based deployment tool that provides a command-line interface to the WebLogic Server deployment API. For details, see "Deploying Applications and Modules with weblogic.Deployer" in *Oracle Fusion Middleware Deploying Applications to Oracle WebLogic Server*.

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 2–2 lists the plug-in service state transitions, what they are triggered by and describes the state.

Table 2–2 Plug-in Service State Transitions

State 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: retrieving configuration data, creating internal objects, and initializing 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.

Table 2–2 (Cont.) Plug-in Service State Transitions

State Transition	Triggered by	Descriptions
`handleSuspending`	Graceful undeployment/redeployment/stopping That is, by invoking `weblogic.Deployer` with -graceful	The plug-in service should order the plug-in instance to reject new traffic, but continue processing of in-flight work. A **com.bea.wlcp.wlng.api.plugin.CompletionBarrier** type object 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** interface type object from the **CompletionBarrier** type object and call the **completed()** method on the **CompletionBarrierCallback** interface.
`handleForceSuspending`	Forced undeployment/redeployment/stopping 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.

Table 2–3 lists the state transitions, what they are triggered by and describes each state. These state transitions are triggered by either the start-up sequence of the server on which the plug-in instance is created, or are an explicit creation of a new instance using the Plug-in manager. See the discussion on "Managing and Configuring the Plug-in Manager" in the *Services Gatekeeper System Administrator's Guide*.

Table 2–3 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 state of the plug-in service is: ■ ACTIVE (ADMIN): The plug-in instance Instantiates and registers the **PluginNorth** and call-back interfaces with the plug-in manager. Instantiates and registers the **PluginSouth** interfaces with the plug-in manager. Instantiates any ConfigurationStore. Registers the MBean for the instance. ■ ACTIVE (RUNNING): The plug-in instance Connects to the network node, if a connection-oriented protocol is used. Registers any callbacks with the network node.
deactivate	Destruction of the plug-in instance using the Plug-in Manager MBean.	The plug-in instance: ■ De-registers any call-backs with the network node. ■ Disconnects from the network node, if connected. ■ De-registers the MBean for the instance. ■ Cancels any timers.

The Plug-in Manager maintains a pool of plug-in instances. This pool is provided to the plug-in when the **init()** method 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 maintains a registry of all **PluginNorth** and **PluginSouth** interfaces. 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 the 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 **PluginNorth** 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.

Understanding the PluginPool

The **PluginPool** interface is a collection of **PluginInstances** interfaces. **PluginPool** is populated when a plug-in instance is created using the **PluginInstanceFactory**. The plug-in service can use **PluginPool** to do the following:

- List plug-in instances
- Get a plug-in instance by its plug-in instance ID.

Obtaining Interfaces and Methods for a Plugin

You use the **getServiceInfo** operation to the PluginManagerMBean to obtain a list of all the interfaces and methods that a plugin offers. You can do this from the Administration Console, or by invoking the MBean directly. For example calling **getServiceInfo** with the service type **Payment-ParlayRestPayment** returns this (truncated) list of interfaces and methods:

```
Interface: oracle.ocsg.parlayrest.plugin.PaymentPlugin
  Method: startAmountReservationTransaction
    Arguments:
     java.lang.String arg0.amountReservationTransaction.clientCorrelator
     java.lang.String arg0.amountReservationTransaction.endUserId
     java.lang.String arg0.amountReservationTransaction.httpResponseCode
     java.lang.String
arg0.amountReservationTransaction.originalServerReferenceCode
     java.math.BigDecimal
arg0.amountReservationTransaction.paymentAmount.amountReserved
     java.math.BigDecimal
arg0.amountReservationTransaction.paymentAmount.chargingInformation.amount
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingInformation.code
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingInformation.currency
```

```
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingInformation.httpResponseCo
de
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.channel
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.httpResponseCode
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.mandateId
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.onBehalfOf
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.productId
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.purchaseCategoryC
ode
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.serviceId
     java.math.BigDecimal
arg0.amountReservationTransaction.paymentAmount.chargingMetaData.taxAmount
     java.lang.String
arg0.amountReservationTransaction.paymentAmount.httpResponseCode
     java.math.BigDecimal
arg0.amountReservationTransaction.paymentAmount.totalAmountCharged
     java.math.BigDecimal
arg0.amountReservationTransaction.paymentAmount.totalAmountRefunded
     java.lang.String arg0.amountReservationTransaction.referenceCode
     int arg0.amountReservationTransaction.referenceSequence
     java.lang.String arg0.amountReservationTransaction.resourceURL
     java.lang.String arg0.amountReservationTransaction.serverReferenceCode
     java.lang.String arg0.endUserIdPath
     java.lang.String arg0.httpResponseCode
    Return arguments:
     java.lang.String arg0.clientCorrelator
     java.lang.String arg0.endUserId
     java.lang.String arg0.httpResponseCode
     java.lang.String arg0.originalServerReferenceCode
     java.math.BigDecimal arg0.paymentAmount.amountReserved
     java.math.BigDecimal arg0.paymentAmount.chargingInformation.amount
     java.lang.String arg0.paymentAmount.chargingInformation.code
     java.lang.String arg0.paymentAmount.chargingInformation.currency
     java.lang.String arg0.paymentAmount.chargingInformation.httpResponseCode
     java.lang.String arg0.paymentAmount.chargingMetaData.channel
     java.lang.String arg0.paymentAmount.chargingMetaData.httpResponseCode
     java.lang.String arg0.paymentAmount.chargingMetaData.mandateId
     java.lang.String arg0.paymentAmount.chargingMetaData.onBehalfOf
     java.lang.String arg0.paymentAmount.chargingMetaData.productId
     java.lang.String arg0.paymentAmount.chargingMetaData.purchaseCategoryCode
     java.lang.String arg0.paymentAmount.chargingMetaData.serviceId
     java.math.BigDecimal arg0.paymentAmount.chargingMetaData.taxAmount
     java.lang.String arg0.paymentAmount.httpResponseCode
     java.math.BigDecimal arg0.paymentAmount.totalAmountCharged
     java.math.BigDecimal arg0.paymentAmount.totalAmountRefunded
     java.lang.String arg0.referenceCode
     int arg0.referenceSequence
     java.lang.String arg0.resourceURL
     java.lang.String arg0.serverReferenceCode
  Method: updateAmountReservationTransaction
.
.
```

.

See the individual communication service chapters in the *Services Gatekeeper Communication Service Reference Guide* for their service types.

See **PluginManagerMBean** in the “All Classes” section of the Services Gatekeeper OAM Java API Reference for details.

Understanding the Plug-in APIs

The **com.bea.wlcp.wlng.api.plugin** package contains interfaces and classes used for building a plug-in. A brief description of some of the most important interfaces and classes in this package is given here. See the *Services Gatekeeper Actions Java API Reference* for details.

Interface: Plug-in

Plug-in is the superinterface for "Interface: PluginNorth", "Interface: PluginNorthCallBack", and "Interface: PluginSouth".

These interfaces must be implemented by any plug-in that handles network-triggered requests, either new requests or notifications.

Interface: PluginNorth

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. All interfaces in the plug-in that implement the traffic interfaces contained in the service WSDL definitions must implement the **PluginNorth** interface. There must be one class per 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 *Services Gatekeeper Licensing Guide* for information about transaction units.

Interface: PluginNorthCallBack

PluginNorthCallback defines the limit between the plug-in and the service callback EJB and further on to an application.

All interfaces in the plug-in that implement the traffic interfaces contained in the service callback WSDL definitions must implement the **PluginNorthCallback** interface. Statistics aspects are applied for classes that implement this interface and counters for transaction units are increased. See *Services Gatekeeper Licensing Guide* for information about transaction units.

Interface: PluginSouth

PluginSouth defines the entry-point for network-triggered requests. It defines the south (or the network-facing) border of a plug-in. This interface must be implemented by the plug-in.

PluginSouth contains methods used to rebuild the object defined by "Interface: RequestContext" for network-initiated requests. The object is rebuilt using information from the object defined by "Interface: ContextMapperInfo" and the methods for resolving the application instance to which the request belongs.

When a network-triggered request arrives at the plug-in, the usual pattern is to correlate the request with a previous subscription for notifications.

Aspects that call the method

```
public String resolveAppInstanceGroupId(ContextMapperInfo)
```

can be applied by extending **PluginSouth** in the class that implements the request.

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** object. This is done using the Context aspect. See "Understanding the Context Aspect" for more information.

After resolving the application instance, the contextual information about the request is set up by calling the following method

```
public void prepareRequestContext(RequestContext ctx, ContextMapperInfo info)
```

In the implementation of this method, the plug-in instance has the option to add additional data to the contextual information about the request in the object defined by **RequestContext**.

Interface: ManagedPluginService

Every plug-in service must implement the **ManagedPluginService** interface which extends the **PluginService**, **PluginInstanceFactory** and **PluginServiceLifecycle** interfaces.

Interface: PluginService

PluginService defines a plug-in service registered in the Plug-in Manager.

A set of fields can 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 object. 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

PlugininInstanceFactory allows a plug-in service to create plug-in instances.

It defines the method:

```
ManagedPluginInstance createInstance(String pluginInstanceId)
```

This method is triggered by the **createPluginInstance** method on the Plug-in Manager MBean. When the **createInstance** method is invoked, the plug-in service is responsible for creating an instance of the class implementing the **ManagedPluginInstance** interface.

Interface: PluginServiceLifecycle

PluginServiceLifecycle defines the life-cycle for a plug-in service. See "Understanding the Plug-in States".

Interface: ManagedPluginInstance

ManagedPluginInstance extends the **PluginInstance** and **PluginInstanceLifecycle** interfaces. It must be implemented by a plug-in instance.

Interface: PluginInstance

PluginInstance defines a plug-in instance 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 lists are arrays of **PluginInterfaceHolder**. The respective lists are returned when the **getNorthInterfaces()** and **getSouthInterfaces()** methods are invoked.

The plug-in instance is also responsible for implementing the **customMatch(RequestInfo requestInfo)** method. This method examines the **RequestInfo** object and decides if the plug-in instance can be used to serve the request. It returns the following constants:

- MATCH_OPTIONAL

  The request is completely stateless and can be served by any plug-in instance.

- 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.

The plug-in instance is also responsible for maintaining information on the connection status with the network node to which it is connected. It returns True or False when the **isConnected()** method is invoked.

The Plug-in Manager invokes the methods in this interface to select the plug-in instance to which it must route the request.

Interface: PluginInstanceLifecycle

PluginInstanceLifecycle defines the life cycle for a plug-in service. See "Understanding the Plug-in States".

Class: RequestFactory

The **RequestFactory** class processes application-initiated requests both before and after a request is processed in the plug-in. Each communication service must have one implementation of the **RequestFactory** for each application-facing interface, named according to the pattern: *interfacename***.PluginFactory**, where *interfacename* is the name of the application-facing interface. A skeleton for the factory is generated by the Platform Developer's Studio Wizard plug-in.

The **RequestFactory** class has two main functions:

- It packages routing information contained in the request into a **RequestInfo** object that the Plug-in Manager then uses to select an appropriate plug-in to process the request. See "Class: RequestInfo" 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 `RequestInfo` object is matched to a plug-in. 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 details see **RequestFactory** in the "All Classes" section of the *Services Gatekeeper Java API Reference*.

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 in the callback WSDL definitions.

The naming pattern is
`com.acompany.example.callback.`*interfacename*`CallbackFactory`

Where *interfacename* is name of the interface (such as Notification in Example 2–1). The implementation of the interface is fetched using the following pattern:

Example 2–1 Example CallbackFactory Implementation

```
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

The **Callback** interface is used by a plug-in to propagate a network-triggered request from the plug-in to the callback EJB. This interface defines a Java representation of the methods in the callback WSDL definitions. There is one of these per interface in the callback WSDL definitions.

The naming pattern is **com.acompany.example.callback.***interface name***Callback**, where *interfacename* is name of the interface.

Class: RequestInfo

The **RequestInfo** object is 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

ServiceType 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.

When they are used, aspects make this service type available in the request thread of each plug-in. The service type is used by various services, including the **EdrService**. See *Services Gatekeeper Application Developer's Guide* for the current set of supported interfaces and plug-ins.

Plug-in Context APIs

The **com.bea.wlcp.wlng.api.plugin.context** package contains interfaces and a class used for providing context for a plug-in. Brief descriptions of the most important interfaces and classes in this package are given below. See *Services Gatekeeper Java API Reference* for the complete Javadoc.

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 are needed by the plug-in for correlation purposes and places them in this very short-lived object. Arguments are stored by key and defined when the annotation is set, making it possible to retrieve a particular value. For example, if an argument is annotated with **@MapperInfo(C)**, its value can be retrieved using the key "**C**".

Methods in the plug-in can access this **ContextMapperInfo** object and retrieve the annotated arguments in order to perform a mapping such as associating a notification with the session ID of the request that established it.

The **PluginSouth** interface includes one such method, **resolveAppInstanceGroupdId**.

Interface: RequestContext

This interface defines a **RequestContext** object which is available in all communication services for both application-initiated and network-initiated requests. This object 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.

Managing Communication Services

These are base classes and annotations for giving the Services Gatekeeper Administration Console or other JMX tools management access to communication services. See "Making Communication Services Manageable" for more information. Also see the *Services Gatekeeper Java API Reference* for the **com.bea.wlcp.wlng.api.management** package documentation.

Managing Communication Service Access with SLAs

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 `<scs>` element defined in the `<serviceContract>` element 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* depends on the WSDL definition for 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.

Sharing Libraries Among Communication Service Plug-ins

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.

3

Developing Communication Services

This chapter provides a high-level description of how to develop communication services using Oracle Communications Services Gatekeeper. It also provides an overview of other parts of the API available to extension developers.

The Javadoc for the container API is available with the Services Gatekeeper documentation. See the "All Classes" section of the *Services Gatekeeper Java API Reference* for details on the individual MBeans shown in this chapter. These javadoc files are also available from the *Middleware_home*/**ocsg_pds/doc/javadoc** directory of the Platform Development studio installation, where *Middleware_home* is the directory in which Oracle WebLogic Server is installed.

Tips for Creating or Extending Communication Services

The following tips are useful to consider when creating or extending communication services in Services Gatekeeper:

- Make sure to follow the naming convention for the plug-in:

  `Plugin_`*web service interface part_network protocol*

- Make sure to implement **customMatch** method of the **PluginInstance** (or **ManagedPluginInstance**) interface to verify if the plug-in can be used. 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 allows fine grain control of the alarms that are generated.
- Have a clean separation between the application-facing and the network-facing sides of the plug-in.
- Make sure to return all application-facing interfaces (callback included) and network-facing interfaces when implementing the **getNorthInterfaces** and **getSouthInterfaces** methods of **PluginInstance**.
- Make sure to implement the **resolveAppInstanceGroupdId** method for each **PluginSouth** instance (if applicable).
- When creating the management interface, consider if the management methods and fields should be cluster-wide or local.
- Annotate the following:
  - Each parameter in the south object methods that you need to have when aspect calls back the **resolveAppInstanceGroupId** or the **prepareRequestContext** methods.

- All the methods you want to be woven using the **@Edr** annotation, when you add additional EDR fields.
- 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.

See "Understanding Aspects, Annotations, EDRs, Alarms, and CDRs" for more information on annotation.

- Add all the EDRs you are triggering to the EDR descriptor.

Communicating with Container Services

The container service APIs enable communication between a communication service and the container services.

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

For a network protocol plug-in of a communication service to interact with Services Gatekeeper, it must be deployable in the context of Services Gatekeeper. After it is deployable, it can have access to certain utility functions.

Table 3–1 lists and describes the container services API packages.

Table 3–1 Summary of the Container Services APIs

Package	Description
com.bea.wlcp.wlng.api.account	Represents an application instance and the related accounts, groups, and the states of the accounts.
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 "Understanding Aspects, 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 "Broadcasting Events".
com.bea.wlcp.wlng.api.interceptor	Interfaces and classes for service interceptors. See "Using Service Interceptors to Manipulate Requests".
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 "Using the Plug-in Packages".
com.bea.wlcp.wlng.api.servicecorrelati on	Interface to implement when you extend the existing service correlation mechanism. See "Correlating Services".
com.bea.wlcp.wlng.api.statistics	Annotation for statistics. See "Generating Statistics with Statistics Service".

Table 3–1 (Cont.) Summary of the Container Services APIs

Package	Description
com.bea.wlcp.wlng.api.storage	Interfaces and classes for the storage service. See "Understanding Service Gatekeeper 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.

For complete documentation of these APIs, see the *Services Gatekeeper Java API Reference*.

Retrieving Implementation Instances Using InstanceFactory

Services Gatekeeper retrieves instances of a specified interface, class, or abstract class by using **InstanceFactory**. You retrieve an instance of the Instance Factory using the public static method **getInstance**. The factory itself has a single method called **getImplementation** which retrieves a class that implements a given interface or extends a given class.

To use an implementation:

1. Locate the list that maps a given interface, class, or abstract class to the preferred implementation of that functionality. This list is provided in the JAR file's **instancemap**, a standard **java.util.Properties** file. Every JAR file can have its own **instancemap**. See Example 3–1 for an example.

   Note: The interface name used in the **instancemap** file 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, instantiate the public constructor or static singleton method in the target class.
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.

Example 3–1 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 **InstanceFactory** in the "All Classes" section of the Services Gatekeeper Java API Reference.

Obtaining JNDI Context with ClusterHelper

ClusterHelper is a helper class in the **com.bea.wlcp.wlng.api.util.cluster** package. It is used to obtain the JNDI Context for the network and access tier.

For details see **ClusterHelper** in the "All Classes" section of the *Services Gatekeeper Java API Reference*.

Broadcasting Events

You can use **EventChannel**, a utility service interface, to broadcast events by other Services Gatekeeper server instances and register listeners for events originating in other Services Gatekeeper server instances. An event has a name and a value, where the name is an identifier for the event and the value is any object implementing the **java.io.Serializable** interface.

Retrieve an **EventChannel** instance using **com.bea.wlcp.wlng.api.event_ channel.EventChannelFactory**. Use the methods of the **EventChannel** interface to do the following:

- Deactivate all registered listeners (**deactivateAllListeners**)
- Publish an event to all registered listeners (**publishEvent**)
- Publish an event to one Services Gatekeeper instance (**publishEventToOneNode**)
- Register an EventListener (**registerEventListener**)
- Unregisters an EventListener (**unregisterEventListener**)

Use the **EventChannelListener** interface to receive events published using EventChannel. It contains the **processEvent(String eventType, Serializable event, String source)** method which receives an event.

Generating Statistics with Statistics Service

You can generate standard statistics or exception-related statistics. In addition to this, you can generate custom statistics explicitly. Register the type of statistics by using the **addStatisticType** operation in the Administration Console. For extensions, the statistics ID should be in the range 1000 to 2250.

Generating Standard Statistics

Standard statistics are generated automatically when a plug-in implements **PluginNorth** and **PluginNorthCallBack** interfaces.

The **@Statistics** annotation generates a statistics event when the method returns. It is defined in the **com.bea.wlcp.wlng.api.statistics.Statistics** package. The syntax of the annotation is:

`@Statistics(id=`*My_Statistics_Type*)

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

Generating Statistics when Exceptions are Thrown

The **@ExceptionStatistics** annotation generates a statistics event if an exception is thrown. It is defined in the **com.bea.wlcp.wlng.api.statistics.ExceptionStatistics** package. The syntax of the annotation is:

`@ExceptionStatistics(id=`*My_Statistics_Type*)

For more information, see the discussion on “Managing and Configuring Statistics and Transaction Licenses” in *Services Gatekeeper System Administrator’s Guide*.

Using the Plug-in Packages

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

See "Understanding Communication Service Components" for more information.

Understanding Communication Service Management

Base classes and annotations for giving the Services Gatekeeper Administration Console or other JMX tools management access to communication services. See "Making Communication Services Manageable" for more information. Also see the **com.bea.wlcp.wlng.api.management**.* packages in *Services Gatekeeper Java API Reference*.

Understanding EDRs

See "Understanding Aspects, Annotations, EDRs, Alarms, and CDRs" for details on EDRs. Also see the **com.bea.wlcp.wlng.api.management**.* packages in *Services Gatekeeper Java API Reference*.

Enforcing Service Level Agreements

Service Level Agreement (SLA) enforcement operates on methods identified by the Java representation of the interface, and the method on the application-facing interface for the communication service or the service type of the communication service. Note that:

- The content of the `<scs>` element defined in the `<serviceContract>` element 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 file 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 Platform Developer’s Studio Wizard. SLA enforcement for service types relates to quotas and request rates and are defined under the `<serviceTypeContract>` element.

For enforcement of custom SLAs, see "Creating Custom Runtime SLAs".

Correlating Services

Service providers often bundle separate services into a single unit for charging purposes. For example, a subscriber sends an SMS to the provider requesting the location of the coffee shop closest to her current location. In completing that request, the application provides three services. It receives the network-initiated SMS, performs a user location lookup on the customer and finally, sends the customer an MMS with a map showing the requested information. Services Gatekeeper provides the framework for a Service Correlation service that uses a service correlation ID (SCID) to combine/correlate all three Services Gatekeeper services into a single service charging unit.

About Service Correlation Identifiers

This service correlation ID is a string that is captured in all CDRs and EDRs generated by Services Gatekeeper. It is propagated in the SOAP header sent to and from the application. The CDRs and EDRs use this data to provide special treatment for a given chain of service invocations, such as applying charging to the chain as a whole rather than to the individual invocations.

Services Gatekeeper does not provide the SCID and it does not check whether the SCID is unique. The SCID is stored in the Oracle Label Security (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**. The application or by an external mechanism that the communication service provides the SCID in the following way:

- When the chain of services is initiated by an application-initiated request, the application provides and ensures a unique SCID within the chain of service invocations.

  Note: If a custom service correlation service supplies the SCID, it is the responsibility of the custom service to ensure the uniqueness of the SCID.

  When the application-initiated request traffic enters the plug-in, the Service Correlation service takes the SCID from the **WorkContext** interface instance and places it in the **RequestContext** class object, where it will be available to the EDR service.

- When the chain of services is initiated by a network-triggered request, Services Gatekeeper calls an external interface to get the SCID. This interface must be implemented by an external system. No utility or integration is provided out of the box; this must be a part of a system integration project. It is the responsibility of the external system to provide a unique SCID.

  When the network-initiated request traffic leaves the plug-in, the Service Correlation service takes the SCID from the **RequestContext** object and places it in the **WorkContext** object, where it can be retrieved by the SOAP Handler and passed along to the application.

Managing Service Correlation Identifiers

A communication service needs to create a way of storing and retrieving the Service Correlator ID (SCIDs), because a **RequestContext** object exists only for the lifetime of a single request and SCIDs may need to be stored across several invocations.

To store and retrieve SCIDs, you use the following:

- **ExternalInvocation** interface

  The **ExternalInvocation** interface has two methods, one to store the Service Correlation ID and the other to retrieve it. The **ExternalInvocation** implementation class should have an empty public constructor or a static method that returns itself. The implementor is free to modify the ID once it has been stored, or to use the **ExternalInvocation** object to create IDs in the first place.

  For an application-initiated request, the Service Correlation service takes the SCID (should there be one) out of the **WorkContext** of the request, and 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 verifies the SCID before storing it in the **WorkContext**. The 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. In this way, if the **ExternalInvocation** object has modified the SCID in any way, the modified version is put in the **WorkContext** and sent on to the application.

- **ExternalInvocationFactory** class

  The **ExternalInvocationFactory** class is used by the Service Correlation service to locate and instantiate the correct **ExternalInvocation** object. It does this by using an instancemap entry such as:

  ```
  com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocation=myPackageSt
  ructure.myImplClass
  ```

  where, *myImplClass* is the **ExternalInvocation** implementation.

- **ServiceCorrelation** package

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

Creating a Custom Service Correlation

To create a custom service correlation:

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

Example 3–2 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 using **ExternalInvocationFactory**.
3. Put the instancemap file in the JAR file. This makes your custom service correlation available to the service interceptor InvokeServiceCorrelation.
4. Put the JAR file in *Domain_Home*/**lib**.

Using Parameter Tunneling

Parameter tunneling is a feature that allows an application to send additional parameters to 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 application-facing interface and can be seen as an extension to it.

See "Dynamically Customizing AVPs for Applications" for more information and an example SOAP header.

Understanding Service Gatekeeper Storage Services

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

- Storing Configuration Data with ConfigurationStore
- Storing Traffic Data with StorageService

Storing Configuration Data with ConfigurationStore

The 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 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 the 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.

Example 3–3 is an example of using the Configuration Store.

Example 3–3   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);
  }
}
```

Storing Traffic Data with 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. See "Managing and Configuring Storage Service" in *Services Gatekeeper System Administrator's Guide* for an overview.

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 interface, 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. An index specified for the Store 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 is not notified if the extension itself removes entries from the Store.

- Cluster locking

  Cluster-wide locking can be done using the Store interface. If the same entry in a Store may be modified on multiple servers at the same time, use cluster locking to avoid errors from 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.

The storage provider offers a set these store types which are described in "Understanding the Storage Service" in *Services Gatekeeper System Administrator's Guide.*

- Cluster store
- Write-behind database store
- Write-through database store
- Refresh-ahead database store
- Database log store

All stores except for the cluster store are backed by a database table that is configured in a store configuration file.

When choosing a store type, take into consideration what kind of data that will be stored, how often it is written and read, and how long the data will stay in the store.

In general, if the lifetime for data is short enough that having the data duplicated in memory on two servers in the cluster, the cluster cache type should provide sufficient persistence. In other cases, a trade-off can be made between the data integrity transaction synchronized write-through operation gives and the performance given by asynchronous write behind. For data that just needs to be added to a database table, and is never read, the database log store is recommended. This store type could be optimized to avoid keeping cache entries in memory that will never be read anyway.

Table 3–2 outlines the recommendations on how to choose a store type.

Table 3–2 Store Type Recommendations

Access Type	Lifetime of Data	Store Type
Read mostly	Short	Cluster store
Read mostly	Long	Write-through database store
Write mostly	Short	Cluster store
Write mostly	Long	Write-behind database store
Write only	Any	Database log store
Read and Write	Short	Cluster store
Read and Write	Long	Write-behind database store

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 for each plug-in. Each configuration store configuration file together with its XSD and any complex data types should be stored, created, and packaged in a JAR file in the directory *domain_Home*/**config/store_schema**, where *domain_Home* is the home directory of the Services Gatekeeper domain. The configuration file must be named **wlng-cachestore-config-extensions.xml** and it must be present in the root of the JAR file.

For details about the store configuration file, see the corresponding xsd: **com.bea.wlcp.wlng.storage_6.0.0.0.jar/wlng-cachestore-config.xsd** in *Middleware_home*/**ocsg/modules** where *Middleware_home* is the home directory of the WebLogic Server domain.

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(); }.

Example 3–4 shows how to retrieve a store identified by key/value classes, operate on it, and release it.

Example 3–4 Using the Store Interface

```
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.

Example 3–5 Retrieving a store by name

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

Store configuration file

The **wlng-cachestore-config-extensions.xml** configuration file 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_6.0.0.0.jar/wlng-cachestore-config.xsd** in *Middleware_home*/**ocsg/modules**.

There is one configuration file for each plug-in. The file must be embedded in a JAR file that contains the file itself and any complex data types used. The JAR file must be stored in *domain_Home*/**config/store_schema**.

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

Example 3–6 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 `<store-config>` and `</store-config>` tags.

Each Store consists of the following elements:

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

Which cache type to use depends on the use case. A store is identified by a class name. The type is given by adding the store type ID prefix followed by an identifier for the store. For example, the store **wlng.db.wt.example_store_notification** uses the cache type **wlng.db.wt**. Table 3–3 describes the correlation between a store type and a store type ID prefix.

Table 3–3 Store Types and Store Type ID

Store Type	Store type ID prefix
Write behind database store	wlng.db.wb.
Write through database store	wlng.db.wt.
Cluster store	wlng.cache.
Database log store	wlng.db.log.

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

The `<store>` element 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 **key-class** attribute and the class for the value part is defined in the **value-class** attribute. If a named store is used, the name is given in the `<name>` element.
- `<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>` element defines the database table used to persist data in store. The **name** attribute defines the table name to use. This name must be the same as the **db_table_name** specified in the `<store>` element. It contains the following elements:

- `<key_column>`: Has the **name** and **data_type** attributes. The **name** attribute specifies the column name for the key and **data_type** specifies the SQL data type for the key.
- `<multi_key_column>`: Has the **name** and **data_type** attributes. The **name** attribute specifies the column name for one part of a multi-key column and **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 two 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 **name** attribute. 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>`: Used if attributes in the value part of the store should be stored in a separate column. The **name** attribute defines the name of the column and the data_type specifies the SQL data type for the column. The `<value_column>` element contains the `<methods>` element, which encloses the `<get_method>` and `<set_method>` elements. The `<methods>` element 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.

Example 3–7 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>
...
```

Example 3–8 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 support a StoreQuery interface.

The `<query>` element specifies a named query and a filter associated with the named query. The attribute name defines the name of the query. The behavior of these named queries are configured as part of the Storage Service configuration files. 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 `<filter-class>` element. The filter implements the setParameters method 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.

Example 3–9 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>
```

Example 3–10 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();
```

Example 3–11 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>` element contains definitions of which storage provider a given type-id is mapped to. This element should 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>".

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>` element contains mappings between the **provider-name** defined in the `<provider-mapping>` element and the factory class for the storage provider. This element should not be changed used unless a custom storage provider is used.

Sharing Common 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 file. All classes in this directory are available for all of the plug-ins in the EAR file.

4 Communication Service Example

This chapter describes the example communication service provided with the Oracle Communications Services Gatekeeper Platform Development Studio.

Overview of the Example Communication Service

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.

SendData and **NotificationManager** are used by an application and implemented by the communication service.

Notification is used by the communication service and implemented by an application.

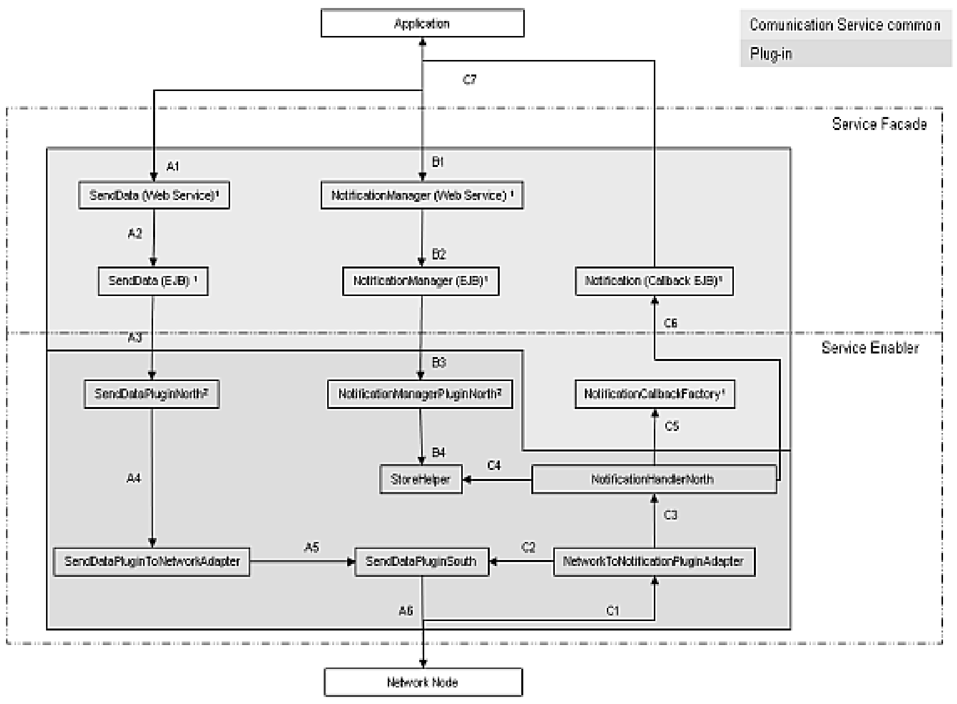
The communication service to network node interface is a simple TCP/IP based interface that defines the two commands:

- **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 4–1 illustrates the flow for these operations.

Figure 4–1 Overview of example Communication Service


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

This is the application-facing interface for the example communication service.

Interface: SendData

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

Operation: sendData

Send data to the network.

Input message: **sendDataMessage**

Table 4–1 sendDataMessage parts

Part name	Part type	Optional	Description
data	xsd:string	N	The data to be sent to the target device

Table 4–1 (Cont.) sendDataMessage parts

Part name	Part type	Optional	Description
address	xsd:anyURI	N	Address of the target device. Example: tel:4154011234

Output message: sendDataResponse

Table 4–2 sendDataResponse parts

Part name	Part type	Optional	Description
none	N/A	N/A	N/A

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**

Table 4–3 startEventNotificationRequest parts

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:anyURI	N	Service activation number. Example: tel:4154567890

Output message: **invokeMessageResponse**

Table 4–4 invokeMessageResponse parts

Part name	Part type	Optional	Description
none	N/A	N/A	N/A

Operation: stopEventNotification

Stop the subscription of event notification from the network.

Input message: **stopEventNotificationRequest**

Table 4–5 stopEventNotificationRequest parts

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

Output message: **stopEventNotificationResponse**

Table 4–6 stopEventNotificationResponse parts

Part name	Part type	Optional	Description
none	N/A	N/A	N/A

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**

Table 4–7 notifyDataReceptionRequest parts

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

Output message: **notifyDataReceptionResponse**

Table 4–8 notifyDataReceptionResponse

Part name	Part type	Optional	Description
none	N/A	N/A	N/A

Network Interface Definition

This is the network-facing interface for the example communication service.

sendDataToNetwork

Send data from the communication service to the network node.

Table 4–9 sendDataToNetwork arguments

Argument	Type	Description
fromAddres s	String	The address from which the request is sent.
toAddress	String	The address to which the request shall be sent.

Table 4–9 (Cont.) sendDataToNetwork arguments

Argument	Type	Description
data	String	The data to send.

receiveData

Send data from the network node to the communication service.

Table 4–10 receiveData arguments

Argument	Type	Description
fromAddres s	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

Middleware_home/**ocsg/example/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

Middleware_home/**ocsg/example/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

Middleware_home/**ocsg/example/communication_ service/example/common/src/acompany/example/plugin**

```
ExceptionType
NotificationManagerPluginFactory
SendDataPluginFactory
```

Plug-in

Middleware_home/**ocsg/example/communication_service/example/common/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.NetworkToNotificationPluginAd
apter
com.acompany.plugin.example.netex.notification.south.NetworkToNotificationPluginAd
apterImpl
com.acompany.plugin.example.netex.notificationmanager.north.NotificationManagerPlu
ginNorth
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.SendDataPluginToNetworkAdapterImp
l
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 and access tier EAR files.

Middleware_home/**ocsg/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 "Enterprise Application Deployment Descriptor Elements" in *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server* at:

http://download.oracle.com/docs/cd/E15523_01/web.1111/e13706/app_xml.htm

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.

Middleware_home/**ocsg/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 “Enterprise Application Deployment Descriptor Elements” in *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server* at:

http://download.oracle.com/docs/cd/E15523_01/web.1111/e13706/app_xml.htm

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

Middleware_home/**ocsg/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 *Services Gatekeeper System Administrator's Guide*.

Middleware_home/**ocsg/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. For details, see **InstanceFactory** in the “All Classes” section of the *Services Gatekeeper Java API Reference*.

instancemap

Middleware_home / **ocsg/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

Middleware_home/**ocsg/example/communication_service/**

build.properties

Defines the installation directory for 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.

Middleware_home/**ocsg/example/communication_service/example/common**

build.xml

Build file for the common parts of the communication service.

Middleware_home/**ocsg/example/communication_service/example/plugins/netex**

build.xml

Build file for the plug-in.

Directories for Classes, JAR, and EAR Files

Middleware_home/**ocsg/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.

Middleware_home/**ocsg/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
```

Middleware_home/**ocsg/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.

Middleware_home/**ocsg/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.

*Middleware_home***/ocsg/example/communication_ service/example/plugins/netex/dist/mbean_generationdir**

Output directory for the MBean that has been processed by the **javadoc2annotation** Apache Ant task.

Classes

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

- Communication Service Common
  - ExceptionType
  - NotificationManagerPluginFactory
- Plug-in Layer
  - ContextTranslatorImpl
  - ExamplePluginService
  - ConfigurationStoreHandler
  - ExampleMBean
  - Management
  - NotificationHandlerNorth
  - NetworkToNotificationPluginAdapter
  - NetworkToNotificationPluginAdapterImpl
  - NotificationManagerPluginNorth
  - SendDataPluginNorth
  - SendDataPluginSouth
  - SendDataPluginToNetworkAdapter
  - SendDataPluginToNetworkAdapterImpl
  - FilterImpl
  - NotificationData
  - StoreHelper

Communication Service Common

This section describes the communication service Common classes.

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

This section describes the Plug-in Layer classes.

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 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. **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 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 Example 4–1.

Example 4–1 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 Example 4–2

Example 4–2 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 **NotificationCallbackFactory**.
- The implementation class is named **NotificationCallbackImpl**
- An interface is named is named **NotificationCallback**.

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 Example 4–3, they are put in the **RequestContext** under the keys E**drConstants.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 Example 4–3, the key is C.

Example 4–3 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 Example 4–4, 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.

Example 4–4 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 fields 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

Example 4–5 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.

Example 4–5 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: *Middleware_ home***/ocsg/pte/resource/sla**.

Example 4–6 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>
```

5 Creating Extensions with Platform Development Studio Wizard

This chapter describes how to use Platform Development Studio Wizard to create extensions for Oracle Communications Services Gatekeeper.

About Platform Development Studio Wizard

Platform Development Studio Wizard is an Eclipse plug-in that streamlines the creation of the following Services Gatekeeper extensions:

- Interceptor Modules
- OAuth2 Extension Handlers
- Platform Test Environment (PTE) Custom Modules
- Web Services API Exposure Project

Note: The **REST API based on WADL (Deprecated** wizard and the **REST and/or SOAP API based on WSDL (Deprecated)** wizard are for backwards compatibility only.

See "Overview of the Platform Development Studio" for more information.

The wizard adapts itself to the type of extension you are creating. It generates the following for each extension:

- Classes and Ant build files
- A build file with Ant targets for packaging the extension for deployment

Configuring Platform Development Studio Wizard

Before you create Services Gatekeeper extensions, configure Platform Development Studio Wizard by:

- Ensuring you have the prerequisite software installed on your system.
- Configuring Platform Development Studio Wizard directories and logging levels.
- Ensuring Platform Development Studio Wizard is using the correct JRE.

Prerequisite Software

Ensure that the following software is installed on your system:

- JDK 1.7.0_15 or higher
- Eclipse 4.0 or higher
- Services Gatekeeper Release 6.1
- Services Gatekeeper Platform Development Studio

Note: Platform Development Studio is automatically installed in your system's *Middleware_home*/**ocsg_pds** directory when you run the Services Gatekeeper multi-tier installer and select to install the Administration Server component.

Configuring Platform Development Studio Wizard Directories and Logging Levels

To configure Platform Development Studio Wizard directories and logging levels:

1. Copy the **oracle.ocsg.eclipse.jar** file from your *Middleware_home*/**ocsg_pds** directory to your *Eclipse_Home*/**plugins** directory.
2. Start Eclipse.
3. From the **Window** menu, select **Preferences**.

   The Preferences window appears.
4. From the navigation tree, select **Services Gatekeeper Platform Development Studio**.

   The Services Gatekeeper Platform Development Studio window appears.
5. In the **Middleware Home Directory** field, enter or browse to the *Middleware_home* directory. For example, **Oracle/Middleware**.

   The *Middleware_home* directory serves as the repository for common files that are used by Oracle Communications products installed on the same machine, such as Services Gatekeeper and WebLogic Server.
6. In the **JDK Installation Directory** field, enter or browse to the directory where the JDK is installed. For example, **/java/jdk1.7.0_15**.
7. In the Logging Level group, do one of the following:
   - To create detailed log files, select **All**.
   - To create less detailed log files, select **Standard**.
8. Click **OK**.

Ensuring Platform Development Studio Wizard Uses JRE 1.7

To ensure that Platform Development Studio Wizard is using JRE 1.7 or higher:

1. In Eclipse, from the **Window** menu, select **Preferences**.

   The Preferences window appears.
2. From the navigation tree, expand **Java** and then select **Installed JREs**.

   The Installed JREs window appears.
3. Ensure that the **Installed JREs** table lists JRE 1.7.0_15 or higher.
4. Click **OK**.

Generating an Interceptor Module

To generate an interceptor module project:

1. Open Eclipse.
2. From the **File** menu, select **New** and then select **Other** to start the New wizard.

   The Select a wizard window appears.
3. From the navigation tree, expand **Services Gatekeeper Platform Development Studio** and then select **Interceptor Module**. Click **Next**.

   The Generate Interceptors module window appears.
4. Enter a unique project name and choose a location for your project.
5. In the Interceptor Settings group, do the following:
   a. In the **Package Name** field, enter the name of the interceptor module package.
   b. To include a life cycle listener, enter the name in the **Application Lifecycle Listener** field.
   c. Click the Add icon.

      The Add Interceptor window appears.
   d. In the **Name** field, enter the name of the interceptor.
   e. In the **Index** field, enter the name of interceptor index.
   f. In the **Point** list, select the location in Services Gatekeeper where the interceptor will intercept events: MO North, MO South, MT North, or MT South.
   g. Click **OK**.
   h. To create more interceptors, click the Add icon and fill out the information in the Add Interceptor window.
6. Click **Finish**.

Generating an OAuth 2.0 Extension Handler

To generate an OAuth 2.0 extension handler project:

1. Open Eclipse.
2. From the **File** menu, select **New** and then select **Other** to start the New wizard.

   The Select a wizard window appears.
3. From the navigation tree, expand **Services Gatekeeper Platform Development Studio** and then select **OAuth2 Extension Handlers**. Click **Next**.

   The Generate OAuth2 Extended Handlers window appears.
4. Enter a unique project name and choose a location for your project.
5. In the OAuth2 Extension Settings group, do the following:
   a. In the **Package Name** field, enter the name of the extension handler.
   b. Click the Add icon.

The Add Handler window appears.

c. In the **Handler Name** field, enter the name of the OAuth2 handler.

d. To add more extensive customization for the OAuth2 handler, select the **Ext** check box.

e. In the **Validator Name** field, enter the name of the validator for the OAuth2 handler.

f. In the **Response Type** list, select the type of response the validator expects: code or token.

g. In the **Grant Type** list, select the grant type that the validator expects: authorization code, refresh token, password, or client credentials.

h. Click **OK**.

i. To create more handlers, click the Add icon and fill out the information in the Add Handler window.

6. Click **Finish**.

Generating a Platform Test Environment Custom Module

To generate a Platform Test Environment custom module:

1. Open Eclipse.
2. From the **File** menu, select **New** and then select **Other** to start the New wizard.

   The Select a wizard window appears.
3. From the navigation tree, expand **Services Gatekeeper Platform Development Studio** and then select **PTE Custom Module**. Click **Next**.

   The Generate PTE modules window appears.
4. Enter a unique project name and choose a location for your project.
5. To define the module by using a custom WSDL file, select the **Use custom WSDL files** option.

   The Configure Service WSDL Files area and Configure Callback WSDL Files area are visible.

   a. In the Configure Service WSDL Files area, click the Add icon.

   The WSDL Files Configuration window appears.

   b. In the **WSDL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.

   c. In the JAX-WS or JAXB binding files area, click the Add icon.

   The Binding File Configuration window appears.

   d. In the **Binding File** field, either enter the path to the binding file or click the Add icon to browse to the binding file’s location. Click **OK**.

   e. Click **OK**.

   f. In the Configure Callback WSDL Files area, click the Add icon.

The WSDL Files Configuration window appears.

g. In the **WSDL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.

h. In the JAX-WS or JAXB binding files area, click the Add icon.

   The Binding File Configuration window appears.

i. In the **Binding File** field, either enter the path to the binding file or click the Add icon to browse to the binding file’s location. Click **OK**.

j. Click **OK**.

6. To define the module by using a WADL file, select the **Use predefined WSDL files** option.

   In the drop-down list, select one of the following predefined communication services:

   - px30_audio_call
   - px21_call_notification
   - px30_call_notification
   - px21_multimedia_messaging
   - px21_presence
   - ews_push_message
   - px21_sms
   - ews_binary_sms
   - ews_subscriber_profile
   - px21_terminal_location
   - px21_third_party_call
   - px30_third_party_call
   - px30_payment

7. To define the module by using a WADL file, select the **Use WADL** option.

   The Configure Service WADL Files area appears.

   a. Click the Add icon.

      The WADL Files Configuration window appears.

   b. In the **WADL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.

   c. Click **OK**.

8. To define the module by using a REST-to-REST WADL file, select the **Rest2Rest WADL** option.

   The Configure Service WADL Files area appears.

   a. Click the Add icon.

      The WADL Files Configuration window appears.

   b. In the **WADL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.

   c. Click **OK**.

9. From the Settings Area, in the **Name** field, enter a name for the PTE module.
10. In the **Package Name** field, enter the name of the Java package that will contain the package classes.
11. In the **Company** field, enter the name of your company.
12. In the **Version** field, enter the version number of the PTE module.
13. Click **Finish**.

Generating a Web Service API Exposure Project

To generate a web service API exposure project:

1. Open Eclipse.
2. From the **File** menu, select **New** and then select **Project** to start the New wizard.

   The Select a wizard window appears.
3. From the navigation tree, expand **Services Gatekeeper Platform Development Studio** and then select **Web Service API Exposure Project**. Click **Next**.

   The Create a Communication Service window appears.
4. Enter a unique project name and choose a location for your project.
5. Click **Next**.

   The Define the Communication Service window appears.
6. In the Configure Service WSDL or WADL Files area, click the Add icon.

   The WSDL/WADL Files Configuration window appears.
7. In the **WSDL/WADL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.
8. Click **OK**.
9. In the Configure Callback WSDL or WADL Files area, click the Add icon.

   The WSDL/WADL Files Configuration window appears.
10. In the **WSDL/WADL File** field, either enter the path to the service file or click the Add icon to browse to the service file's location.
11. Click **OK**.
12. In the **Company** field, enter the name of your company.
13. In the **Version** field, enter the version number of the communication service.
14. In the **Identifier** field, enter an identifier to tie together a collection of Web services. The identifier is used in the names of the generated WAR and JAR files and the service type for the communication service. For example:

    ```
    sms_Identifier.war
    sms_Identifier_callback.jar
    ```
15. In the **Service Type** field, enter the service type. The service type is used in EDRs and statistics. For example:

    ```
    smsServiceType
    MultimediaMessagingServiceType
    ```

16. In the **Java Class Package Name** field, set the package names to use. For example: **com.mycompany.service**.
17. In the **Web Services Context Path** field, set the context path for the Web service. For example: **myService**.
18. In the **API Exposure Interface** list, select **REST** to generate a RESTful Service Facade for the communication service, or **SOAP** to generate SOAP Service Facade for the communication service.
19. In the **Content Type** list, select the HTTP content type that Services Gatekeeper uses when it sends a request to the network: **application/xml** or **application/json**.
20. In the **Callback Content Type** list, select the HTTP content type that Services Gatekeeper uses when it sends a request to the application: **application/xml** or **application/json**.
21. Click **Finish**.

Adding and Removing Extension Plug-ins

This section describes how to add or remove plug-ins from an extension.

Adding a Plug-in to a Services Gatekeeper Project

To add a plug-in to an existing Services Gatekeeper project:

1. Open Eclipse.
2. From the **Window** menu, click **Show View** and then click **Package Explorer**.

   The Package Explorer window appears.
3. Right-click your Services Gatekeeper project and then choose **Properties**.

   The Properties for *ProjectName* appears.
4. From the navigation tree, select **Plugins Configuration**.

   The Plugins Configuration window appears.
5. Click the Add Plugin icon.

   The Add Plugin window appears.
6. In the **Protocol** field, enter an identifier for the network protocol that the plug-in implements. It is used as a part of the names of the generated JAR file and the service name. For example:

   ```
   communicationServiceIdentifier._protocol.jar
   Plugin_communicationServiceIdentifier._protocol
   ```
7. In the **Schemes** field, enter the address schemes that the plug-in can handle. Use a comma-separated list if multiple schemes are supported. For example: tel or sip.
8. In the **Package Name** field, enter the package names to be used.
9. In the **Company** field, enter the name of your company.
10. In the **Version** field, enter the version number of the plug-in.

11. In the **Type** area, select SOAP, SOAP to SOAP, or SIP.
12. Click **OK**.
13. Click **Apply** and then **OK**.

Removing a Plug-in from a Communication Service

To remove a a plug-in from an existing Services Gatekeeper project:

1. Open Eclipse.
2. From the **Window** menu, click **Show View** and then click **Package Explorer**.

   The Package Explorer window appears.
3. Right-click your Services Gatekeeper project and then choose **Properties**.

   The Properties for *ProjectName* appears.
4. From the navigation tree, select **Plugins Configuration**.

   The Plugins Configuration window appears and a list of plug-ins defined for the project is displayed.
5. Select the plug-in to remove.
6. Click the Remove Plugin icon.

   The plug-in definitions are removed from the list.
7. Click **Apply** to remove the plugin from the project.

   Warning: This removes all parts of the project, including any manually edited or added files.

6 Understanding the Communication Service Project Output

The chapter describes a project generated from the Oracle Communications Services Gatekeeper Platform Development Studio Wizard.

About the Generated Communication Service

The section describes a project generated from the Platform Development Studio Wizard.

About the Communication Service Project

Generating a communication service project creates the directory structure illustrated in Example 6–1.

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

- **build.properties**: properties file for the build files:
  - **wlng.home** is set to *Middleware_home*, the base directory for the installation.
  - **pds.home** is set to *Middleware_home*/**ocsg_pds**, the base directory for the Platform Development Studio.
  - **wls.home** is set to *Middleware_home*/**wlserver**, 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, Apache Ant task and targets used by all build files in the project.

The directories and files in bold in Example 6–1 are generated when building the common parts of the communication service; the others are generated by the PDS (Eclipse) Wizard.

Example 6–1 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.

About the RESTFul Service Facade

You can generate a RESTful Service Facade using the PDS wizard. The sections below describe the default generation of the RESTful Service Facade and how to modify it.

Default RESTful Service Facade

When a RESTful Service Facade is generated, the following files are generated in addition to the directory structure described in Example 6–1:

- **rest_***identifier***.war** in the **common/dist** directory
- **rest/***identifier***/index.html**, in the **common/dist/tmp/wars/rest**_identifier directory
- **rest-config.xml**, in the identifier**/common/resources/facade/rest** directory

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 **common/dist/tmp/wars/rest_***identifier* directory. 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 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 PDS 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 6–1 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 6–1. This information is provided in the generated API documentation.

Table 6–1 Example of a RESTful resource as used by an application

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

Table 6–1 (Cont.) Example of a RESTful resource as used by an application

Attribute	Value
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 1.0 is used to deliver network-triggered messages or notifications to an application.

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, see the discussion on using the HTTP publish-subscribe server in *Oracle Fusion Middleware Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server* at:

`http://download.oracle.com/docs/cd/E15523_01/web.1111/e13712/pubsub.htm`

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/** 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 Services Gatekeeper installation.
- *identifier* is specified in the field identifier in the PDS wizard.

Notifications are sent by using Bayeux Deliver Event messages.

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, you should update rest-config.xml with a `<response-header>` element. See "Customize the RESTful Service Facade" for more information.

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 *Middleware_home*/**ocsg/applications/rest.jar**. Table 6–2 contains a description of the mappings.

Table 6–2 Structure and Description of rest-config.xml

Element/Type	Description
<resources>	Main element. 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).

Table 6–2 (Cont.) Structure and Description of rest-config.xml

Element/Type	Description
<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>/<pathinfo>?<name1>=<value1>&<name2>=<value2> The attribute uri corresponds to <servlet-path> in the URI.
<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 6–2 (Cont.) 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 using 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>/<pathinfo>?<name1>=<value1> invokes the given operation. If the URI used by an application is: http://host:port/<context-root/<servlet-path>/<pathinfo>?<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.
<path-info-param>	Parent element: <operation>. Has the attribute: ▪ name Defines 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>/<pathinfo>?<name1>=<value1>&<name2>=<value2> The attribute name corresponds to <pathinfo> in the URI.

Table 6–2 (Cont.) Structure and Description of rest-config.xml

Element/Type	Description
<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 PDS 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.
<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 RESTFul 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 6–2 (Cont.) 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 6–2 (Cont.) 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.SendDat a" name="binaryField"/> </data-type-adapter>
<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.

Table 6–2 (Cont.) Structure and Description of rest-config.xml

Element/Type	Description
<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"/>
<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 6–2 (Cont.) 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&name2=value2
```

The following applies:

- *servlet-path* must match the attribute URI of the `<resource>` element.
- *pathinfo* must match the attribute name of the `<path-info-param>` element. 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 `<request-param>` element.
  - *value* must match the attribute value of the `<request-param>` element.

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 PDS wizard and the interface name derived from the WSDL:

```
package name from wizard.service name from wizard.rest.interface name from
WSDLRestImpl
```

The method name is derived from the WSDL. The resource URI is derived from the namespace definition in the WSDL. The `<httpMethod>` element 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.NotifySmsReception"
      id="notifySmsReception"/>
    <data class=
       "org.csapi.schema.parlayx.sms.notification.v2_
2.local.NotifySmsDeliveryReceipt"
       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 **handleRequest** method is invoked before a request is passed on to the Service Enabler.

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

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

About the Communication Service Plug-in

When you create a plug-in for a communication service, the directory structure illustrated in Example 6–2 is created under the top-level directory. The base directory depends on the type of communication service the plug-in belongs to, 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 existing directories; for example a Parlay X 30 Audio Call plug-in the **px30_audio_call** directory, a Parlay X 2.1 Short Messaging in the **px21_sms**, and so on.

If the plug-in is for a new communication service, the base directory is given in the Identifier entry field in the PDS Wizard.

The base directory contains the directory plugins, which contains subdirectories for each protocol being added. The names of the directories are the same as the name chosen for the Protocol field in the PDS 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 instance map that is 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 Example 6–2 are generated when building the plug-in, the others are generated by the PDS Wizard.

Example 6–2 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
```

About the SOAP2SOAP Plug-in

When you create a SOAP2SOAP plug-in, the directory structure described in "About the Communication Service Plug-in" is created under the top-level directory. In addition, the directories and files in "Generated project for a SOAP2SOAP plug-in" are generated. The directories and files in bold are created when building the plug-in; the others are generated by the PDS Wizard.

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

Example 6–3 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 Example 6–3, 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 file 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.

In the start script, the **-Dweblogic.wsee.soap.81CustomException** flag must be set to `true` in order to push the soap faults defined in WSDL as-is.

Generated Artifacts for a SOAPSOAP Communication Service

When a SOAP2SOAP communication service is generated, the following directory structure is created in the directory *domain_home*/**soap2soap** on the administration server.

Example 6–4 Directory Structure from Generating SOAP2SOAP Communication Services

```
+- <Communication service name>_<version>/
+- build.properties
+- build.xml
+- common.xml
+- <Communication service name>/
+- common/+- resources/
+- enabler/
+- facade/
+- handlerconfig.xml
+- dist/
+- com.bea.wlcp.wlng.soap2soap.<service type>.store_<version>.jar
+- wlng_at_<Communication service name>.ear
+- wlng_nt_<Communication service name>.ear
+- plugins/
|+- soap/
 +- tmp/
```

All Java source files, build files, configuration files, and deployable artifacts are created under the directory *domain_home*/**soap2soap**. The source files are there mainly for reference. The only files that are necessary to deploy the communication service are:

- **wlng_at_communication_service_name.ear**
- **wlng_nt_communication_service_name.ear**

wlng_at_communication_service_name.ear should be deployed in the access tier server.

wlng_nt_communication_service_name.ear should be deployed in the network tier server.

The generated communication service can be deployed as a part of the generation process, as described above, or using standard WebLogic Server tools.

See the discussion on deploying and undeploying communication services and plug-ins in *Services Gatekeeper System Administrator's Guide* for information on how to deploy the files using WebLogic Server tools.

Properties for SOAP2SOAP Plug-ins

Table 6–3 lists the SOAP2SOAP plug-in properties.

Table 6–3 SOAP2SOAP Plug-in Properties

Property	Description
Managed object in Administration Console	*domain name*, then OCSG, then *server name*, then Communication Services, then *plug-in instance ID*

Table 6–3 (Cont.) SOAP2SOAP Plug-in Properties

Property	**Description**

MBean	Domain=com.bea.wlcp.wlng
	Name=wlngt
	InstanceName is same as plug-in instance ID
	Type=com.bea.wlcp.wlng.httpproxy.management.HTTPProxyMBean

Network protocol plug-in service ID	Defined when generating the SOAP2SOAP plug-in using the Platform Development Studio. When generated fm the Administration console, the ID is *Name*_ soap_plugin.
Network protocol plug-in instance ID	The ID assigned when the plug-in instance is created: see the discussion on configuring and managing the plug-in manager in *Services Gatekeeper System Administrator's Guide* for more information.
Supported Address Scheme	Defined when generating the SOAP2SOAP plug-in using the Platform Development Studio. When generated fm the Administration console, the address scheme is always an empty string.

Application-facing interfaces	Derived from the package name of the network protocol plug-in, assigned when the plug-in was generated, and the name of the interface as defined in the WSDL.
	For application-initiated request:
	`package name.plugin.interface namePlugin`
	For network-triggered requests:
	`package name.callback.interface nameCallback`
	See see the discussion on configuring and managing the plug-in manager in *Services Gatekeeper System Administrator's Guide* information on how to list the interfaces.

Service type	Given when the plug-in was generated using the Platform Development Studio or the administration console.
Exposes to the service communication layer a Java representation of:	Depends on the WSDLs used.
Interfaces with the network nodes using:	The same protocol as exposed by the application-facing interfaces.
Deployment artifacts	*communication service identifier_protocol*_plugin.jar, *communication service identifier*_service.jar and *communication service identifier*_ callback.war, packaged in wlng_nt_*communication service*.ear *communication service identifier*_callback.jar and *communication service identifier*.war, packaged in wlng_at_*communication service identifier*.ear *communication service identifier* was given in the PDS wizard or the Administration console when the communication service was generated. *protocol* is configurable when using the PDS wizard. If the communication service was generated using the Administration console, it is always `soap`.

About the SIP Plug-in

When creating a SIP plug-in, the directory structure described in "About the Communication Service Plug-in" is created under the top-level directory. In addition, the directories and files in Example 6–5 are generated. The directories and files in bold are created when building the plug-in; the others are generated by the PDS Wizard.

Example 6–5 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 Example 6–3, a set of additional classes and configuration files for the SIP type plug-in is generated compared to the standard plug-in.

Table 6–4 contains a summary of the added items.

Table 6–4 Additional files generated for a SIP plug-in

File	Description
sip.xml	SIP Application deployment descriptor. See: WebLogic 11g - “Developing and Programming SIP Applications” in *Oracle WebLogic Communication Services Developer's Guide* at the WebLogic 11g documentation web site: http://docs.oracle.com/cd/E36909_01/doc.1111/e13807/partpage_ii.htm#sthref41
web.xml	HTTP Servlet deployment descriptor. See “Understanding WebLogic Server Deployment” in *Oracle WebLogic Server Developing Applications for WebLogic Server*.
*identifier_protocol_*sip.war	Deployable SIP application.

Table 6–4 (Cont.) Additional files generated for a SIP plug-in

File	Description
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.

About the Diameter Plug-in

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

Converged Application Server includes an implementation of the base Diameter protocol that supports the core functionality and accounting features described in RFC 3588. 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.

Services Gatekeeper uses the Diameter support provided by 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 Converged Application Server, see the discussion on overview of the diameter API in:

Oracle WebLogic Communication Services Developer's Guide at:

`http://download.oracle.com/docs/cd/E15523_01/doc.1111/e13807/diameter_1.htm`.

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

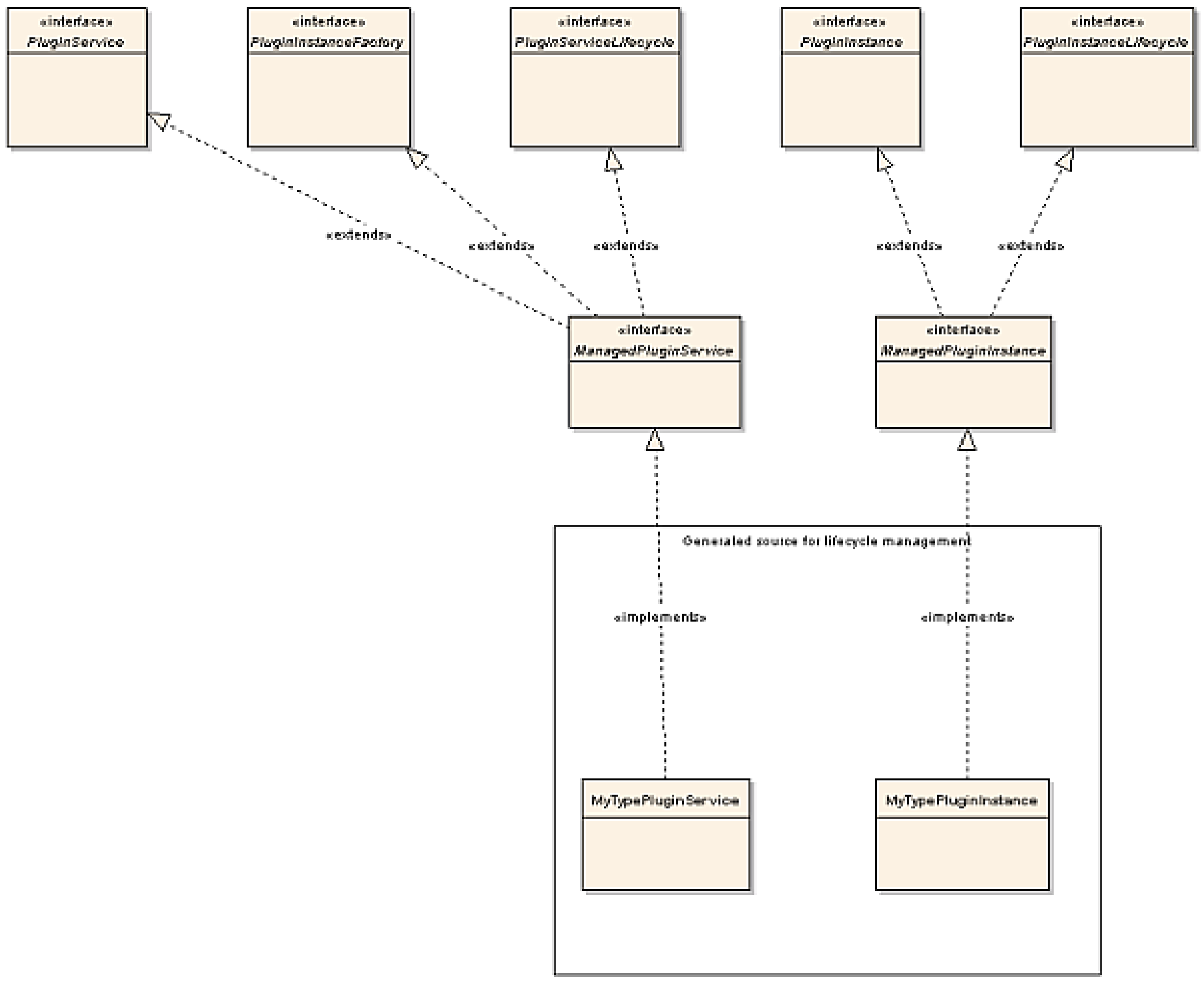
The Diameter API is packaged in *Middleware_home*/**wlserver/sip/server/lib/wlssdiameter.jar**.

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

Figure 6–1 illustrates the generated plug-in classes for life-cycle management and their relationships with other interfaces.

Figure 6–1 Generated Plug-in Classes


«interface» PluginService
«interface» PluginInstanceFactory
«interface» PluginServiceLifecycle
«interface» PluginInstance
«interface» PluginInstanceLifecycle
«extends»
«extends»
«extends»
«extends»
«extends»
«interface» ManagedPluginService
«interface» ManagedPluginInstance
Generated source for lifecycle management
«implements»
«implements»
MyTypePluginService
MyTypePluginInstance

Interface: ManagedPluginService

The interface a plug-in service needs to implement.

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 "Understanding the Plug-in States".

PluginService

Class

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

Defines the life-cycle for a plug-in service, see "Understanding the Plug-in States" for more information.

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 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 "Understanding the Plug-in States" for more information.

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 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

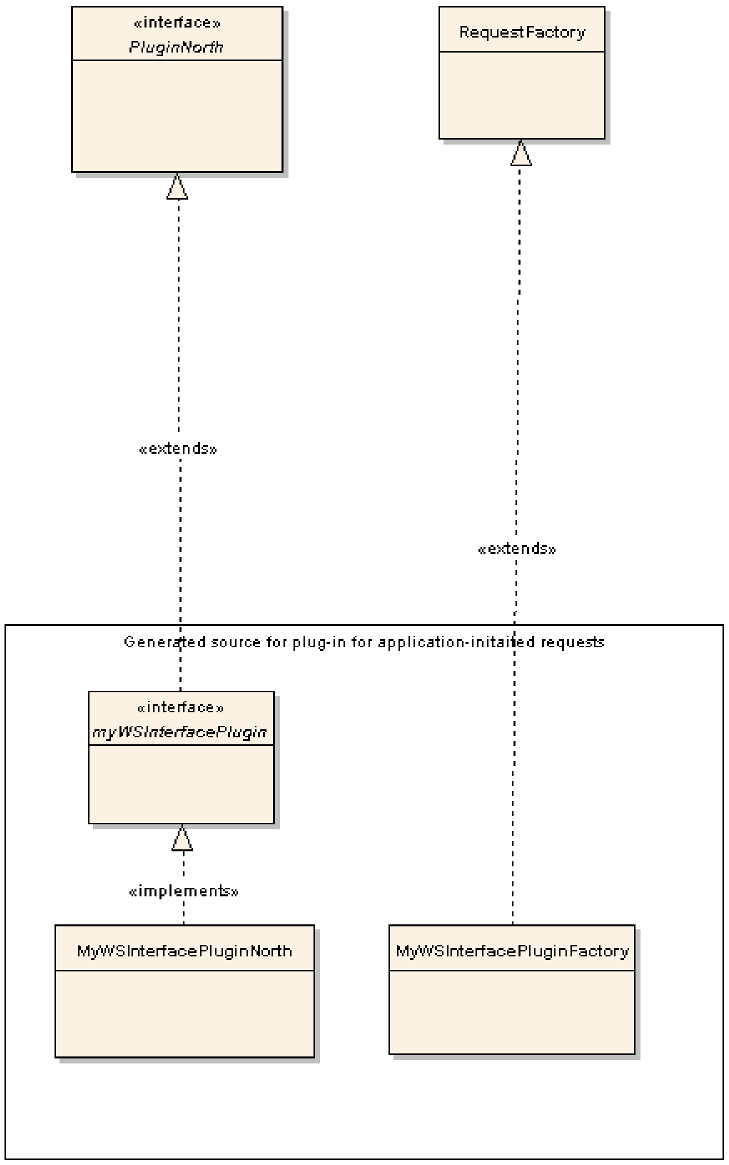
This is an empty implementation of the Plug-in North interface. This interface is generated based on the WSDL files that define the application-facing interface. This is the starting point for the plug-in implementation.

The following files are 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 6–2 shows the PluginNorth and RequestFactory class diagram.

Figure 6–2 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 are 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 *Services Gatekeeper 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 InterfaceService_Impl
- Web Services InterfaceService

*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 **HTTPProxyMBean**. For details see the “All Classes” section of the *Services Gatekeeper Java API Reference*.

HeartbeatManagement is described in **HearbeatMBean**. For details see the “All Classes” section of the *Services Gatekeeper Java API Reference*.

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: RequestInfo".
- 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.

HTTPProxyMBean Reference

Set field values (attributes) and use methods (operations) for the **HTTPProxyMBean** from the Administration Console or a Java application. For information on the methods and fields, see the “All Classes” section of *Services Gatekeeper OAM Java API Reference*.

Adding a SOAP2SOAP Communication Services

This section explains how to generate, deploy and configure Services Gatekeeper SOAP2SOAP communication services.

About SOAP2SOAP Communication Services

You can create SOAP2SOAP communication Services using the Partner and API Management Portal GUI.

Based on a set of service WSDLs and callback WSDLs, a SOAP2SOAP communication service acts as a proxy service and provides the functionality provided by Services Gatekeeper container services, such as:

- SLA enforcement

- EDR, CDR, and alarms
- Authentication, access control, and accounting

SOAP2SOAP plug-in services can be instantiated using the plug-in manager. The plug-in instances process traffic and connect to individual network nodes. The instances are managed independently of each other.

For application-initiated requests, all requests are routed to the network node defined for the plug-in instance.

For network-triggered requests, the network node should distinguish the application instance the request is targeted to by adding the application instance ID to the URL:

```
http://IP_Address_of_AT_server:port/context-root_nt/plug-in instance ID_
nt/application_instance_ID
```

If the number of HTTP methods that time-out during a certain time-period exceeds the maximum allowed, the plug-in instance is taken out of operation for a configurable time-period. The HTTP time-out behavior is valid for request between the plug-in instance and the network node. The purpose of this feature is to prevent network nodes that are malfunctioning from incoming requests.

Generated Artifacts for a SOAP2SOAP Communication Service

When a SOAP2SOAP communication service is generated, the following directory structure is created in the directory *domain_Home*/**soap2soap** on the administration server.

Example 6–6 Directory Structure from Generating SOAP2SOAP Communication Services

```
+- <Communication service name>_<version>/
+- build.properties
+- build.xml
+- common.xml
+- <Communication service name>/
+- common/+- resources/
+- enabler/
+- facade/
+- handlerconfig.xml
+- dist/
+- com.bea.wlcp.wlng.soap2soap.<service type>.store_<version>.jar
+- wlng_at_<Communication service name>.ear
+- wlng_nt_<Communication service name>.ear
+- plugins/
|+- soap/
 +- tmp/
```

All Java source files, build files, configuration files, and deployable artifacts are created under the directory *domain_Home*/**soap2soap**. The source files are there mainly for reference. The only files that are necessary to deploy the communication service are:

- **wlng_at_communication_service_name.ear**
- **wlng_nt_communication_service_name.ear**

wlng_at_communication_service_name.ear should be deployed in the access tier server.

wlng_nt_communication_service_name.ear should be deployed in the network tier server.

The generated communication service can be deployed as a part of the generation process, as described above, or using standard WebLogic Server tools.

See the discussion on managing APIs for partner applications in *Services Gatekeeper Partner and API Management Portal Online Help* for information on how to create and deploy a SOAP2SOAP communication service.

Managing and Configuring SOAP2SOAP Communication Services

This section describes configuration attributes and operations available for SOAP2SOAP-type of plug-in instances.

Table 6–5 Task Overview

To see a	Refer to
Detailed list of necessary for managing and configuring a plug-in instance	Properties for SOAP2SOAP Plug-ins
List of operations and attributes related to management and provisioning	Provisioning Workflow for SOAP2SOAP Communication Services
Reference of management attributes and operations	HTTPProxyMBean Reference

Properties for SOAP2SOAP Plug-ins

Table 6–6 Properties

Property	Description
Managed object in Administration Console	*domain name*, then OCSG, then *server nam*e, then Communication Services, then *plug-in instance ID*

MBean	Domain=com.bea.wlcp.wlng
	Name=wlngt
	InstanceName is same as plug-in instance ID
	Type=com.bea.wlcp.wlng.httpproxy.management.HTTPProxyMBe an

Network protocol plug-in service ID	Defined when generating the SOAP2SOAP plug-in using the Platform Development Studio. When generated fm the Administration console, the ID is *Name_* soap_plugin.
Network protocol plug-in instance ID	The ID assigned when the plug-in instance is created: see the discussion on configuring and managing the plug-in manager in *Services Gatekeeper System Administrator's Guide* for more information.
Supported Address Scheme	Defined when generating the SOAP2SOAP plug-in using the Platform Development Studio. When generated fm the Administration console, the address scheme is always an empty string.

Table 6–6 (Cont.) Properties

Property	**Description**

Application-facing interfaces	Derived from the package name of the network protocol plug-in, assigned when the plug-in was generated, and the name of the interface as defined in the WSDL.
	For application-initiated request:
	`package name.plugin.interface namePlugin`
	For network-triggered requests:
	`package name.callback.interface nameCallback`
	See see "Configuring and Managing the Plug-in Manager" in *Services Gatekeeper System Administrator's Guide* information on how to list the interfaces.

Service type	Given when the plug-in was generated using the Platform Development Studio or the administration console.
Exposes to the service communication layer a Java representation of:	Depends on the WSDLs used.
Interfaces with the network nodes using:	The same protocol as exposed by the application-facing interfaces.
Deployment artifacts	*communication service identifier_protocol*_plugin.jar, *communication service identifier*_service.jar and *communication service identifier*_ callback.war, packaged in wlng_nt_*communication service*.ear *communication service identifier*_callback.jar and *communication service identifier*.war, packaged in wlng_at_*communication service identifier*.ear *communication service identifier* was given in the PDS wizard or the Administration console when the communication service was generated. *protoco*l is configurable when using the Platform Development Studio wizard. If the communication service was generated using the Administration console, it is always `soap`.

Provisioning Workflow for SOAP2SOAP Communication Services

For each application that uses SOAP2SOAP communication services and supports network-triggered requests, set up a mapping between the application instance ID and the URL for the Web service that the application instance implements. Use these **HTTPProxyMBean** methods to manage the callback URLs for the application instances:

- **addApplicationEndPoint**
- **getApplicationEndPoint**
- **listApplicationEndPoints**
- **removeApplicationEndPoint**

Build Files and Targets for a Communication Service Project

This section describes the build files, including the targets and associated Ant tasks, 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 communication Service.
- **stage**, copies the JARs for the plug-ins to the directory stage.
- **make-facade**, creates a deployable EAR file for the access tier in the directory dist.
- **make-enabler**, creates a deployable EAR file for the network tier in the directory dist.
- **deploy-facade**, deploys the service facade EAR file to the access tier.
- **undeploy-facade**, undeploys the service facade EAR file from the access tier.
- **deploy-enable**r, deploys the service enabler EAR file from the network tier.
- **undeploy-enabler**, undeploys the service enabler EAR file 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 "Securing SOAP-Based Web Services" in *Services Gatekeeper 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 *Middleware_home*/**ocsg_pds/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 6–7 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 *Middleware_home*, this depends on the installation. Example: **c:\bea\ocsg**.
pdsHome	Path to *Middleware_home*/**ocsg_pds**.
classpath	Defines the necessary classpaths. Must include: *Middleware_home*/**ocsg**/server/lib/weblogic.jar *Middleware_home*/**ocsg/server/lib/webservices.jar** *Middleware_home*/**ocsg**/server/lib/api.jar *Middleware_home*/**ocsg_pds**/lib/wlng/wlng.jar *Middleware_home*/**ocsg_pds/lib/log4j/log4j.jar**
servicewsdl	URL to the WSDL that defines the service.

Example:

```
<cs_gen destDir="${dist.dir}"
   packageName="com.bea.wlcp.wlng.example"
   name="say_hello"
   serviceType="example"
   company="BEA"
   version="6.0"
   contextPath="sayHello"
   soapAttachmentSupport="false"
   wlngHome="${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 6–8 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 file.
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.
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="${dist.dir}"
   packageName="com.bea.wlcp.wlng.example.bla"
   name="say_hello"
   serviceType="example"
   esPackageName="com.bea.wlcp.wlng.example"
   protocol="bla"
   schemes=""
   company="BEA"
   version="6.0"
   pluginifjar="${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 6–9 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 file 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 file
fileset	Should point to the communication service JAR file.
zipfileset	Should point to the plug-in JAR file(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.

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

Table 6–10 javadoc2annotation Ant Macro

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

7 Service Enabler Example with SIP plug-in

This section describes the example network protocol plug-in for SIP connectivity provided in Oracle Communications Services Gatekeeper Platform Development Studio.

Overview of the Service Enabler Example with SIP Plug-in

Services Gatekeeper includes an example SIP Plug-in the 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.

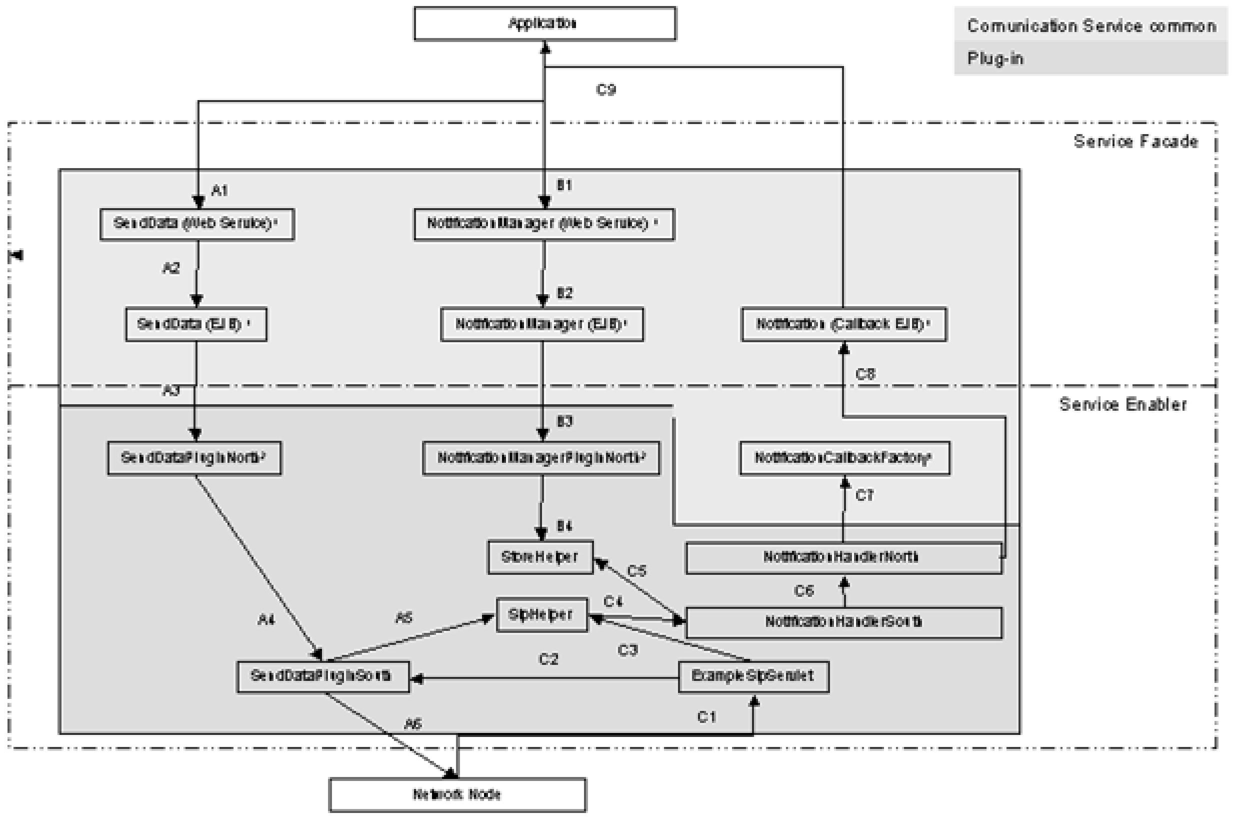
The communication service to network node interface is a simple SIP based interface that defines the two commands:

- **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 7–1 illustrates the flow for these operations.

Figure 7–1 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.

Understanding the SIP Example Interfaces

The example 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 Communications 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.

SIP Example Directory Structure

The directory structure is similar to the directory structure for the example communication service, see "SIP Example Directory Structure" in "Communication Service Example" but adds these classes, descriptors, and artifacts:

```
| +- 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". This section lists the classes that have been 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.NotificationHandlerSouth**. 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.ja**r
- **example_sip_plugin.ja**r
- **example_sip.war**

SIP Example 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 *Middleware_home*/**ocsg_pds/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 *Middleware_home*/**ocsg_pds**/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 *Middleware_home*/**ocsg_pds/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 *Middleware_home*/**ocsg_pds/example/communication_service/example/plugins/sip/config/edr**.

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

SIP Example Classes

This section describes the classes that are different from those in the netex plug-in described in "Communication Service Example".

ExampleServlet

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

Extends **javax.servlet.sip.SipServlet**

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 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

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.

SIP Example SLA

The SLA is on the communication service level and identical to the one for the example communication service. See "Communication Service Example" for details.

8 Using the SMPP API

This chapter provides an overview of the Oracle Communications Services Gatekeeper Short Messaging Peer to Peer Protocol (SMPP) API Java interface. It also contains some guidance on how to develop a custom SMPP plug-in using the Services Gatekeeper Platform Development Studio and the SMPP APIs.

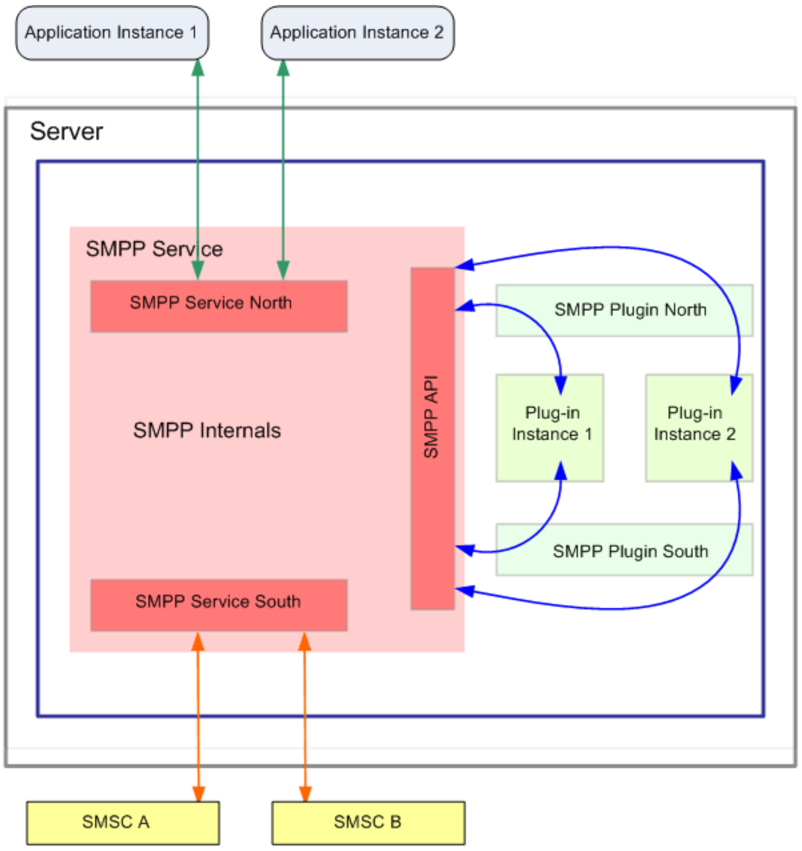
Understanding the SMPP API

The Services Gatekeeper SMPP implementation depends on a core module, the SMPP Service, which provides connectivity services for SMPP plug-ins. The SMPP API defines the interfaces between the plug-ins and the SMPP Service.

Using this API, platform developers can create SMPP plug-ins without having to manage the low-level tasks of connecting from Services Gatekeeper to applications and to SMSCs.

Figure 8–1 illustrates the basic Services Gatekeeper SMPP architecture.

Figure 8–1 SMPP Architecture in Services Gatekeeper



SMPP Service Interfaces

The SMPP Service performs connection services on behalf of the standard SMPP plug-ins (Native SMPP and ParlayX 2.1 SMPP) as well as any custom SMPP plug-ins.

The SMPP Service handles the following tasks:

- Receives SMPP data from the socket.
- Constructs the SMPP protocol data unit (PDU).
- Associates the current PDU with the correct application instance.
- Invokes the plug-in.
- Manages connections between Services Gatekeeper and applications.
- Manages connections between Services Gatekeeper and Short Message Service Centers (SMSCs).

See the **oracle.ocsg.protocol.smpp.service** package in the *Services Gatekeeper Java API Reference* for documentation of the SMPP Services interfaces.

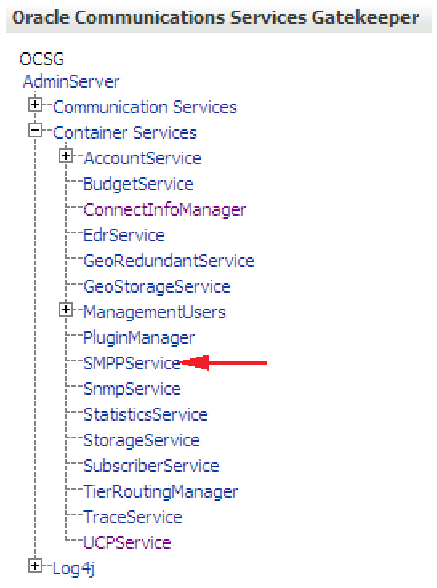
The SMPPServiceNorth interface processes requests received from an application-facing plug-in and sends them to the application.

The SMPPServiceSouth interface processes requests received from a network-facing plug-in and sends them to the SMSC.

The SMPP Service is a standard Services Gatekeeper WebLogic Server (WLS) service. You can access its Operations, Administration, and Maintenance (OAM) (operations and management) functions from the Administration console as **SMPPService** under **es**.

Figure 8–2 shows the SMPP service menu in the Administration Console.

Figure 8–2 SMPP Service in the Administration Console



SMPPPluginSouth

The SMPPluginSouth interface processes network-triggered operations received from SMPPServiceSouth and sends them to SMPPServiceNorth. You would extend and implement this interface to add a new network-facing SMPP protocol.

SMPPPluginNorth

The SMPPluginNorth interface processes requests received from SMPPServiceNorth and sends them to SMPPServiceSouth. You would extend and implement this interface to add a new application-facing SMPP protocol.

Additional Information You Need to Build an SMPP Plug-in

In addition to the information in this chapter, developers should consult the following documents for information on how to build an SMPP plug-in:

- *Services Gatekeeper Actions Java API Reference*

  Of special interest are the following packages, which include the interfaces and classes for the SMPP service and plug-ins:

  – oracle.ocsg.protocol.smpp.service

  – oracle.ocsg.protocol.smpp.plugin

  – oracle.ocsg.protocol.smpp.event

  – oracle.ocsg.protocol.smpp.common

  – oracle.ocsg.protocol.common

You also need resources from various generic packages such as:

- com.bea.wlcp.wlng.api.edr
- com.bea.wlcp.wlng.api.management
- com.bea.wlcp.wlng.api.plugin

- *Services Gatekeeper Communication Service Reference Guide*

  See the Native SMPP chapter for an overview of the Services Gatekeeper Native SMPP communication service, which uses the SMPP Service. This chapter includes general information about how the SMPP Service handles connectivity and documents the configurable attributes and operations of the SMPP Service.

Creating a Custom SMPP Plug-in

The most common task is to add a custom network-facing SMPP plug-in using the south interfaces. It is also possible to create a custom application-facing SMPP module using the north interfaces. The following procedures cover both scenarios.

The basic steps for creating a custom SMPP plug-in are as follows:

1. Using the Services Gatekeeper SCE PDS wizard, generate a customized network plug-in for the SMPP communication service.

   You can also use this wizard to create a custom interceptor, if necessary.

   See "Communication Service Example" for a description of a generated project.

2. Create the service type for the customized plug-in by extending the ServiceType class.

   When the plug-in registers itself, an object of this type is passed to the Plug-in Manager.

3. Implement the **ManagedPluginService** interface.

   This class activates, deactivates and initializes the plug-in service. It implements the **PluginService**, **PluginServiceLifecycle** and **PluginInstanceFactory** interfaces.

   See "Understanding Communication Service Components" and "Understanding the Communication Service Project Output" for more information.

4. Implement the **ManagedPluginInstance** interface.

   This class activates a plug-in instance that has been created with the Plug-in Manager, after which the plug-in should register its MBeans and prepare to accept traffic. The plug-in service that activates this plug-in instance must be in the ACTIVE (ADMIN) or ACTIVE (RUNNING) state when the **activate** method is called.

   This class also initializes and deactivates the plug-in instance, determines whether the plug-in instance is capable of servicing the current request, and sets up the session information cache.

   See "Understanding Communication Service Components" for more information.

5. Extend and implement the SMPPPluginMBean interface and register the MBean using the SMPP API.

6. If you are implementing a network-facing SMPP module, extend and implement the SMPPPluginSouth interface to process network-triggered events received from SMPPServiceSouth. See the **oracle.ocsg.protocol.smpp.plugin** package in *Services Gatekeeper Java API Reference* for the list of methods in this interface. See also Using

the SMPP APIs.

7. Send the processed requests and responses to the application using the SMPPServiceNorth interface.
8. If you are implementing an application-facing SMPP module, extend and implement the SMPPPluginNorth interface to process application-initiated events received from SMPPServiceNorth. See the oracle.ocsg.protocol.smpp.plugin package in *Services Gatekeeper Java API Reference* for the list of methods in this interface. See also Using the SMPP APIs.
9. Send the processed requests and responses to the SMSC using the SMPPServiceSouth interface.
10. Maintain a session information class to cache session values such as client and server connection IDs, source and destination addresses, whether a delivery notification is required, and so on.
11. Create CDRs and EDRs to trace the message flow, if necessary.

    See "Understanding Aspects, Annotations, EDRs, Alarms, and CDRs" for more information.
12. Build the plug-in project and create EAR package, which will be deployed to Services Gatekeeper.

    Make sure that the **smpp_api.jar** is in the build class path; for example:

    ```
    <path path="${target.dir}/protocol/modules/smpp_
    api/oracle.ocsg.protocol.smpp_api_6.0.0.0.jar"/>
    ```
13. Use the Platform Test Environment (PTE) to test and debug the plug-in.

    See *Services Gatekeeper Platform Test Environment User's Guide* for information.

Configuration Settings Affecting SMPP Connections

The System Administrator can configure several attributes that control how the SMPP Service manages connections.

The System Administrator can also set some parameters on how the SMPP Service behaves on a per application basis, such as whether certain operations are allowed after sending a short message or whether network-triggered notification is enabled. These parameters are set using the **addApplicationSpecificSettings** operation.

These settings can affect how requests and responses should be processed before they are sent. The SMPP Service API provides methods for querying some of these settings. See "SMPPService" for more information.

For a complete list of the SMPP Service attributes and operations, see the reference material for the SMPP server service in the “Native SMPP” in *Services Gatekeeper Communication Service Reference Guide*.

About the SMPP Interfaces

The packages for developing an SMPP plug-in are:

- oracle.ocsg.protocol.common
- oracle.ocsg.protocol.smpp.service
- oracle.ocsg.protocol.smpp.plugin
- oracle.ocsg.protocol.smpp.common

- oracle.ocsg.protocol.smpp.event

oracle.ocsg.protocol.common

The oracle.ocsg.protocol.common package includes the ProtocolServiceProxyFactory interface, which is derived from the AbstractProtocolService class. This is the base class for the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods.

The SMPP plug-in implementations use the **getProtocolServiceNorth** method to get a reference to the interface used to send PDUs to applications on server connections and the **getProtocolServiceSouth** method to get a reference to the interface used to send PDUs to SMSCs on client connections.

This package also includes the ProtocolServiceNorth and ProtocolServiceSouth interfaces from which the SMPPServiceNorth and SMPPServiceSouth interfaces are derived.

oracle.ocsg.protocol.smpp.service

The oracle.ocsg.protocol.smpp.service package includes the interfaces for the SMPP Service:

- SMPPService
- SMPPServiceNorth
- SMPPPluginSouth

SMPPService

This interface provides methods for generic SMPP Service tasks. These include checking whether available or active client connections exist for a plug-in instance, registering the SMPP work manager, and registering the plug-in MBean object, which exposes configurable attributes and operations to the SMPP Service.

It provides methods for querying the following SMPP Service configuration settings:

- **ConnectionBasedRouting**: an attribute in the SMPP service
- **LooseBinding**: an attribute in the SMPP service
- **notificationEnabled**: an application-specific setting in the SMPP Service
- **subsequentOperationsAllowed**: an application-specific setting in the SMPP Service

For details about these settings, see s the reference material for the SMPP server service in the Native SMPP chapter in *Services Gatekeeper Communication Service Reference Guide*.

SMPPServiceNorth

The SMPPServiceNorth interface maintains a server connection pool that provides connections between Services Gatekeeper and applications. Services Gatekeeper is a server in this relationship.

When the application sends a successful **BIND** request to Services Gatekeeper, the plug-in obtains a server connection from the server connection pool and uses the implementation of the SMPPServiceNorth interface to send messages to the application.

The server connection:

- Receives messages from the application.
- Invokes the SMPPPluginNorth interface through a proxy.
- Sends messages to the application.
- Manages SMPP timers and windowing toward the application.
- Stores transaction mapping information in cache.

This interface provides the following methods to send northbound requests and responses submitted by the plug-in: **cancelSmResponse**, **dataSm**, **dataSmResponse**, **deliverSm**, **querySmResponse**, **replaceSmResponse**, **submitSmMultiResponse**, **submitSmResponse**.

SMPPServiceSouth

The SMPPServiceSouth interface maintains a client connection pool that provides connections between Services Gatekeeper and Short Message Service Centers (SMSCs). Services Gatekeeper is a client in this relationship.

The service processes **BIND** and **UNBIND** requests from the plug-in and obtains client connections on which to perform SMPP operations toward the SMSC.

The client connection:

- Receives messages from the SMSC.
- Invokes the SMPPPluginSouth interface through a proxy.
- Sends messages to the SMSC.
- Manages SMPP timers and windowing toward the SMSC.
- Stores transaction mapping information in cache.

This interface provides the following methods to send southbound requests and responses submitted by the plug-in: **bind**, **cancelSm, dataSm**, **dataSmResponse**, **deliverSmResponse**, **querySm**, **replaceSm**, **submitSm**, **submitSmMulti**, **unbind**.

oracle.ocsg.protocol.smpp.plugin

The oracle.ocsg.protocol.smpp.plug-in package defines the interfaces between the SMPP service and the SMPP plug-ins:

- SMPPServiceNorth
- SMPPServiceSouth
- SMPPPluginMBean

The plug-in developer extends and implements these interfaces for a custom plug-in.

SMPPPluginNorth

A plug-in extends and implements the SMPPPluginNorth interface to process the following supported application-initiated operations:

- BIND
- CANCEL_SM
- DATA_SM
- DATA_SM_RESPONSE
- DELIVER_SM_RESPONSE

- QUERY_SM
- REPLACE_SM
- SUBMIT_SM
- SUBMIT_SM_MULTI

The SMPPPluginNorth implementation uses the SMPPServiceSouth interface to send these operations to the SMSC.

SMPPPluginSouth

The plug-in extends and implements the SMPPluginSouth interface to process supported network-triggered operations, such as:

- CANCEL_SM_RESPONSE
- DATA_SM
- DATA_SM_RESPONSE
- DELIVER_SM
- QUERY_SM_RESPONSE
- REPLACE_SM_RESPONSE
- SUBMIT_SM_MULTI_RESPONSE
- SUBMIT_SM_RESPONSE
- UNBIND

The SMPPPluginSouth implementation uses the SMPPServiceNorth interface to send these operations to the application.

SMPPPluginMBean

This interface defines the network-facing connection attributes of the plug-in. A custom plug-in extends and implements this interface to provide the facilities to manage and query the plug-in.

The SMPPPluginNorth and SMPPPluginSouth implementations use this interface to query values in the plug-in while processing requests and responses.

oracle.ocsg.protocol.smpp.common

This package provides the SMPPException class.

oracle.ocsg.protocol.smpp.event

This package provides classes for SMPP events.

Using the SMPP APIs

The basic procedure for processing and sending an incoming request or response through the SMPP Service is as follows:

1. Get the SMPPService object.
2. Process the fields in the incoming request or response.

Depending on the particular request or response typical processing may involve setting various fields in the request or response. For a response, you may need to process event data from the original request.

It may be necessary to query some SMPP Service configuration settings using the SMPPService methods. See "SMPPService" for more information.

3. Get the SMPPService object's protocol interface for sending data.

   For sending data to the SMSC, you need the interface for SMPPServiceSouth to get a client-side connection. For sending data to the application, you need the interface for SMPPServiceNorth to get a server-side connection.

4. Send the request or response using the methods provided by the SMPPServiceNorth or SMPPServiceSouth.

The following sections illustrate how the SMPP Server APIs and settings are used in processing some requests and responses. They focus on sample tasks involving the SMPP API. Logging, exception handling, session information management, alarm creation, and other tasks not using the SMPP APIs are not considered.

- Processing a BIND Request from an Application
- Processing a SUBMIT_SM Request from an Application
- Processing a SUBMIT_SM Response from the SMSC
- Processing a DELIVER_SM Request from the SMSC
- Processing a DELIVER_SM Response from an Application

These are among the tasks performed in custom SMPPPluginNorth and SMPPPluginSouth implementations.

Processing a BIND Request from an Application

When the plug-in receives a **BIND** request from an application, the SMPPPluginNorth class processes the request and sends it to the SMSC.

The SMPPPluginNorth **bind** method:

1. Gets the plug-in instance id and sets it in the request.
2. Gets the SMPP Service object.
3. Gets the service object's protocol interface for sending data on a client connection.
4. Sends the request using the SMPPServiceSouth's **bind** method.

For example:

```
public BindResponse bind(Bind request) {
    BindResponse bindResp = null;

    // Set the plug-in instance id
    request.setPluginInstanceId(plugin.getPluginInstanceId());

    // Get the SMPP service object
    SMPPService smppService = plugin.getSMPPService();

    // Get the interface for sending data on a client-side connection
    SMPPServiceSouth serviceSouth =
smppService.getProtocolServiceSouth(SMPPServiceSouth.class);
    // Send the request
      bindResp = serviceSouth.bind(request);
```

```
        return bindResp;
}
```

Processing a SUBMIT_SM Request from an Application

When a plug-in receives a **SUBMIT_SM** request from an application, the SMPPPluginNorth class processes the request and sends it to the SMSC.

The SMPPPluginNorth **submitSm** method:

1. Gets the plug-in instance and application instance IDs and sets them in the request.
2. Queries the SMPP Service's application-specific **notificationEnabled** setting and sets the **registeredDelivery** field in the request accordingly.

   ```
   if (request.getRegisteredDelivery() != 0 &&
   !plugin.isNotificationAllowed(aigId)) {
         request.setRegisteredDelivery(0);
   }
   ```

3. Gets the SMPP Service object.

   ```
   SMPPService smppService = plugin.getSMPPService();
   ```

4. Gets the service object's protocol interface for sending data on a client connection.

   ```
   SMPPServiceSouth serviceSouth =
   smppService.getProtocolServiceSouth(SMPPServiceSouth.class)
   ```

5. Process any extra parameters (xparams) in the request.
6. Sends the request using the SMPPServiceSouth's **submitSm** method.

   ```
   serviceSouth.submitSm(request);
   ```

Processing a SUBMIT_SM Response from the SMSC

When a plug-in receives a **SUBMIT_SM_RESPONSE** from the SMSC, the SMPPPluginSouth class processes the response and sends it to the application.

The SMPPPluginSouth **submitSmResponse** method:

1. Gets the SMPP Service object.

   ```
   SMPPService smppService = plugin.getSMPPService();
   ```

2. Gets the plug-in message ID and sets it in the response.
3. Gets and processes the request event data from the original request to which this is the response.
4. Queries the SMPP Service and application-specific settings to determine whether a delivery receipt will be provided. For example, the following example checks the **notificationEnabled** and **isSubsequentOperationsAllowed** application-specific settings and the **ConnectionBasedRouting** SMPP Service attribute.

   ```
   boolean needDR = plugin.isNotificationAllowed(aigId) &&
   originalRequest.getRegisteredDelivery() != 0 &&
          !plugin.isConnectionBasedRoutingEnabled();
         if (plugin.isSubsequentOperationsAllowed(aigId) || needDR) {
   ```

```
            // Set the session information accordingly . . .
    }
```

5. Gets the SMPP Service object's interface for sending data on a server-side connection.

   ```
   SMPPServiceNorth serviceNorth =
   smppService.getProtocolServiceNorth(SMPPServiceNorth.class);
   ```

6. Sends the response on that connection using SMPPService North's **submitSmResponse** method.

   ```
   serviceNorth.submitSmResponse(response);
   ```

Processing a DELIVER_SM Request from the SMSC

A **DELIVER_SM** request from the SMSC can be a simple SMS message from the network, or it can be the SMSC sending a delivery receipt for a previously submitted **SUBMIT_SM** request.

When a plug-in receives a **DELIVER_SM** request from the SMSC, the SMPPPluginSouth **deliverSm** method first examines the isDeliveryReceipt field in the request to determine whether the request is for a delivery receipt or a network-triggered SMS message. For example:

```
public void deliverSm(final DeliverSm request)  {
    request.setPluginInstanceId(plugin.getPluginInstanceId());
    final boolean isDeliverReceipt = request.isDeliverReceipt();

    if (isDeliverReceipt) {
      deliverSmForDeliveryReceipt(request);
    } else {
      deliverSmForMO(request);
    }
  }
```

If the **DELIVER_SM** request is not for a delivery receipt, the processing is simple. The SMPPPluginSouth's method for processing the request:

1. Gets the SMPP Service object.
2. Gets the SMPP Service object's interface for sending data on a server-side connection.

   ```
   SMPPServiceNorth serviceNorth =
   smppService.getProtocolServiceNorth(SMPPServiceNorth.class);
   ```

3. Sends the request using the SMPPServiceNorth's **deliverSm** method.

   ```
   serviceNorth.deliverSm(request)
   ```

If the request requires a delivery receipt, the SMPPPluginSouth method for processing the request performs some additional tasks before sending the request:

1. Gets and sets the receipted message ID in the request.

   ```
   String msgId = createPluginMessageId(request.getReceiptedMessageId());
   request.setReceiptedMessageId(msgId);
   ```

2. Using the plug-in's implementation of the SMPPPluginMBean, gets the response command status.

   ```
   failureCommandStatus =
   ```

```
plugin.getManagement().getMySMPPPluginMBean().getDeliverSmRespCommandStatus();
```

You would implement the **getDeliverSmCommandStatus** method in your SMPPPluginMBean class to get the outcome of the **DELIVER_SM** request. The status should indicate whether the application was reached.

3. Uses the SMPPService **isConnectionBasedRouting** method to establish whether connection-based routing is enabled in the SMPP Service and processes the request accordingly.

   If connection-based routing is enabled, the operator can send a delivery receipt to a site other than the one through which the original message was submitted. See the discussion of connection-based routing in *Services Gatekeeper Communication Service Reference Guide* for information about how connection-based routing works in combination with other configuration settings.
4. Queries any additional relevant configuration settings for the plug-in using the custom management methods implemented by the plug-in in the SMPPPluginMBean and processes accordingly. For example, you may want to query whether to delete SMPP session information after the delivery receipt is received.
5. Uses the SMPPService **isSubsequentOperationsAllowed** method to query whether subsequent operations are allowed for the application instance and sets the session information accordingly.
6. Gets an SMPP Service object.
7. Gets the SMPPService object's interface for sending data on a server-side connection.
8. Sends the request using the SMPPServiceNorth's **deliverSm** method.

Processing a DELIVER_SM Response from an Application

A **DELIVER_SM** response from an application can be the response for the receipt of an mobile-originated SMS message or of a delivery receipt.

The SMPPPluginNorth **deliverSmReponse** method gets the original request event associated with the response, determines whether the response if for a delivery receipt, and passes the request as well as the response to the method that will process and send the response.

```
public void deliverSmResponse(DeliverSmResponse response) {
    DeliverSm originalRequest = (DeliverSm)response.getRequestEvent();
    if (originalRequest != null && !originalRequest.isDeliverReceipt()) {
      deliverSmResponseForMO(response, originalRequest);
    } else {
      deliverSmResponseForDeliveryReceipt(response, originalRequest);
    }
 }
```

The appropriate **deliverSmResponse** method processes any information needed from the response and its associated request.

A method that processes a response for a mobile-originated SMS message may need to construct EDR data before sending the response to the SMSC using the SMPPServiceSouth **deliverSmResponse** method.

9 Using the UCP API

This chapter provides an overview of the Oracle Communications Services Gatekeeper Universal Computer Protocol (UCP) API Java interface. It also contains some guidance on how to develop a customized UCP plug-in using the Services Gatekeeper Platform Development Studio and the UCP APIs.

Understanding the UCP Protocol API

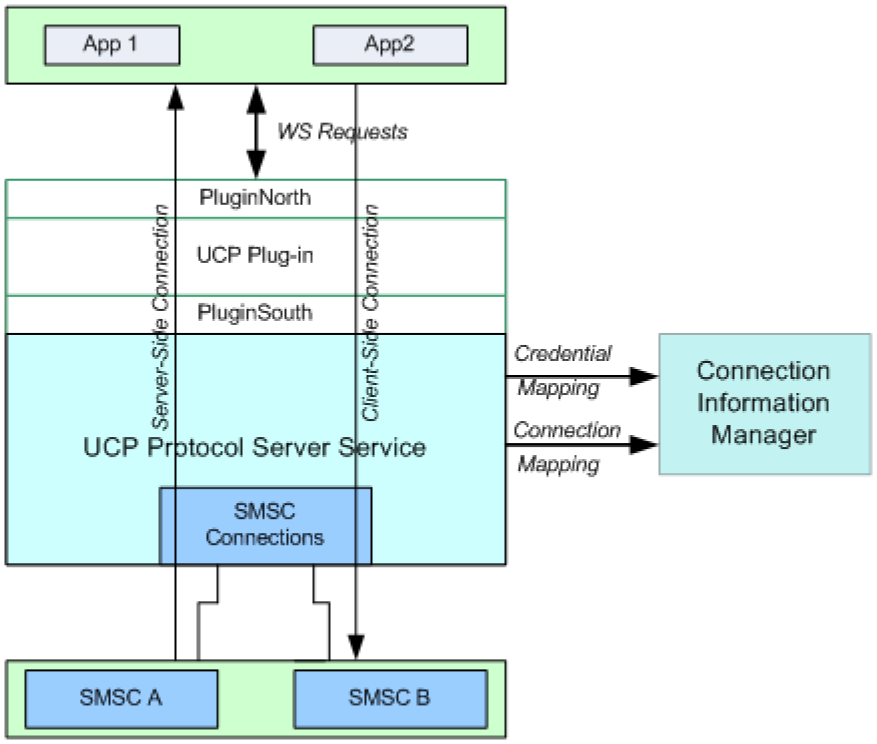
The UCP protocol APIs enable platform developers to create custom UCP plug-ins without having to set up and manage connections from Services Gatekeeper to applications and SMSCs.

The UCP Protocol Server Service manages the low-level connectivity details, in conjunction with a configurable Connection Information Manager service, which stores mappings between plug-in instances and the hosts and ports and mappings between application instances and network node credentials.

Using the Protocol Server Service APIs, a plug-in obtains a connection to an application or SMSC and sends a protocol data unit (PDU) or acknowledgement on that connection. The APIs include classes for constructing UCP PDUs.

Figure 9–1 shows the UCP architecture.

Figure 9–1 UCP Architecture in Services Gatekeeper



A client-side connection is a connection between Services Gatekeeper and the SMSC, since Services Gatekeeper acts as client in this relationship. In the context of this architecture, a server-side connection is a connection between an application and Services Gatekeeper, since Services Gatekeeper acts as server in this relationship.

UCP Protocol Server Service

The UCP Protocol Server Service provides connection services on behalf of UCP plug-ins. It communicates with external applications and SMSCs using UCP over TCP/IP. This service:

- Sends and receives UCP data from the socket.
- Constructs the UCP PDU.
- Associates the current PDU with the correct application instance.
- Calls the plug-in.

All requests from a plug-in instance to the Protocol Server Service contain a plug-in instance ID. The Protocol Server Service performs connection and network credential mapping based on the configuration set up in the Connection Information Manager.

The UCP Protocol Service API defines the interface between the UCP Protocol Server Service and UCP plug-ins. See the oracle.ocsg.protocol.ucp and oracle.ocsg.protocol.ucp.pdu packages in *Services Gatekeeper Java API Reference* for documentation on this API.

The Protocol Server Service is a standard Services Gatekeeper WLS service. You can access it from the Administration console as **UCPService** under **es**.

Understanding the Connection Information Manager Service

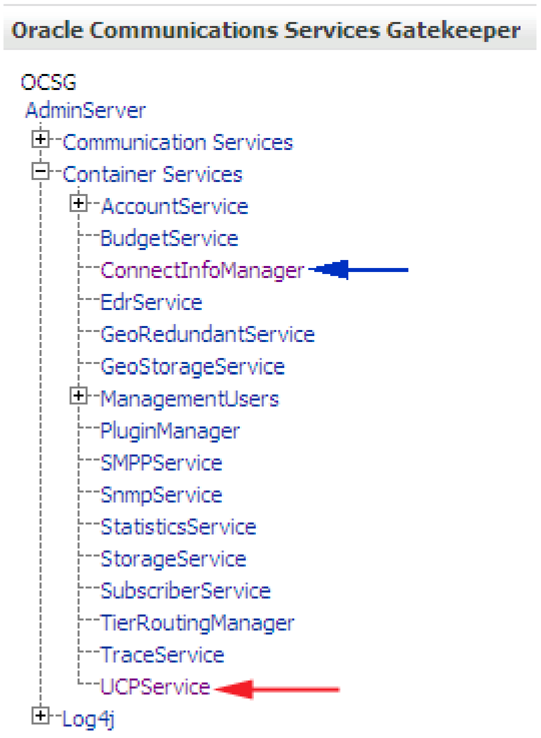
The Connection Information Manager is a standard Services Gatekeeper service, which creates and stores connection and credential mappings that UCP plug-in instances need to connect to network elements and applications.

The UCP Protocol Service uses the Connection Information Manager to map plug-instance IDs to SMSC IP addresses and ports.

You can also optionally configure in the Connection Information Manager the local address and port to bind to when setting up a client-side connection to an SMSC. When Services Gatekeeper connects to the remote network node, it uses the specified local host IP address and port combination to bind the socket on the Services Gatekeeper side of the connection. The Protocol Service uses the specified port as a starting offset and increments the port number by one for each additional connection additional associated with the same plug-in instance ID. If the local host address is not configured, an ephemeral port is used.

You manage connection information settings from the Administration console. See **ConnectInfoManager** under **es**, as shown in Figure 9–2. See "Managing and Configuring Native UCP Connections" in *Services Gatekeeper System Administrator's Guide* for information about specific operations.

Figure 9–2 UCP Protocol Server Service and Connection Information Manager in the Administration Console



PluginNorth

A plug-in implements the PluginNorth interface to perform the following tasks on behalf of application-initiated requests:

- Send a mobile-terminated (MT) SMS message
- Open a UCP session

- Send an ACK to the SMSC
- Send a NACK to the SMSC

You would extend and implement this interface to add a new application-facing UCP protocol plug-in.

PluginSouth

A plug-in implements the PluginSouth interface to perform the following tasks on behalf of network-triggered requests:

- Deliver a mobile-originated (MO) SMS message
- Deliver a message delivery notification associated with a previously-sent MT SMS
- Send an ACK to the application
- Send a NACK to the application

You would extend and implement this interface to add a new network-facing UCP protocol plug-in.

Additional Information You Will Need

In addition to the information in this chapter, developers should consult the following documents for information on how to build a UCP plug-in:

- *Services Gatekeeper Java API Reference*

  Of special interest are the following packages, which include the interfaces and classes for the UCP Protocol Server Service:

  - **oracle.ocsg.protocol.ucp**
  - **oracle.ocsg.protocol.ucp.pdu**
  - **oracle.ocsg.protocol.common**

  The following packages include the plug-in interfaces and classes for the Native SMPP plug-in, which is part of the standard Services Gatekeeper Native UCP communication service. They can serve as a reference for developing customized north and south UCP plug-ins.

  - oracle.ocsg.plugin.nativefacade.ucp.north
  - oracle.ocsg.plugin.nativefacade.ucp.south

  In addition, you will need resources from various generic packages such as:

  - **com.bea.wlcp.wlng.api.edr**
  - **com.bea.wlcp.wlng.api.managemen**t
  - **com.bea.wlcp.wlng.api.plugin**
  - **com.bea.wlcp.wlng.api.plugin.commo**n
  - **com.bea.wlcp.wlng.api.plugin.context**
  - **com.bea.wlcp.wlng.api.util**

- *Services Gatekeeper Communication Service Reference Guide*

  See the discussion on Native UCP. This discussion provides an overview of the Services Gatekeeper Native UCP communication service. It documents the attributes and operations provided to manage the UCP Protocol Server Service.

The protocol server service is available for any UCP plug-in to access using the UCP Protocol Server Service APIs.

- *Services Gatekeeper System Administrator's Guide*

  See the connection information discussion. The Connection Information Manager creates and stores connection and credential mappings used by UCP plug-ins.

Procedure for Creating a Customized UCP Plug-in

This procedure outlines the basic steps to perform to add a custom UCP plug-in.

1. Using the Services Gatekeeper SCE PDS Wizard, generate a customized network plug-in for the UCP communication service.

   You can also use this wizard to create a custom interceptor, if necessary.

   See "Understanding the Communication Service Project Output" for more information.

2. Create the service type for the customized plug-in by extending the **ServiceType** class.

   When the plug-in registers itself, an object of this type is passed to the Plug-in Manager.

3. Implement the **ManagedPluginService** interface. This class activates, deactivates and initializes the plug-in service. It implements the **PluginService**, **PluginServiceLifecycle** and **PluginInstanceFactory** interfaces.

   See "Understanding Communication Service Components" for more information.

4. Implement the ManagedPluginInstance interface.

   This class activates a plug-in instance that has been created with the Plug-in Manager, after which the plug-in should register its MBeans and prepare to accept traffic. The plug-in service that activates this plug-in instance must be in the ACTIVE (ADMIN) or ACTIVE (RUNNING) state when the **activate** method is called.

   This class also initializes and deactivates the plug-in instance and determines whether the plug-in instance is capable of servicing the current request.

   See "Understanding Communication Service Components" for more information.

5. If you are implementing an application-facing UCP module, extend and implement the PluginNorth interface: **SubmitSm**, **openSession**, **ack** and **nack**.

6. If you are implementing a network-facing UCP module, extend and implement the PluginSouth interface: **deliverSm**, **deliveryNotification**, **ack** and **nack**.

7. Create CDRs and EDRs to trace the message flow, if necessary.

8. From the Administration console, configure the connection and credential mappings in the Connection Information Manager.

   See the discussion on managing and configuring connection information in *Services Gatekeeper System Administrator's Guide*.

9. Build the plug-in project and create the EAR package, which will be deployed to Services Gatekeeper.

10. Use the Platform Test Environment (PTE) to test and debug the plug-in.

    See *Services Gatekeeper Platform Test Environment User's Guide* for more information.

About the UCP Protocol Server Service Interfaces

The packages for the protocol server service are:

- oracle.ocsg.protocol.common
- oracle.ocsg.protocol.ucp
- oracle.ocsg.protocol.ucp.pdu

Using the UCP Protocol Server Service API, you can develop a custom UCP plug-in without having to implement the low-level connection functionality. The API provides a wrapper to bind, send, and receive messages and allows customization of PDUs.

oracle.ocsg.protocol.common

The **oracle.ocsg.protocol.common.package** provides four basic interfaces from which the UCP Protocol Server Service APIs are derived:

- **AbstractProtocolService**

  This is the base class for the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods. The UCPNetworkingServiceImpl class inherits from AbstractProtocolService to implement these methods.

- **ProtocolServiceProxyFactory**

  Gets references to the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods for use by the plug-in.

- **ProtocolServiceNorth**

  This is the base interface for creating network-facing connections.

- **ProtocolServiceSouth**

  This is the base interface for creating application-facing connections.

oracle.ocsg.protocol.ucp

The main protocol server service interfaces used by a UCP plug-in are:

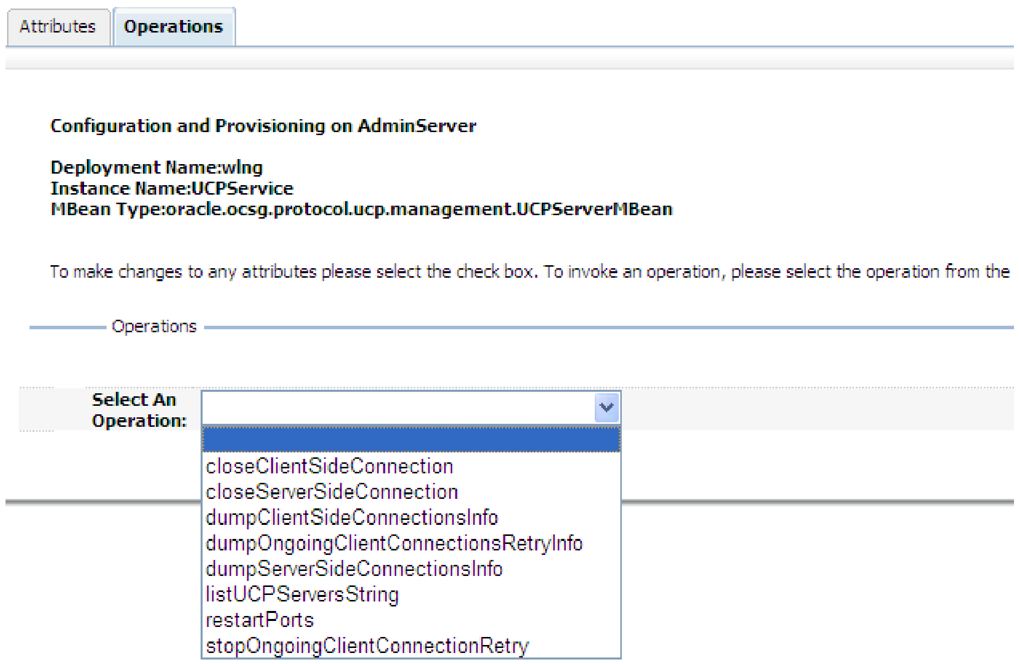
- **UCPNetworkingService**

  The **UCPNetworkingService** interface provides methods to add, remove, list and otherwise manage server-side and client-side connections. See the **UCPNetworkingService** interface in the **oracle.ocsg.protocol.ucp** package in the *Services Gatekeeper Java API Reference* for a list of the methods in this interface.

  In addition to accessing these methods programmatically, a System Administrator can also access most of the methods in the **UCPNetworkingService** interface as operations and management (OAM) operations from the **UCPService** pane of the Administration console.

  Figure 9–3 shows how to access the **UCPService** pane.

Figure 9–3 Protocol Service Operations in Administration Console



Configuration and Provisioning on AdminServer

Deployment Name:wlng
Instance Name:UCPService
MBean Type:oracle.ocsg.protocol.ucp.management.UCPServerMBean

To make changes to any attributes please select the check box. To invoke an operation, please select the operation from the

Operations

Select An Operation:

- closeClientSideConnection
- closeServerSideConnection
- dumpClientSideConnectionsInfo
- dumpOngoingClientConnectionsRetryInfo
- dumpServerSideConnectionsInfo
- listUCPServersString
- restartPorts
- stopOngoingClientConnectionRetry

- **UCPNetworkingServiceClient**

  This interface implements methods for sending PDUs, ACKs, and NACKs on a client-side connection. It extends the Services Gatekeeper **oracle.ocsg.protocol.common.ProtocolServiceSouth** interface.

  The plug-in uses this interface's **sendPDUOnClientConnection** method to send the plug-in instance ID and the PDU. The method returns a connection ID that identifies the connection to the SMSC on which the request was sent.

- **UCPNetworkingServiceServer**

  This interface implements methods for sending PDUs, ACKs, and NACKs on a server-side connection. It extends the Services Gatekeeper **oracle.ocsg.protocol.common.ProtocolServiceNorth** interface.

- The plug-in uses the **sendPDUOnServerConnection** method to send the connection ID and the PDU. The method returns a connection ID that identifies the connection to the application on which the request was sent.

oracle.ocsg.protocol.ucp.pdu

This package provides utility classes for building UCP PDUs for the supported UCP operations. This package provides classes for all of the supported UCP abstract data types (ADTs), as well as a generic UCP ADT, UCP constants, headers, and parameters

Connection Mapping

The Protocol Server Service uses mappings between application instances to network nodes configured in the Connection Information Manager to set up the connections that are used by the plug-ins.

At a minimum you need to configure the credential map, host address, and user password using these operations:

- **createOrUpdateCredentialMap**
- **createOrUpdateRemoteHostAddress** or **createOrUpdateLocalHostAddress**
- **createOrUpdateListenAddress**
- **createOrUpdateUserPasswordCredentialEntry**

There are various possible mapping logics; for example:

- One connection to the SMSC for all Services Gatekeeper applications
- One connection to the SMSC for a group of Services Gatekeeper applications
- One connection to the SMSC for each Services Gatekeeper application

The simplest scenario is to configure a plug-in always to use the same application instance for all UCP requests. This requires only one connection to the SMSC. You would create the application instance in Services Gatekeeper and dedicate it to UCP southbound requests in the Connection Information Manager. Before making the call to the UCP Protocol Server Service, the plug-in can switch context to the UCP-dedicated application instance

Another scenario would configure the plug-in to use the current application instance to send requests through the service. This results in multiple connections to the SMSC, at least one per application instance. In this case, you must configure the Connection Information Manager with connection credentials and SMSC address and port mappings for all application instances.

There is no single correct solution. The mapping logic that you choose depends on the demands of your situation.

OAM Attributes Affecting UCP Network Connectivity

The **UCPProtocol** read-only attribute contains the UCP protocol string. This value is set to the listen address defined by the **createOrUpdateListenAddress** operation in the Connection Information Manager.

In the Administration console, you can configure two attributes that control how the Protocol Server Service handles reconnection attempts:

- **MaxReconnectAttempts**: Specifies the maximum number of reconnection attempts permitted. Set to -1 for no maximum, 0 for no reconnection attempts, or a positive integer indicating the maximum number of reconnections to attempt.
- **TimeBetweenReconnectAttempts**: Specifies the time in milliseconds between reconnection attempts.

Using the APIs

The first three examples in this section provide some guidance related to common tasks using the UCP APIs that would be performed by a custom application-facing UCP plug-in that implements and extends PluginNorth. The examples are based on a prototype for a ParlayX2.1 SMS plug-in. The last example is for a PluginSouth implementation processing a **DELIVER_SM** request.

The tasks include:

- Sending a submitSm Request to the SMSC

- Creating a UCP PDU
- Sending an openSession Request to the SMSC
- Sending a DeliverSm to an Application

Sending a submitSm Request to the SMSC

When a plug-in receives a **SUBMIT_SM** request from an application, the PluginNorth implementing class processes the parameters in the request, constructs the PDU, and sends it to the SMSC using the **UCPNetworkingServiceClient** APIs.

The plug-in:

1. Gets the UCP **NetworkingService** object. For example:

   ```
   UCPNetworkingService ucpService =
   PX21UCPPluginInstanceImpl.getUCPNetworkingService();
   ```

2. Gets any outstanding standing **SUBMIT_SM** requests.
3. Creates the submit PDU, using the classes in the oracle.ocsg.protocol.ucp.pdu package. See "Creating a UCP PDU" for more information.
4. Gets the source connection ID.
5. Gets the UCP **NetworkingService** protocol interface for sending data on a client connection. For example:

   ```
   UCPNetworkingServiceClient client =
   ucpService.getProtocolServiceSouth(UCPNetworkingServiceClient.class);
   ```

6. Sends the PDU on the client connection, using the UCPNetworkingServiceClient **sendPDUOnClientConnection** method. For example:

   ```
   clientConnectionID = client.sendPDUOnClientConnection
   (
       px21UCPPluginInstanceImpl.getPluginInstanceId(),
       px21UCPPluginInstanceImpl.getSourceServerPort(),
       submitSMPDU,
       outstandingSubmitSMRequests,
       sourceConnectionID
   );
   ```

Creating a UCP PDU

To create a UCP PDU, you can use the classes in the **oracle.ocsg.protocol.ucp.pdu** package.

The following method creates the **submitSM** PDU used in "Sending a submitSm Request to the SMSC". It uses the using the **UcpHeader**, **UcpParameter**, **GenericUcpAdt** classes defined in the **pdu** package.

```
private UcpPDU createSubmitSMPDU(SendSms parameters) {
        UcpHeader ucpHeader = new UcpHeader();
        UcpParameter orParam = new UcpParameter("O");
        ucpHeader.setParameter(UcpHeader.PARAM_OR, orParam);
        UcpParameter otParam = new UcpParameter(UcpConstants.OT_SUBMIT_SHORT_
MESSAGE);
        ucpHeader.setParameter(UcpHeader.PARAM_OT, otParam);
        UcpParameter trnParam = new UcpParameter("01");
        ucpHeader.setParameter(UcpHeader.PARAM_TRN, trnParam);
        UcpParameter lenParam = new UcpParameter("00000");
```

```
        ucpHeader.setParameter(UcpHeader.PARAM_LEN, lenParam);

        GenericUcpAdt data = new GenericUcpAdt(33);

        //ADC
        URI[] destAddresses = parameters.getAddresses();
        String uriStringDestAddress = destAddresses[0].toASCIIString();

        //Strip "tel:"
        String destAddressString = stripURIPrefix(uriStringDestAddress);
        UcpParameter adcParam = new UcpParameter(destAddressString);
        data.setParameter(Ucp50Adt.PARAM_ADC, adcParam);

        //OADC
        String senderName = parameters.getSenderName();
        UcpParameter oadcParam = new UcpParameter(senderName);
        data.setParameter(Ucp50Adt.PARAM_OADC, oadcParam);

        //NRQ and NT
        SimpleReference simpleRef = parameters.getReceiptRequest();
        String nrq = "";
        String nt = "";
        if(simpleRef != null){
            nrq = "0"; //0 == NADC not used
            nt = "7"; // 7 == all
        }
        UcpParameter nrqParam = new UcpParameter(nrq);
        data.setParameter(Ucp50Adt.PARAM_NRQ, nrqParam);
        UcpParameter ntParam = new UcpParameter(nt);
        data.setParameter(Ucp50Adt.PARAM_NT, ntParam);

        //If LRq is empty, the contents of LRAd and LPID are ignored

        //Message type 3 == "Alphanumeric message encoded into IRA characters."
        UcpParameter mtParam = new UcpParameter("3");
        data.setParameter(Ucp50Adt.PARAM_MT, mtParam);

        String message = parameters.getMessage();
        String iraEncodedMessage = iraEncodeMessage(message);
        UcpParameter msgParam = new UcpParameter(iraEncodedMessage);
        data.setParameter(Ucp50Adt.PARAM_MSG, msgParam);

        return new UcpPDU(ucpHeader, data);
    }
```

Sending an openSession Request to the SMSC

A connection from the UCP plug-in to the SMSC is implicitly established on receipt of the **openSession** request. Upon receiving the **openSession** request, the Protocol Server Service uses the current context as a key to determine the connection and credential mapping to use for the new connection that it is creating. The user and plug-in instance ID must therefore be configured in the Connection Information Manager before the **openSession** request is sent; otherwise the **openSession** request fails.

The APIs do not provide a specific open session method.

To create a new session to the SMSC, create an **openSession** PDU using the pdu package and use the **sendPDUOnClientConnection** with a that **openSessionPDU** as the PDU parameter:

```
UCPNetworkingServiceClient client =
ucpService.getProtocolServiceSouth(UCPNetworkingServiceClient.class);

String connectionID = client.sendPDUOnClientConnection
    (myUCPPluginInstanceImpl.getPluginInstanceId(),
     sourceServerPort,
     openSessionPDU,
     outstandingOpenSessionRequests,
     sourceConnectionId);
```

A UCP plug-in uses the Protocol Server Service API after it receives an **openSession** PDU. The UCP Protocol Server Service creates a new socket connection for each session management operation of subtype openSession that is sent. The created connections are later used for sending **SUBMIT_SM** requests.

Sending a DeliverSm to an Application

When a plug-in receives a **DELIVER_SM** request from the SMSC, the **PluginSouth** implementing class processes the parameters in the request.

If the plug-in is communicating with a web services-based application, it typically analyzes the parameters in the request to find the correct application callback reference (URL) to which the mobile-originated SMS message should be sent and then sends it.

If the plug-in is communicating with a UCP-based application, it typically constructs a DELIVER_SM PDU, which it sends to the application-facing UCP **NetworkingServerService** APIs.

After notifying the application of the message, the plug-in should send an ACK or NACK to the SMSC to report whether the notification was successful.

The following process flow is for a plug-in communicating with a web services-based application:

1. Gets the UCP **NetworkingService** object. For example:

   ```
   UCPNetworkingService ucpService =
   PX21UCPPluginInstanceImpl.getUCPNetworkingService();
   ```

2. Processes the incoming **deliverSM** PDU to get the source and destination addresses. This implementation uses the **UCP50Adt** class to extract the data from the PDU:

   ```
   String destinationAddress = deliverSMPDU.getData().getParameter(Ucp50Adt.PARAM_
   ADC).getValueAsString();
   String originatingAddress = deliverSMPDU.getData().getParameter(Ucp50Adt.PARAM_
   OADC).getValueAsString();
   ```

3. Gets the notification callback references.
4. Implements support for using criteria and storing the mobile-originated message.
5. Creates the **deliverSM** PDU.
6. Send the **deliverSM** notification PDU. For example:

   ```
   boolean notificationOK = sendDeliverSMNotification(callbackRef,
   destinationAddress, originatingAddress, deliverSMPDU);
   ```

7. Send ACK or NACK to the SMSC depending on the outcome of the notification. For example:

   ```
   if(notificationOK){
   ```

```
    sendAck(ucpService, connectionId, deliverSMPDU);
}else{
    sendNack(ucpService, connectionId,deliverSMPDU, UcpConstants.ERROR_CODE_
SYNTAX_ERROR);
            }
```

10 Using Service Interceptors to Manipulate Requests

This chapter provides a high-level overview of service interceptors (interceptors) and describes both the standard interceptors in Oracle Communications Services Gatekeeper and the process of developing your own custom interceptors and deploying them in Services Gatekeeper.

Understanding Service Interceptors in Services Gatekeeper

Service interceptors provide a mechanism to intercept and manipulate a request flowing through any arbitrary communication service in Services Gatekeeper. (See "Understanding Communication Service Components" for a description of communication services.) Additionally, service interceptors supply an easy way to modify the flow for a request and simplify the routing mechanism for plug-ins associated with a request.

Often, interceptors may make a decision to permit, deny, or stay neutral to a particular request. Some typical use cases for service interceptors are to:

- Deny a request if the user does not subscribe to a particular service in the application layer.
- Deny a request if the personal identification number is not valid.
- Verify that a request’s parameters are valid.
- Perform argument manipulation (such as aliasing).

Each interceptor in Services Gatekeeper 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**.

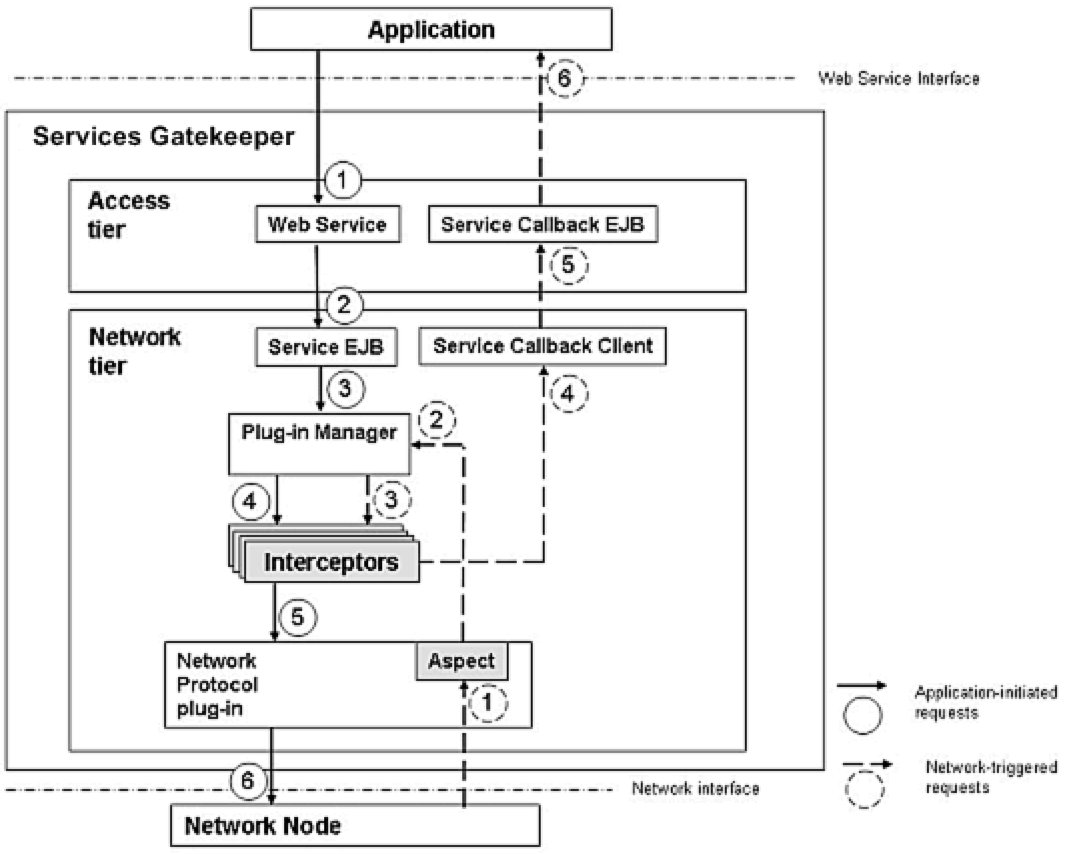
For example, the **EnforceBlacklistedMethodFromSLA** interceptor is identified by **com.bea.wlcp.wlng.interceptor.EnforceBlacklistedMethodFromSLA**. (See the entry for **EnforceBlacklistedMethodFromSLA** in Example 10–1.)

A set of standard interceptors are provided with Services Gatekeeper. Some interceptors are required, while others provide extra functionality. In addition, Services Gatekeeper enables you to develop and deploy custom interceptors. The invocation order of the interceptors currently active in Services Gatekeeper is defined in an XML-based configuration file, described later in this chapter.

Understanding How Requests are Triggered

Figure 10–1 illustrates where interceptors are triggered, both for application-initiated and network-triggered requests.

Figure 10–1 Interceptors and Request Flow



Application-initiated requests proceed south in the following order:

1. The requests starts in the web service (access tier).
2. The web services converts the request to the Service EJB in the network tier.
3. The Service EJB sends the request to the plug-in manager.
4. The plug-in manager sends it to the interceptor stack.
5. From the interceptor stack the request is sent do the network protocol plug-in.
6. The network protocol sends the request to the network node.

Network triggered requests proceed north in the following order:

1. The request starts in the network tier.
2. From the network tier, the request goes to the network protocol plug-in.
3. From the network plug-in it goes to the interceptor stack.
4. From the interceptor stack it goes to the service callback client.
5. From the service callback client it goes to the service callback EJB (access tier).

6. Finally the request arrives at the application.

Understanding How the Plug-in Manager Works with Interceptors

As Figure 10–1 shows, the Plug-in Manager is responsible for calling the first interceptor in the chain of interceptors as defined in the interceptor configuration file, described later in this chapter.

For application-initiated requests, the Plug-in Manager is called automatically by the service Enterprise Java Beans (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. For more information on Aspect, see "About Aspects and Annotations".

The interceptor chain is invoked at the point-cut that is a Java representation of the application-facing interface. 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.

Request Context Data Used to Handle Request Flow

Interceptors in Services Gatekeeper have access to context data associated with each request and use that data to arrive at the appropriate decisions to forward, return or abort the request.

The actual data that is available to an interceptor depends on the context of the request. In general, the application-facing interface defines the data that is available. This data includes the following:

- 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

  For information on **RequestContext**, see "Interface: RequestContext".
- 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

  For information on **RequestInfo**, see "Class: RequestInfo".
- A list of plug-ins that matches the specified **RequestInfo**

- The interception point that indicates whether the request is network-triggered or application-initiated.

Example 10–6 shows the method used to retrieve data located in **RequestContext**.

Data Available for Modification

Custom interceptors that you develop can be designed to access and modify certain elements of the data in **RequestContext**.

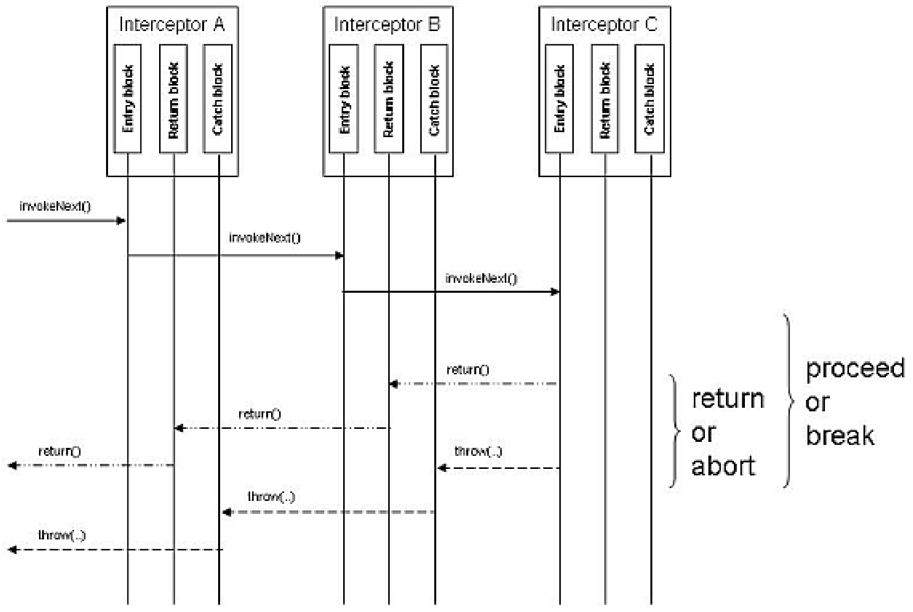
Interceptors can set the following data in a request:

- In the **RequestContext**:
  - Session ID
  - Transaction ID
- A list of plug-ins that matches the specified **RequestInfo**. For information on **RequestInfo**, see "Class: RequestInfo".
- Arguments to the method targeted by the request

Specifying a Destination for the Request

Each interceptor is responsible for deciding whether to proceed with the request flow (by forwarding the request down the chain of interceptors) or to break the flow. The interceptor may break the request flow in one of two ways, either by returning the request or by aborting it.

Figure 10–2 Decisions and the Interceptor Chain



Proceeding with the Request Flow

The decision to proceed with the request flow translates to continuing down the invocation chain by calling the next interceptor in the chain.

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 points in the flow, the return path traverses the interceptors that were used in the calling path. Doing so makes it possible for interceptors to manipulate the request on its return path and ultimately return it to the originator of the request, the application or the network node.

Returning the Request

The decision to return a request may be arrived at because the needs of the request may have been fulfilled and, therefore, there maybe no need to call the plug-in or the application. The remaining and final step is simply to return the request.

In such a scenario, the request is rolled back through the previous interceptors using a regular return statement. Doing so makes it possible for the previous interceptors to manipulate the request in the rollback path which ends with the originator of the request, the application or the network node.

Aborting the Request

The decision to abort a request is arrived at when there is a violation. For example, a parameter in the request is out of bounds, or certain usage policies have been violated by this request. Such events lead to a **PluginDenyException** being thrown.

When the decision is made to abort a request, the interceptor breaks the chain for that request. The request is rolled back through each interceptor's exception catch-block rather than being returned 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. For such a scenario, interceptors should use **com.bea.wlcp.wlng.api.plugin.DenyPluginException**.
- For network-triggered requests, it is the responsibility of the plug-in to act on the thrown exception.

Invoking the Next Service Interceptor to Handle the Request

Each interceptor is responsible for calling the next interceptor in the chain. 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 returns 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 returns 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 actual telephone number of the subscriber.

Last Service Interceptor in the Chain

For application-initiated requests, the last interceptor in the chain is responsible for calling the plug-in.

For network-triggered requests, the last interceptor in the chain is responsible for calling the callback service EJB, which calls the application. In Example 10–2, the example interceptor configuration file lists **InvokeApplication** as the last entry for the tag called `<position name="MO_NORTH">`.

Understanding the Standard Interceptors

Standard interceptors are service interceptors that are provided by Services Gatekeeper.

WARNING: Do not change the order of the standard interceptors.

The standard interceptors in Services Gatekeeper can be:

- Enabled: The interceptors that are enabled in Services Gatekeeper are listed in the **config.xml** interface configuration file provided with the installation. See "Config.xml File".
- Disabled: A few interceptors are not yet enabled in the default set of interceptors enabled in Services Gatekeeper. If necessary, you can enable these interceptors by updating **config.xml**. See the steps in "Updating the Config.xml File".
- Internal: Any interceptor with a **$** in its name is internal to Services Gatekeeper. Such an interceptor should not be used, changed or deleted.

Table 10–1 provides a description of the standard interceptors that are available in Services Gatekeeper and lists any dependencies enforced by Services Gatekeeper.

Table 10–1 Standard Interceptors Provided by Services Gatekeeper

Interceptor	Description
CreatePluginList	Enabled at installation time. Creates a list of plug-ins that are capable of handling the given request. Populates the **RequestInfo** object. See "Class: RequestInfo".
EdrParameterInterceptor	Maps a request context parameter to an EDR attribute. You configure this using the **loadEdrParameterXml** operation to the **EdrServiceMBean**. You can list the parameters using the **getserviceinfo** operation to **PluginManagerMBean**.
EnforceApplicationBudget	Enabled at installation time. Enforces the application budget.
EnforceApplicationSla	Enabled at installation time. Enforces the application SLA.
EnforceApplicationState	Enabled at installation time. Enforces the application state. Verifies that the application with which the request is related has established a session with Services Gatekeeper.
EnforceNodeBudget	Enabled at installation time. Enforces budgets related to the global and service provider node SLAs.

Table 10–1 (Cont.) Standard Interceptors Provided by Services Gatekeeper

Interceptor	Description
EnforceServiceProviderBudget	Enabled at installation time. Enforces the budget defined in for a service provider group. See the discussion on service provider groups in *Services Gatekeeper Portal Developer's Guide*.
EnforceServiceProviderSla	Enabled at installation time. Enforces the budget defined in the service provider group SLA. See the discussion on service provider service level agreements in *Services Gatekeeper Portal Developer's Guide*.
EnforceSlaBlacklistedMethods	Enabled at installation time. Enforces the method blacklist as specified in the service provider group and application group SLAs. Is related to the SLA `<blacklistedMethod>` element in the `<methodAccess>` element. See the discussion on service provider groups and service level agreements in *Services Gatekeeper Portal Developer's Guide*.
EnforceSlaFilterResult	Enabled at installation time. Enforces the filter list as specified in the service provider group and application group SLAs. See the discussion on service provider groups and service level agreements in *Services Gatekeeper Portal Developer's Guide*.
EnforceSlaMethodParameters	Enabled at installation time. Enforces method parameters as specified in the service provider group and application group SLAs. See the discussion on service provider groups and service level agreements in *Services Gatekeeper Portal Developer's Guide*.
EnforceSlaRequestContextParams	Enabled at installation time. Enforces the request context parameter list as specified in the service provider group and application group SLAs. See the discussion on service provider groups and service level agreements in *Services Gatekeeper Portal Developer's Guide*.
FilterPluginListUsingCustomMatch	Enabled at installation time. Invokes the custom match method of each plug-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.
InjectXParametersFromRequestContext	Enabled at installation time. Takes tunnelled parameters from the RequestContext and puts them in the the SOAP header of either a request to an application or a response to a request from an application.
InvokeApplication	Enabled at installation time. Invokes the Application via the service callback EJB. This should be the last interceptor for an network-triggered (mobile originated) request.
InvokeNetworkNode	Enabled at installation time. Invokes the Network Node via the service callback EJB. This is the only interceptor for an network-triggered (mobile terminated) request.
InvokePlugin	Enabled at installation time. Invokes the plug-in(s). This should be the last interceptor for an application-initiated (mobile terminated) request.

Table 10–1 (Cont.) Standard Interceptors Provided by Services Gatekeeper

Interceptor	Description
InvokeServiceCorrelation	Enabled at installation time. Invokes the service correlation feature. See "Correlating Services".
NativeSMPPAddressRouting	Not enabled, by default If enabled, ensures that, for networks having multiple SMSCs, SMS messages sent to the same address will be sent through the same plug-in instance.
NativeUCPAddressRouting	Not enabled, by default If enabled, ensures that messages sent to the same address are sent to the same SMSC.
OutboundProxyEnabler	Allows proxy parameters to be set in SLAs, or request context to be used when making application callbacks.
RemoveInactivePlugin	Enabled at installation time. Removes any plug-in that is not active from the current plug-in list.
RemoveInvalidRoute	Enabled at installation time. Enforces the plug-in routing logic.
RemoveOptional	Enabled at installation time. 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.
RequestMonitor	Allows requestcounters to be retrieved by WebLogic Diagnostic Framework (WLDF).
RetryPlugin	Enabled at installation time. Performs retries of request. See "Retry Functionality for plug-ins".
RoundRobinPluginList	Enabled at installation time. 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.
ValidateRequestUsingRequestFactory	Enabled at installation time. Validates the request using the **RequestFactory** corresponding to the type of plug-in the request is intended for. For a description of **RequestFactory**, see "Class: RequestFactory".

Locating the Standard Interceptors

All the standard interceptors can be found packaged in *Middleware_home*/**ocsg/applications/interceptors.ear.** (See Table 10–3.)

Retry Functionality for plug-ins

When a **RetryPluginException** exception is thrown during the handling of a request, a retry is attempted among the plug-ins that were chosen based on the data provided in the request. Retries are performed among the plug-ins in the same Services Gatekeeper instance only.

When a plug-in throws a **RetryPluginException**, the **RetryPlugin** interceptor is triggered. The **RetryPlugin** interceptor captures the **RetryPluginException**, 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 in Table 10–2.

Table 10–2 Retry Plug-in Interceptor Scenarios

Objects with which the RequestInfo objects in the RequestContext are associated	Action(s) Taken by RetryPlugin interceptor
PluginHolder objects that are marked as optional	Remove the failed **RequestInfo** from **RequestContext** and invoke the next interceptor in the chain.
PluginHolder objects that are marked as required	Treat the request itself as failed. No retry is performed, and an exception is thrown.

Note that the **Subscriber Profile/LDAPv3** is the only (standard) plug-in that throws the **RetryPluginException**.

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.

Interceptors.ear File

The **interceptors.ear** file is located at *Middleware_home*/**ocsg/applications**.

File Contents

Table 10–3 describes the contents of the top-level of the multi-level folder in the **interceptors.ear** file.

Table 10–3 Contents of interceptor.ear File

Folder Name/File Name	Content
APP-INF	A multi-level folder which contains information about the applications. ■ **classes**: A multi-level folder where the sub-folder **/APP-INF/classes/com/bea/wlcp/wlng/interceptor** contains the standard interceptor classes. ■ **config.xml**: Interceptor configuration file. See "Config.xml File". ■ **config.xsd**: Schema for **config.xml**.
dummy.war	Empty WAR file. Present in order to deploy the interceptors. Do not remove or change this file.
META-INF	This folder contains the following files: ■ **application.xml**: Deployment descriptor. ■ **MANIFEST.MF**: Manifest file for the interceptor infrastructure. ■ **weblogic-application.xml**: WebLogic extensions to application.xml. Do not edit or remove the contents of **META-INF**.

Table 10–3 (Cont.) Contents of interceptor.ear File

Folder Name/File Name	Content
WEB-INF	For internal use. Do not edit or remove its contents.

Maintaining Interceptor Data Integrity

If you deploy a new interceptor, you need to update the **interceptors.ear** file to support the functionality for your custom interceptor.

At all times when you do so, be sure to maintain the general structure shown in Table 10–3. The following data must not be changed:

- Data listed in Table 10–3 as not be changed or removed.
- **/APP-INF/classes/com/bea/wlcp/wlng/interceptor/deploy**: (Infrastructure for the interceptor functionality.)
- **/APP-INF/classes/com/bea/wlcp/wlng/interceptor/util**

Location for All Standard Interceptor Classes

All standard interceptors provided with Services Gatekeeper (and described in Table 10–1) are located in **/APP-INF/classes/com/bea/wlcp/wlng/interceptor/**.

Config.xml File

The **config.xml** located at **/interceptors/APP-INF/classes** displays the standard interceptors that are currently enabled in Services Gatekeeper.

To access this file, expand the compressed *Middleware_home***/ocsg/applications/interceptors.ear** to a folder at a suitable location. It would be good practice to create and store a backup of the original **config.xml** file to keep the base version that was provided at installation time.

Elements in Config.xml

Table 10–4 describes the structure of **config.xml**.

Table 10–4 Description of Interceptor Configuration File

Element	Description
`<interceptor-config>`	Main element. Contains zero or more `<position>` elements.
`<position>`	The `<position>` element separates the interceptors according to the following four attributes. ■ `MT_NORTH`: This position name indicates that all `<interceptor>` elements encapsulated by this element are valid for application-initiated (mobile terminated) requests. ■ `MO_NORTH`: This position name indicates that all `<interceptor>` elements encapsulated by this element are valid for network-triggered (mobile originated) requests. ■ `MO_SOUTH`: Currently, the interceptors listed for `MO_SOUTH` attribute are internal to Services Gatekeeper. They should not be altered in anyway. ■ `MT_SOUTH`: Currently, the interceptors listed for `MO_SOUTH` attribute are internal to Services Gatekeeper. They should not be altered in anyway.

Table 10–4 (Cont.) Description of Interceptor Configuration File

Element	Description
`<interceptor>`	Has the following attributes: ▪ `class`: The class attribute identifies the class for the interceptor implementation. ▪ `index`: The index attribute indicates the invocation order relative to other interceptors within the same `<position>` element. The order is ascending. The index value must be unique within the same `<position>` element.

Standard Interceptors in the MT_NORTH Section

The standard interceptors found in the **MT_NORTH** section of the default configuration file are listed here for your convenience. Note that `InvokePlugin` (with the fully classified name **com.bea.wlcp.wlng.interceptor.InvokePlugin**) is the last interceptor in this section.

Example 10–1 MT_NORTH Position Interceptors

```
 <position name="MT_NORTH">
    <interceptor class="oracle.ocsg.interceptor.RequestMonitor" index="25"/>
    <!-- #49 is reserved for OAuth -->
    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePluginList" index="50"/>
    <!-- #51, #52 are reserved for OAuth -->

    <interceptor class="oracle.ocsg.interceptor.WebSocketLogin" index="58"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.ValidateRequestUsingRequestFactory"
index="100"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceApplicationState" index="150"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceServiceProviderSla" index = "200"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceApplicationSla" index = "250"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePolicyData" index = "300"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaMethodParameters" index = "350"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaBlacklistedMethods" index = "400"/>
    <!-- duplicated with index 350
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaMethodParameters" index = "450"/>
-->
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaRequestContextParams" index = "500"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaFilterResult" index = "550"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveInactivePlugin" index="600"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveInvalidRoute" index="650"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.FilterPluginListUsingCustomMatch"
index="700"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveOptional" index="750"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.RoundRobinPluginList" index="800"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.InvokeServiceCorrelation" index="850"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.RetryPlugin" index="900"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceServiceProviderBudget" index = "950"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceApplicationBudget" index = "1000"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceNodeBudget" index="1050"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.EnforceSubscriberBudget" index="1100"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.InjectXParametersFromRequestContext"
index="1150"/>
    <interceptor class="oracle.ocsg.interceptor.EdrParameterInterceptor" index="1200"/>
     <interceptor class="oracle.ocsg.interceptor.InvokePlugin" index="99999"/>
```

```
  </position>
```

Standard Interceptors in the MO_NORTH Section

The standard interceptors found in the MO_NORTH section of the default configuration file are listed here for your convenience. Note that `InvokeApplication` with the fully classified name **com.bea.wlcp.wlng.interceptor.InvokeApplication** is the last interceptor in this section.

Example 10–2 MO_NORTH Position Interceptors

```
 <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="oracle.ocsg.interceptor.EnforceServiceProviderSla" index = "250"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceApplicationSla" index = "300"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePolicyData" index="400"/>

    <interceptor class="oracle.ocsg.interceptor.EnforceSlaMethodParameters" index="500"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceSlaBlacklistedMethods" index = "600"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceServiceProviderBudget" index = "700"/>
    <interceptor class="oracle.ocsg.interceptor.EnforceApplicationBudget" index = "750"/>

    <interceptor class="com.bea.wlcp.wlng.interceptor.InjectXParametersFromRequestContext"
index="800"/>
    <interceptor class="oracle.ocsg.interceptor.OutboundProxyEnabler" index="900"/>
    <interceptor class="com.bea.wlcp.wlng.interceptor.InvokeApplication" index="99999"/>
  </position>
```

Standard Interceptors in the MO_SOUTH Section

The standard interceptors found in the MO_SOUTH position of the default configuration file are listed here for your convenience. The last interceptor in this section is `InvokeApplication` with the fully classified name **com.bea.wlcp.wlng.interceptor.InvokeApplication**.

Currently, the interceptors listed for MO_SOUTH are internal to Services Gatekeeper. They should not be altered in anyway.

Example 10–3 MO_SOUTH Position Interceptors

```
 <position name="MO_SOUTH">
  <interceptor class="com.bea.wlcp.wlng.interceptor.CreatePluginList" index="100" />
  <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveInactivePlugin" index="200" />
  <interceptor class="com.bea.wlcp.wlng.interceptor.FilterPluginListUsingCustomMatch" index="300"
/>
  <interceptor class="com.bea.wlcp.wlng.interceptor.RemoveOptional" index="400" />
  <interceptor class="com.bea.wlcp.wlng.interceptor.RoundRobinPluginList" index="500" />
  <interceptor class="com.bea.wlcp.wlng.interceptor.RetryPlugin" index="600" />
  <interceptor class="com.bea.wlcp.wlng.interceptor.InvokePlugin" index="700" />
  </position>
```

Standard Interceptors in the MT_SOUTH Section

Only one standard interceptor is found in the **MT_SOUTH** position of the default configuration file and is listed here for your convenience.

Currently, the interceptors listed for **MT_SOUTH** are internal to Services Gatekeeper. They should not be altered in anyway.

Example 10–4 MT_SOUTH Position Interceptor

```
<position name="MT_SOUTH">
    <interceptor class="com.bea.wlcp.wlng.interceptor.InvokeNetworkNode" index="99999"/>
 </position>
```

Creating and Using Custom Interceptors

This section describes how you can develop and deploy custom interceptors to modify the handling of requests using the existing Services Gatekeeper software. The custom interceptors are developed using the Introspection method.

Understanding Custom Interceptors

Custom interceptors, as their name suggests, are tailored to the individual needs of each application. This section describes the points to keep in mind when you develop custom interceptors.

How to Provide Your Custom Interceptors

When you create a custom interceptor, you need to package it in an EAR file to enable Services Gatekeeper to use it.

You can set up your custom interceptor in one of two ways:

- Develop and deploy your custom interceptor in the common **/applications/interceptors.ear** file. This method is described in "Using the Default EAR File to Add a Custom Interceptor".
- Develop and deploy your custom interceptor in a separate EAR file. This method is described in "Using a Custom EAR File to Add a Custom Interceptor".

For each method, the description focuses on the creation of a single custom interceptor. In practice you can create as many interceptors as you require.

Required Packages, Interfaces and Methods

Determine and provide the required and relevant Java (or other) packages for your custom interceptor.

For example, all the classes necessary for *SampleInterceptor* in Example 10–5 are available in the package:

com.bea.wlcp.wlng.api.interceptor located in *Middleware_home*/**ocsg_pds/lib/api/wlng.jar**.

You can find all publicly available classes in the "All Classes" section of the *Services Gatekeeper Java API Reference*.

Creating a Backup

It would be good practice to create the necessary backups of the current configuration before you embark on any changes to the current setup.

For example, you should create a backup of the current version of the **/applications/interceptors.ear** file located at *Middleware_home*/**ocsg/applications** and store it in a desired location.

On Customer Interceptor Implementation

Note the following points about a custom interceptor:

- Application: The interceptor that you create can be placed in any type of JavaEE or WebLogic application.
- Interface to implement: Your custom interceptor must implement the interface **com.bea.wlcp.wlng.api.interceptor.Interceptor**.
- Override: Your custom interceptor must override the **invoke** method of that interface. See Example 10–5 and Example 10–6.
- Actions: The logic in your custom interceptor depends on what it needs to do:
  - Some interceptors contain logic that results in a decision that may affect the request flow. (See Example 10–5). For decisions, see "Specifying a Destination for the Request".
  - Other interceptors serve additional functions. For example, the custom interceptor *ExtractXParamExample* in Example 10–6 retrieves the context data in the **RequestContext** object for a specific type of request.
- Registration: A custom interceptor can be registered or unregistered when the status of that application changes. The information used to register a custom interceptor must be synchronized with the existing data for interceptors in Services Gatekeeper.
- Index value: The index value used to register the interceptor should be unique with respect to the entries in the **config.xml** file of **/applications/interceptors.ear** file and other custom interceptors.

  A collision will occur if more than one interceptor is registered with the same index value, In such a situation, only the last interceptor to register at the index will be executed. The other interceptor(s) with that index value will be overwritten.
- Positioning with respect to **RetryPlugin**: Where you position your custom interceptor with respect to **RetryPlugin** will determine whether your custom interceptor is invoked once or more for a request:
  - Before **RetryPlugin**: If you add your custom interceptor before the **RetryPlugin** interceptor, your custom interceptor will be triggered only once for the request.
  - After **RetryPlugin**: If you add your custom interceptor after **RetryPlugin**, your custom interceptor will be triggered once for every plug-in that is attempted.

    Note that the **Subscriber Profile/LDAPv3** is the only (standard) plug-in which throws the **RetryPluginException**. If you have a custom plug-in which throws **RetryPluginException**, place your custom interceptor after **RetryPlugin** interceptor if your custom interceptor should be invoked for each "tried" plug-in instance.

    For more information, see "Retry Functionality for plug-ins".
- Request Flow: A custom interceptor is responsible for invoking the *next* interceptor in the chain by using the `invokeNext` method. See Example 10–5 and Example 10–6.
- Thread safety: It must be thread-safe.
- Debugging: Log statements at the debug level enable you to debug your custom interceptor. These statements will be needed to turn on the logging mechanism when you wish to debug your code.

Testing the Custom Interceptor

The Platform Test Environment (PTE) can be used to test your custom interceptor before you use it in a production environment. For more information, see *Services Gatekeeper Platform Test Environment User's Guide*.

Understanding the Example Interceptors

Two example interceptors are provided for your reference. In the first example, the custom interceptor makes some decisions, while the second shows how context data can be extracted from the **RequestContext** object.

For information on **RequestContext**, see "Interface: RequestContext".

You can find additional interceptor examples on the Oracle Learning Library (OLL) website at:

`https://apex.oracle.com/pls/apex/f?p=44785:24:0::NO:24:P24_CONTENT_ID,P24_PREV_PAGE:9578,29`

General Example

This example shows the structure of a simple interceptor designed to make some decisions on a request. The code will require logic to determine the value for `decision` and for the `ReturnValue` object.

Example 10–5 General 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;
    }
  }
}
```

Interceptor that Extracts Context Data from RequestContext

The following example interceptor extracts some of the arguments present in the **RequestContext** data object. For more on the data in **RequestContext** that can be modified by a custom interceptor, see "Data Available for Modification".

As the code shows, the interceptor intercepts and retrieves the **RequestContext** data associated with **SendSms** requests only.

Example 10–6 Custom Interceptor to Extract Data from RequestContext Object

```
package com.bea.wlcp.wlng.interceptor;
```

```
// Provide all the Required Interfaces/Classes. This example imports:
// Java API for Method;
// Context, the parameter passed to Invoke;
// the Interface Interceptor;
// the package which contains the required Plugin;
// the package which contains the required RequestContext;

import java.lang.reflect.Method;
import org.apache.log4j.Logger;
import com.bea.wlcp.wlng.api.interceptor.Context;
import com.bea.wlcp.wlng.api.interceptor.Interceptor;
import com.bea.wlcp.wlng.api.plugin.Plugin;
import com.bea.wlcp.wlng.api.plugin.context.RequestContext;


// Example Interceptor implements Interface Interceptor
public class ExtractXParamExample implements Interceptor {

// Create a log object for ExtractXParamExample
       private Logger logger = Logger.getLogger(ExtractXParamExample.class);

// Define the tag and the value that must be extracted.
       public static final String TLV_OPTIONAL_INT_PARAM_TAGS =
"smpp_optional_int_tlv_param_tags";
       public static final String TLV_OPTIONAL_INT_PARAM_VALUES =
"smpp_optional_int_tlv_param_values";


// Method Override
       @Override


// Implement the invoke method of interceptor inteface
       public Object invoke(Context ctx) throws Exception {
              Object[] args = ctx.getArguments();

// Retrieve a class object corresponding to the Plugin type
              Class<? extends Plugin> pluginType = ctx.getType();

// Retrieve the name of the  method used
              Method calledMethod = ctx.getMethod();

// Check to see if the method is "SendSms" with the appropriate plugin
// If it is not what we’re looking for, do nothing
// If it is Send Sms,
// Call extractTLVXParameters to retrieve contents of RequestContext object

             if (pluginType.getName().equals(
                      "com.bea.wlcp.wlng.px21.plugin.SendSmsPlugin")&&
                     calledMethod.getName().equals("sendSms")) {
                            extractTLVXParameters(ctx, args[0]);
                    }

// All done. invokeNext must be called here.
    return ctx.invokeNext(this);
       }


       private void extractTLVXParameters(Context ctx, Object arg) {
             RequestContext rctx = ctx.getRequestContext();
             logger.info("Extracted XParams for: " +
             TLV_OPTIONAL_INT_PARAM_TAGS +"::"+
             rctx.getXParam(TLV_OPTIONAL_INT_PARAM_TAGS)+
             " and: " + TLV_OPTIONAL_INT_PARAM_VALUES+"::"
             +rctx.getXParam(TLV_OPTIONAL_INT_PARAM_VALUES));
       }

}
```

Interceptor that Functions as a Black List for SMSs

This pseudo code illustrates how to check whether an SMS from the class `org.csapi.schema.parlayx.sms.send.v2_2.local.SendSms` contains the word "bomb" and if so, rejects it.

Example 10–7 Interceptor that Rejects all SMSs with the Word "Bomb"

```
// keyword blacklist
if (message.toUpperCase().indexOf("BOMB") > -1) {
     throw new Exception("Blacklisted keyword found in message");
}
```

Interceptor that Replaces a Word with a Variable String in an SMS

This pseudo code searches each SMS from `org.csapi.schema.parlayx.sms.send.v2_2.local.SendSms` for the word "weather" and replaces it with the string defined for the variable `WX_MSG`.

Example 10–8 Replaces a the word "Weather" With a Variable String

```
 // short codes
Class c = ctx.getArguments()[0];
if ((c.getName().equals("org.csapi.schema.parlayx.sms.send.v2_2.local.SendSms"))
&& (message.indexOf("WEATHER") > -1)) {

     System.out.println("changing shortcode: " + message);
     Method mm = c.getMethod("setMessage", new String().getClass());
     mm.invoke(argz[i], WX_MSG);
}
```

Using the Default EAR File to Add a Custom Interceptor

This section describes how you can develop custom interceptors for use with the default **interceptors.ear** file. See "Interceptors.ear File" for information about this file.

Developing the Custom Interceptor for Deployment

Use the Platform Development Studio wizard to create custom service interceptors. See "Creating Extensions with Platform Development Studio Wizard" for information on using the PDS Wizard.

Artifacts for a Custom Interceptor Module

When you use the wizard to create the skeleton of an interceptor module, the wizard generates the following artifacts:

- **build.xml**:

  This is the build file used to build all the interceptor modules and to package them into a single EAR file for deployment.

- **build.properties**

  This is the properties file required by the Apache Ant process to build the module.

- **common.xml**

  This file defines the common properties used by the module, such as environment variables, WebLogic library path, and so on.

- **CustomizedApplicationLifecycleListener.java**

  This is an implementation of the WebLogic **ApplicationLifecycleListener** used to manage the module life cycle.

- **InterceptorXXX.java**

  This is your interceptor implementation, where *XXX* is the name that you assign to the interceptor in the wizard.

Generating a Custom Interceptor Module

See "Generating an Interceptor Module" for information on generating a custom interceptor using the custom interceptor PDS wizard.

Deploying Custom Interceptors

Use the following procedure to deploy your custom interceptor:

1. Expand the **/applications/interceptors.ear** file and review its contents to determine the location for your new interceptor. If necessary, create a backup file at a desired location.
2. Create the necessary customer interceptor class, (for example, *MyCustomEnforceThis.class*). For details, see "On Customer Interceptor Implementation".
3. Update the **config.xml** file. Position the interceptor in the appropriate `<position>` section(s). See "Updating the Config.xml File".
4. Verify the changes to the invocation order in **config.xml**. See "Updating the Config.xml File".
5. Repackage the **interceptors.ear** file. See "Rebuilding the Interceptors.ear File".

Updating the Config.xml File

This section describes how to update the **config.xml** file to provide the desired invocation order for the interceptors. See "Config.xml File" for a description of the file.

Creating a Backup of the Current Config.xml

To change the order in the interceptor configuration file, always use the current **interceptors.ear** file deployed on the Administration Server.

As a general rule, before you proceed with any changes to **config.xml**, be sure to back up that file in a secure location and using a different name, for example, **config.xml_***backup_as_of_May252013*.

Adding the Custom Interceptor to the Current Chain

To add your custom interceptor to the interceptor chain:

1. Access the **config.xml** file.
2. Add the entry/entries for the custom interceptor(s).

   For every new custom interceptor, a `<interceptor>` element 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 should be present. Ensure that:

   - The entry for the interceptor is placed in the required `<Position name=...>` block(s).
   - The `<class=...>` attribute names the required class.
   - The `<index=...>` attribute contains a value that is appropriate and unique to the specific section.

   For example, if the interceptor main class is **com.***acompany***.interceptor.***DoStuff*, and the chosen index value is *1150*, the corresponding entry in **/APP-INF/classes/config.xml** would be

   ```
   <interceptor class="com.acompany.interceptor.DoStuff" index="1150"/>
   ```

3. Save the **config.xml** file.

Rearranging the Invocation Order

To rearrange the invocation order for an interceptor chain:

1. Access the **config.xml** file.
2. Edit the **config.xml** file and change the `index` attribute for the appropriate `<interceptor>` element if necessary. Ensure that the new value is appropriate and unique within that `<position>` element.
3. Save the **config.xml**.

Excluding an Interceptor from the chain

To exclude an interceptor element from an interceptor chain:

1. Access the **config.xml** file.
2. Edit the **config.xml** file and comment out the specific `<interceptor>` element(s).
3. Save the **config.xml**.

Rebuilding the Interceptors.ear File

To re-build the common **interceptors.ear** file:

1. Build the class file for the custom interceptor module using the artifacts generated by the PDS wizard.
2. Place the class file for the interceptor in the appropriate location where the other standard interceptors are located.
   - For example, an installation maintains the default settings provided by Services Gatekeeper at installation time. In such a scenario, the main class for the interceptor would be **com.bea.wlcp.wlng.interceptor**. All the standard interceptors would be located in **/APP-INF/classes/**.
   - If for example, the main class for your custom interceptor is **com.***mycompany***.interceptor.***DoStuff*, place *DoStuff***.class** in **/APP-INF/classes/com/***mycompany***/interceptor**.

3. Repackage the EAR file, making sure that you maintain the original structure of common **interceptors.ear** file. See "Maintaining Interceptor Data Integrity".

Re-deploying Common Interceptors. ear File

Use standard WebLogic procedures to redeploy the application **interceptor.ear** to all servers in the network tier cluster from the Administration server. For more information on re-deploying applications on Oracle WebLogic Server, see the description in *Developing Applications with WebLogic Server*.

Using a Custom EAR File to Add a Custom Interceptor

This section describes how you can provide a custom EAR file for use with your custom interceptor.

Points to Note

If you use a custom EAR file for your custom interceptor:

- A custom interceptor meant for use with a custom EAR file should not be included in the **config.xml** file in the common **interceptors.ear** file.
- When providing index values for your custom interceptor, maintain the order for the indexes used in the current **config.xml** in the common **interceptors.ear** file.
- Do not modify **config.xml**, the standard configuration file in Services Gatekeeper (and located in **interceptors.ear**). Services Gatekeeper will continue to use the default interceptors in that **config.xml** file.
- If, currently, there is no custom EAR file:
  – An alternate listener class must be provided to take over the registration process for your custom interceptor at server startup and restart events. See "Creating a Custom Listener".
  – A custom EAR file must be built. See "Building a Custom EAR File".
- If a custom EAR file exists, update that EAR file appropriately. This is described under "Updating an Existing Custom EAR to Add Custom Interceptors".
- The custom interceptors can be placed in a JavaEE or WebLogic application. Ensure that the registration process handles these interceptors appropriately.

Steps to Build a Custom EAR for Use with a Custom Interceptor

To provide a separate EAR file for your custom interceptor:

1. Create the necessary customer interceptor class, (for example, *MyCustomEnforceThis.class*). See "On Customer Interceptor Implementation".
2. Create a listener to register the custom interceptor. (For example, *MyCustomListener*.**class**).
3. Set up the registration process to handle the registration of the *MyCustomEnforceThis.class* custom interceptor.
4. Build a Custom EAR file (for example, *MyCustomEar.ear*) which will be the active EAR in your deployment.

Information Needed to Register Custom Interceptors

The following information is necessary to register your custom interceptor and enable it in Services Gatekeeper:

- The fully qualified name for your custom interceptor
- Its position in the request flow associated with the `<position>` element seen in **config.xml**, (for example, `MT_NORTH` and/or `MT_SOUTH`)
- The exact point in the invocation order, specified as the value for the `index` attribute. See Table 10–4.

Place this information (for example, as an XML file) in the custom ear to be parsed and used for the registration.

Synchronizing with Invocation Order in Config.xml

Check to make sure that the index values used to set up the invocation point(s) for your custom interceptors maintain the general order for the indexes currently used in **config.xml** in the common **interceptors.ear** file.

Creating a Custom Listener

By default, Services Gatekeeper employs **InterceptorListener** to automatically register the standard interceptors in the common **interceptors.ear** file. It does this when the Administration Server starts (or when it restarts after a status change).

If this is the first time a custom EAR will be used for a custom interceptor, you need to create a custom listener for your custom interceptor. The custom listener that you create must do the work done by the standard **InterceptorListener** for the standard interceptors.

Implementing ApplicationLifecycleListener

In order to create such a custom listener, (for example, *MyCustomListener*.**class**), implement **weblogic.application.ApplicationLifecycleListener**. For more information on **ApplicationLifecycleListener**, see:

`http://download.oracle.com/docs/cd/E11035_01/wls100/javadocs/weblogic/application/ApplicationLifecycleListener.html`

Note that, the **ApplicationLifecycleListener** interface is only used with reference to the application state and not with reference to the state of the WebLogic server.

Ensure that, *MyCustomListener*.**class** is called whenever there is a change in *myCustomEar*.**ear** application status. When it is called, *MyCustomListener*.**class** must complete the registration process for your custom interceptors.

The Registration Process

You can select to hard-code the information necessary to register your interceptors or use the **InterceptorManager** interface.

Hard-coding the Information

If you have a limited set of custom interceptors, and their behavior may not often be changed, you can hard-code this information in *MyCustomListener*.**class**.

Providing a Data File for Registration

If you plan to use the **InterceptorManager** interface, create a data file that will contain the information necessary to register the custom interceptor as described in

"Information Needed to Register Custom Interceptors"). This data (the position, and index information for each custom interceptor) will be used by *MyCustomListener***.class** in the `register` method described below.

Registering (and unregistering) Your Custom Interceptors

If you are not hard-coding the registration information, ensure that you have the external data file necessary to parse and register your interceptors.

Use the `register` or `unregister` method in the **InterceptorManager** interface in your custom listener (*MyCustomListener*) to register/unregister your custom interceptor.

Note that:

- Retrieve the using the **getInstance** method in **com.bea.wlcp.wlng.api.interceptor.InterceptorManagerFactory**. They are:
- The **InterceptorManager** interface handles the registration, unregistration and invocation of the interceptors. Use the following methods in this interface:
  - `register`: The `register` method should be called in the **postStart** callback when the application has been started.
  - `unregister`: The `unregister` method should be called in the **preStop** callback when the application is being stopped.
  - `update`: The `update` method must be invoked immediately following `register` or `unregister` to activate the changes caused by each method.

Note that, unless `update` is called, the changes will not take effect.

Example

Example 10–9 shows an example of how to register an interceptor manually.

Example 10–9 Registering an interceptor

```
// Get Interceptor manager
InterceptorManager im = InterceptorManagerFactory.getInstance();

// Register the custom MyCustomEnforceThis interceptor
im.register(MyCustomEnforceThis, InterceptionPoint.MT_NORTH.MT_NORTH, myIndex);

// Changes do not take effect until update() is called
im.update();
```

Building a Custom EAR File

Build your custom EAR file, (for example, *myCustomEar***.ear**), with the necessary elements. The structure of the common **interceptors.ear** file has been described in "File Contents".

Example 10–10 shows the structure:

Example 10–10 Example Structure for Custom EAR File

```
/APP-INF
+--/lib/name_your_jar.jar
/META-INF
+--application.xml
+--weblogic-application.xml
+--MANIFEST.MF
```

Contents of META-INF/weblogic-application.xml

The **META-INF/weblogic-application.xml** file should contain the fully qualified class name for your implementation of **weblogic.application.ApplicationLifecycleListener**. Here is an example:

Example 10–11   Custom META-INF/weblogic-application.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <listener>

<listener-class>weblogic.application.ApplicationLifecycleListener.MyCustomListener
</listener-class>
  </listener>
</weblogic-application>
```

Contents of META-INF/application.xml

The **META-INF/application.xml** file should specify the contents of the custom EAR. Here is an example:

Example 10–12   Custom META-INF/application.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.4">
  <display-name>myCustomEAR</display-name>
    <module>
       <java>name_of_my_jar.jar</java>
  </module>
</application>
```

Deploying Your Custom EAR File

If you deploy custom interceptors in a custom EAR file, always deploy the EAR file using the Administration Server and use standard WebLogic procedures to deploy the application to all servers in the cluster from the Administration Server.

For more information, see *Oracle WebLogic Server Documentation* at
`http://www.oracle.com/technetwork/indexes/documentation/index.html`

Updating an Existing Custom EAR to Add Custom Interceptors

If a custom EAR file exists in Services Gatekeeper,

1. Access that custom EAR file and review its contents. Before you update this file, create a backup in a safe location.
2. Create the necessary customer interceptor class, (for example, *MyCustomEnforceThisToo.class*). For details, see "On Customer Interceptor Implementation".
3. Ensure that you have the information necessary to register your custom interceptor and that the index value(s) are synchronized appropriately. For details, see "Information Needed to Register Custom Interceptors".
4. Check the existing custom listener to see whether the registration information currently in use was hard-coded in the custom listener or provided in a data file. (See the discussion under "Creating a Custom Listener".

5. Ensure that the new registration information is made available to that custom listener in the same way. (See the discussion under "The Registration Process".
6. Rebuild the custom EAR file making sure that you preserve its structure. (See the discussion under "Building a Custom EAR File".

Customizing the Interceptor Chain for a Communication Service

This section describes how interceptor chains can be customized for a specific communication service. Interceptor rules can be used to define which interceptors are used based on communication service.

The Services Gatekeeper Plug-in Manager retrieves a list of all eligible interceptors when an initial service request is received. The Plug-in Manager references an internal interceptor rule configuration and removes disabled interceptors for subsequent requests. The customized interceptor chain is then stored in cache so future requests to the same communication service are handed off automatically to the custom interceptor chain.

The interceptor rule configuration is stored in the Services Gatekeeper database and loaded into cache at initial startup. Changes to the configuration will result in a flush of the cached interceptor chains. Subsequent requests then trigger Services Gatekeeper to refresh the cached rule configuration from the database.

Available interceptors in Services Gatekeeper are determined by the **config.xml** file located in the **interceptors.ear** file. See "Interceptors.ear File" for information on configuring available interceptors, including custom interceptors.

Managing Custom Interceptor Filter Rules

To create a custom interceptor rule, create an XML file based on the **interceptorRule.xsd** file located in the *middleware_home*/**ocsg/modules/com.bea.wlcp.wlng.plugin.mngr_6.0.0.0.jar**. The Plug-in Manager MBean is used to create, edit and delete interceptor rule configuration.

This MBean can be accessed from a variety of interfaces including the WebLogic Administration Console, the Platform Test Environment (PTE) or by using the WebLogic Scripting Tool (WLST).

For information on using the WebLogic Administration Console and WSLT, see the Operation and Maintenance chapter in *Services Gatekeeper System Administrator's Guide*.

For information on using the PTE with the Plug-in Manager Mbean, see “Configuring communication services by Changing MBean Attributes and Operations” in *Services Gatekeeper Platform Test Environment User's Guide*.

Interceptor Rule Parameters

The default interceptor configuration is provided in the **interceptorRule.xml** file located in the **com.bea.wlcp.wlng.plugin.mngr_6.0.0.0.jar**. When a new Services Gatekeeper domain is created this configuration is used. Interceptors not included in the configuration file are enabled by default.

Use the Mbean operations available in the Administrator Console to edit the configuration. See "Summary of Tasks Related to Interceptors" for more information.

Interceptor rules contain the elements listed in Table 10–5.

Table 10–5 Interceptor Rule Elements

Name	Type	Description
packageName	string	The plug-in for the communication service for which the rule is to be valid for. Regular expressions can be used in the package name to specify more than one package.
methodName	string	The method for which the rule is to be valid for. Regular expressions can be used in the method name to specify more than one method.
interceptorPoint	tns:InterceptorPoint	The topological system location in Services Gatekeeper where the interceptor chain is applied (**MT_NORTH**, **MT_SOUTH**, **MO_NORTH** or **MO_SOUTH**.
interceptorName	string	The interceptor for which the rule applies. Multiple interceptors can be included if each is enclosed in the **<interceptorName>** tag.
enable	boolean	Boolean indicating if the interceptor(s) listed should be **enabled** or **disabled** in the rule.

Example 10–13 contains an interceptor rule configuration file that performs the following:

- Applies the rule configuration to all parlayrest plug-ins by using a wildcard:
  - **packageName** is set to **..*$**
- Enables the standard Services Gatekeeper interceptors for the **MT_NORTH** interceptor for all parlayrest plug-ins:
  - **interceptorPoint** is set to **MT_NORTH**
  - **interceptorName** lists the standard interceptors included in the rule
  - **enable** is set to **true** allowing all the listed interceptors to run
- Disables the **MT_NORTH** OAuth 2.0 interceptor using a second rule

Example 10–13 Sample interceptorRule.xml Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:interceptorConfig xmlns:tns=
 "http://ocsg.oracle/plugin/xsd/interceptorRule"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ocsg.oracle/plugin/xsd/interceptorRule
interceptorRule.xsd ">
  <!-- following are retrieved from ServiceType.java
oracle.ocsg.parlayrest.plugin.MmsPlugin
oracle.ocsg.parlayrest.callback.MessageNotificationCallback
oracle.ocsg.parlayrest.plugin.PaymentPlugin
oracle.ocsg.parlayrest.plugin.ParlayRestSmsPlugin
oracle.ocsg.parlayrest.callback.ClientSmsNotificationCallback
oracle.ocsg.parlayrest.plugin.TerminalLocationPlugin
  -->
  <tns:interceptorRule>
    <tns:packageName>^oracle\.ocsg\.parlayrest\.plugin\..*$</tns:packageName>
    <tns:methodName>^.*$</tns:methodName>
    <tns:interceptorPoint>MT_NORTH</tns:interceptorPoint>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceApplicationState</tns:in
```

```
terceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceSpAppBudget</tns:interce
ptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceComposedBudget</tns:inte
rceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.FindAndValidateSLAContract</tns
:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.CheckMethodParametersFromSLA</t
ns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceBlacklistedMethodFromSLA
</tns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.InjectValuesInRequestContextFro
mSLA</tns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceNodeBudget</tns:intercep
torName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceSubscriberBudget</tns:in
terceptorName>
    <tns:enable>true</tns:enable>
  </tns:interceptorRule>

  <tns:interceptorRule>

  <!-- Enable/disable OAuth2 interceptor -->
  <tns:interceptorRule>
    <tns:packageName>^.*$</tns:packageName>
    <tns:methodName>^.*$</tns:methodName>
    <tns:interceptorPoint>MT_NORTH</tns:interceptorPoint>

<tns:interceptorName>oracle.ocsg.oauth2.interceptor.OAuth2Interceptor</tns:interce
ptorName>
    <tns:enable>false</tns:enable>
  </tns:interceptorRule>
</tns:interceptorConfig>
```

Summary of Tasks Related to Interceptors

The following is a summary of tasks related to Interceptor Rules.

Interceptor Rules

Table 10–6 lists the tasks related to application accounts and the **PluginManagerMBean** methods you use to perform those tasks.

Table 10–6 Tasks Related to Application Accounts

Task	PluginManagerMBean Method to Use
List the enabled interceptors loaded in Services Gatekeeper	**listInterceptors**
Retrieve the current interceptor rule configuration file	**retrieveInterceptorConfiguration**

Table 10–6 (Cont.) Tasks Related to Application Accounts

Task	PluginManagerMBean Method to Use
Update the interceptor rule configuration file	**updateInterceptorConfiguration**

For a description of the **PluginManagerMBean** MBean operations, see the "All Classes" section of *Services Gatekeeper OAM Java API Reference*.

11

Understanding Aspects, Annotations, EDRs, Alarms, and CDRs

This chapter describes aspects and generation of EDRs, alarms, CDRs, and statistics in Oracle Communications Services Gatekeeper.

About Aspects and Annotations

Aspects allow developers to manage cross-cutting concerns in their code in a straightforward and coherent way. Aspects in Services Gatekeeper (pointcuts, advice, and so on) 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 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 application-facing and network-facing 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.

Understanding the Context Aspect

The Context aspect is woven at compile time, using **PluginSouth** as a marker.

While requests coming from the application-facing 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 network-facing interface do not have a valid context.

The Context aspect solves this problem by rebuilding the context as soon as a network-facing interface method is invoked: after this aspect is executed, a valid context is 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 network-facing interface methods and be able to rebuild the context:

- Each Plug-in must explicitly register its application-facing and network-facing interfaces.
- Each network-facing interface must implement the **resolveAppInstanceGroupdId()** and **prepareRequestContext()** methods of the **PluginSouth** interface.
- application-facing interfaces must implement **PluginNorth** and network-facing 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 actual implementation is 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 aspects call the **resolveAppInstanceGroupId()** method.
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.

Example 11–1 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.

Example 11–2 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();
  }
```

Follow these steps to make your plug-in compliant with the Context aspect:

- Make sure to register all your **PluginSouth** objects before registering your plug-in with the Plug-in Manager.
- Make sure to implement the **resolveAppInstanceGroupdId**() method for each **PluginSouth** instance.
- Annotate each parameter in network-facing object methods that you need to have when aspects call back the **resolveAppInstanceGroupId**() or the **prepareRequestContext**() methods. All the annotated parameters are available in the **ContextMapperInfo** parameter. The aspects need to have them annotated to be able to store them into the **ContextMapperInfo** object.

See the "All Classes" section of the *Oracle Communications Services Gatekeeper Java API Reference* documentation for details on the **PluginSouthMBean** and **PluginNorthMBean**.

Generating EDRs from Communication Services

You can generate EDRs:

- Automatically using aspects at given points in the traffic execution flow in a plug-in.
- Manually anywhere in the code using the **EdrServiceMBean**.

You should generate EDRs at the plug-in boundaries (application-facing and network-facing), 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 these 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 11–1 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

Table 11–1 (Cont.) Manual Annotation for EDRs

Trigger	When	Modifiers restrictions	What is woven
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

These values are always available in an EDR when it is generated from an aspect:

- class name
- method name
- direction the request is going toward (network-facing, application-facing)
- position (before, after)
- interface (application-facing, network-facing, other, null)
- source (method, exception)

See the "All Classes" section of the *Oracle Communications Services Gatekeeper Java API Reference* documentation for details on the **EdrServiceMBean**.

EDR Exception Scenarios

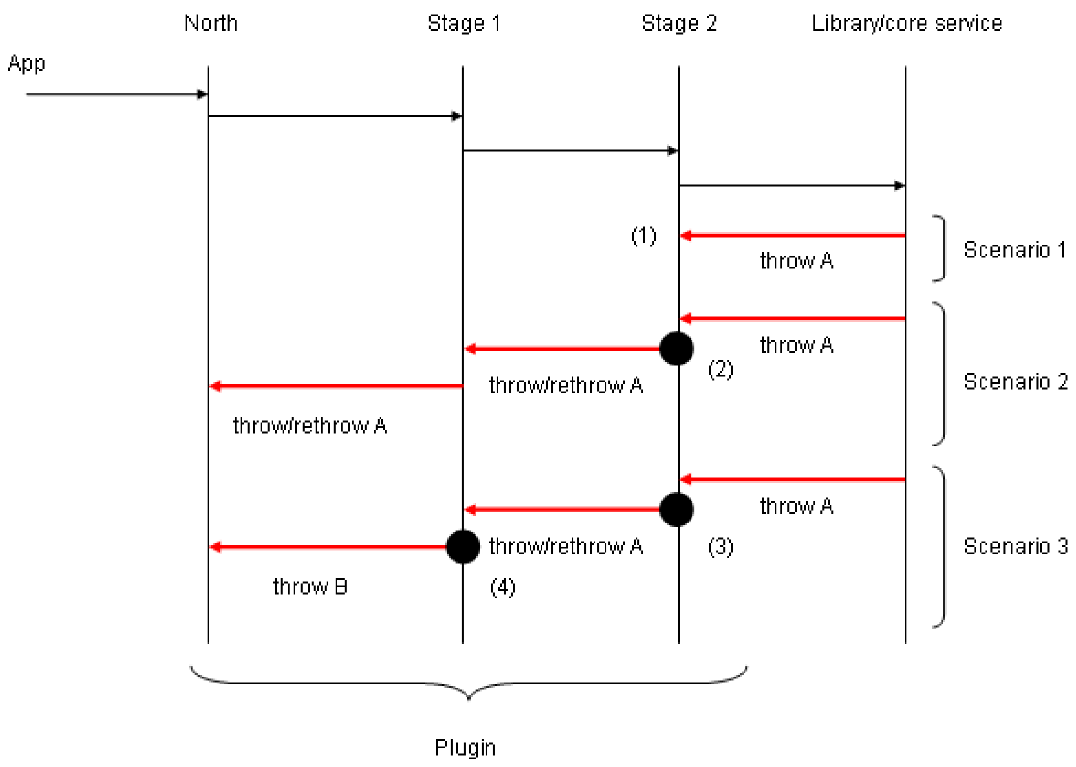
Exceptions are automatically woven by the aspect.

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.

Figure 11–1 illustrates typical scenarios when a library (or core service) throws an exception in the plug-in.

Figure 11–1 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 an EDR

You can add additional key/value pairs to an EDR by using the **putEdr()** method to the **RequestContextMBean**.You can find **putEDR()** here:

```
com.bea.wlcp.wlng.api.plugin.context.RequestContextManager.getCurrent().putEdr("ke
y","value");
```

Also see **RequestContextMBean** in the "All Classes" section of the *Oracle Communications Services Gatekeeper Java API Reference*.

Example 11–3 Adding values using RequestContext

```
...
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 **com.bea.wlcp.wlng.api.edr.EdrConstants** class.

Using translators

When a parameter is a more complex object, you can specify a translator to extract 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

Example 11–4 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.

Example 11–5 Example 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 **ContextTranslatorManager**.

Example 11–6 Registering a Context Translator

```
ContextTranslatorManager.register(ACContextTranslator.class.getName(), new
ACContextTranslator());
```

Table 11–2 is a summary of annotations to use.

Table 11–2 Context Translator 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.

See the "All Classes" section of the *Oracle Communications Services Gatekeeper Java API Reference* documentation for details on the **ContextTranslatorMBean**.

Triggering an EDR Programmatically

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 must manually create and trigger the EDR by following these steps:

1. Create an EdrData object.
2. Trigger the EDR using the EdrService instance .

Example 11–7 shows how to trigger an EDR from inside a plug-in.

Example 11–7 Triggering 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);
      ...
   }
}
```

Understanding Communication Service EDR Content

Table 11–3 describes the content of an EDR generated by a communication service. It shows which values are mandatory, who is responsible for providing these values, and other information.

These fields are always added to a default communication service EDR:

- Class name
- Method name
- Direction (application-facing, network-facing), if the request is travelling from Services Gatekeeper to the network or from the network to Services Gatekeeper
- Position (before, after), if the EDR was emitted before or after the method was invoked or the exception was thrown
- Interface (application-facing or network-facing), if the EDR was emitted from the application-facing interface or from the network-facing interface of the plug-in
- Source (method, exception), if the EDR is related to a method invocation or to an exception

Legends:

- A: Automatically provided by Services Gatekeeper
- H: Provided if the **EdrDataHelper createData** API is used to create the **EdrData** (Oracle recommends this way)
- M: Provided manually in the **EdrData**
- X: Provided in the EDR descriptor.
- C: Custom filter. Use the `<attribute>` element to specify a custom filter.

Note: EDRs triggered by aspects will have all the mandatory fields provided by the aspect.

See the "All Classes" section of the *Oracle Communications Services Gatekeeper Java API Reference* documentation for details on the **EdrData** and **EdrDataHelper**.

Table 11–3 Communication Service EDR Fields

Name	Description	Filter tag name
ApplicationId	Application account ID. Fields in EdrConstants: FIELD_APP_ACCOUNT_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No	C
AppInstanceId	Application instance ID. Fields in EdrConstants: FIELD_APP_INSTANCE_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No.	C

Table 11–3 (Cont.) Communication Service EDR Fields

Name	Description	Filter tag name
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>
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
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>
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>

Table 11–3 (Cont.) Communication Service EDR Fields

Name	Description	Filter tag name
Facade	Facade. Fields in EdrConstants: FIELD_FACADE Values in EdrConstants: VALUE_FACADE_REST, VALUE_FACADE_SOAP Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No.	C
HttpMethod	HTTP request method. For example "POST", or "GET". Fields in EdrConstants: FIELD_HTTP_METHOD Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	
InterceptorChai n	List of all the interceptors that are triggered. Fields in EdrConstants: FIELD_INTERCEPTOR_CHAIN Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	
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>
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 11–3 (Cont.) Communication Service EDR Fields

Name	Description	Filter tag name
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
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>
PluginID	The unique ID of the plug-in instance.	NA
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 Services Gatekeeper server. Fields in EdrConstants: FIELD_SERVER_NAME Provider INSIDE plug-in: H Provider OUTSIDE plug-in: H Mandatory: Yes	C
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

Table 11–3 (Cont.) Communication Service EDR Fields

Name	Description	Filter tag name
SessionId	Session ID. Fields in EdrConstants: FIELD_SESSION_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No	C
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>
State	Where the EDR was dispatched. Fields in EdrConstants: FIELD_STATE Values in EdrConstants: ENTER_AT, ENTER_NT, ENTER_NET, EXIT_AT, EXIT_NT, EXIT_NET Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	<state>
SubscriberId	Subscriber identifier (using route address) Fields in EdrConstants: FIELD_SUBSCRIBER_ID Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	
TagEdr	To get the ID, use getIdentifier() in EdrConfigDescriptor. This value is provided in the EDR descriptor in *domain_home/config/custom/wlng-edr.xm*. Provider INSIDE plug-in: X Provider OUTSIDE plug-in: X Mandatory: Yes	C

Table 11–3 (Cont.) Communication Service EDR Fields

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
TransactionId	Transaction ID. Fields in EdrConstants: FIELD_TRANSACTION_ID Provider INSIDE plug-in: H Provider OUTSIDE plug-in: M Mandatory: No.	C
URL	URL. Fields in EdrConstants: FIELD_URL Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	
WebAppName	Name of the current web application. Fields in EdrConstants: FIELD_WEB_APP_NAME Provider INSIDE plug-in: M Provider OUTSIDE plug-in: M Mandatory: No	
<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.

Example 11–8 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.

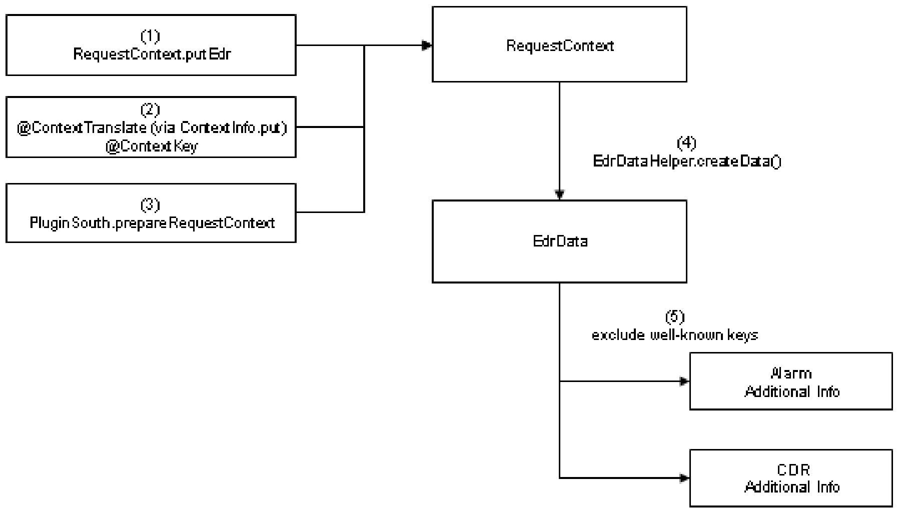
Example 11–9 Pattern to store one single or multiple addresses in field destination using RequestContext.

```
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

Figure 11–2 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 11–2 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 uses 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" and "Understanding CDR Content".

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 *Services Gatekeeper System Administrator's Guide*. File representations of these must be included in **edrjmslistener.jar** if you are using external EDR listeners.

Understanding EDR Descriptors

Each EDR contains a list of descriptors that define it as a pure EDR, as an alarm, or as a CDR.

Table 11–4 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>` element

Table 11–5 describes the elements allowed in the `<filter>` element:

Table 11–5 Elements Allowed in <filter> Element of an EDR Descriptor.

Source	Filter	Min occurs	Max occurs	Description
<method>	N/A	0	unbounde d	Filter EDR triggered by a method
<method>	<name>	0	unbounde d	Name of the method that triggered the EDR

Table 11–5 (Cont.) Elements Allowed in <filter> Element of an EDR Descriptor.

Source	Filter	Min occurs	Max occurs	Description
<method>	<class>	0	unbounde d	Name of the class that triggered the EDR
<method>	<direction>	0	2	Direction of the request
<method>	<interface>	0	3	Interface where the EDR has been triggered
<method>	<position>	0	2	Position relative to the method that triggered the EDR
<exception>	N/A	0	unbounde d	Filter EDR triggered by an exception
<exception>	<name>	0	unbounde d	Name of the exception that triggered the EDR
<exception>	<class>	0	unbounde d	Name of the class where the exception was thrown
<exception>	<method>	0	unbounde d	Name of the method where the exception was thrown
<exception>	<direction>	0	2	Direction of the request
<exception>	<interface>	0	3	Interface where the EDR has been triggered
<exception>	<position>	0	2	Position relative to the method that triggered the EDR
<attribute>	N/A	0	unbounde d	Filter EDR by looking at custom attribute
<attribute>	<key>	1	1	Name of the key
<attribute>	<value>	1	1	Value

Table 11–6 describes the values allowed for each element of the `<filter>` element:

Table 11–6 Values Allowed in Each Element of the <filter> Element

Source	Filter	Allowed values	Comment
<method>	<name>	"returntype nameofmethod([args])"	Method name. The arguments can be omitted with the parenthesis. See "Special characters" below.
<method>	<class>	"fullnameofclass"	Fully qualified class name. See "Special characters" below.
<method>	<direction>	"south", "north"	N/A
<method>	<interface>	"north", "south", "other"	N/A
<method>	<position>	"before", "after"	N/A
<exception >	<name>	"fullnameofexceptionclass"	Fully qualified exception class name. See "Special characters" below.

Table 11–6 (Cont.) Values Allowed in Each Element of the <filter> Element

Source	Filter	Allowed values	Comment
<exception>	<class>	"fullnameofclass"	Fully qualified class name where the exception was triggered. See "Special characters" below.
<exception>	<method>	"returntype nameofmethod([args])"	Method name. The arguments can be omitted with the parenthesis See "Special characters" below.
<exception>	<direction>	"south", "north"	N/A
<exception>	<interface>	"north", "south", "other"	N/A
<exception>	<position>	"before", "after"	N/A
<attribute>	<key>	"astring"	N/A
<attribute>	<value>	"astring"	N/A

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: Using special characters disables the caching of the filter containing them. To avoid a performance hit, Oracle strongly recommends that you avoid special characters in filters.

Table 11–7 Example Filter Elements

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>
All methods starting with void sendInfo.	<method> <name>void sendInfo*</name> ... </method>

Table 11–7 (Cont.) Example Filter Elements

To match on	Use the filter
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 what 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:

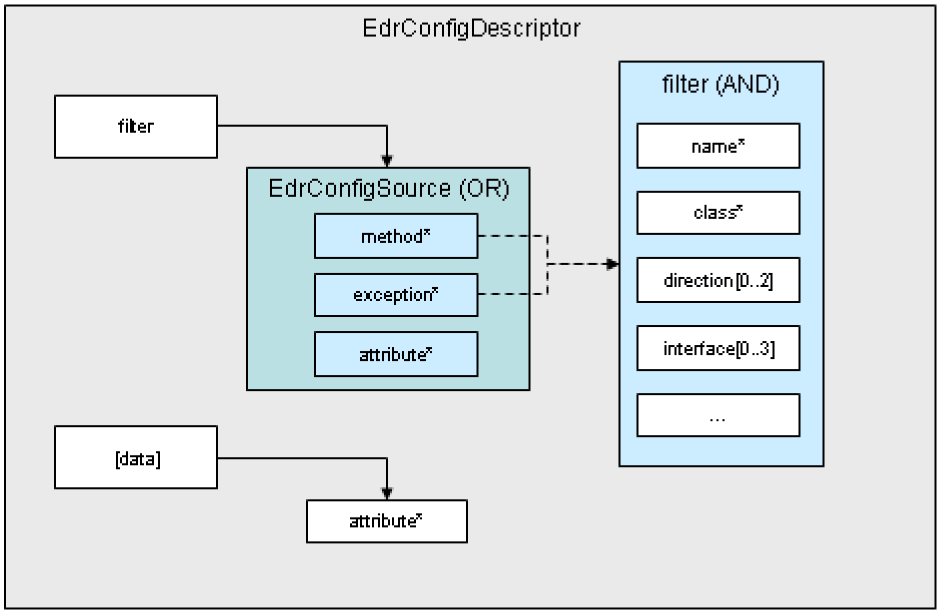
```
com.bea.wlcp.wlng.plugin.sms.smpp.SMPPManagedPluginImpl
```

Boolean Semantic of the Filters

Figure 11–3 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), it is combined with OR.

Figure 11–3 Filter Mechanism



Example EDR Filters

This section provides some example filters that you can use to manipulate EDRs.

Filtering EDRs by Plugin, Direction, and Execution Status

Example 11–10 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 network-facing (direction = south)
- AND the interface where the EDR was trigger is application-facing
- AND the EDR has been triggered after the method has been executed (position = after)

Example 11–10 Categorizing EDRs

```
<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>
```

You can add the exception shown in Example 11–11

- The exception is the **com.bea.wlcp.wlng.plugin.PluginException** class 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.

Example 11–11 Filter Categorizing EDRs as Alarms

```
<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>
```

Categorizing EDRs as Alarms by PluginException, Name, Plugin Name, or Direction

Example 11–12 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 application-facing (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

Example 11–12 Filter Categorizing EDRs as Alarms

```
<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>
```

Filtering Pure EDRs by Plugin Names and Attributes

Example 11–13 (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 application-facing 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 plug-in application-facing 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

Example 11–13 Categorizes EDRS with the ID 1002

```
<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>
```

Manually Triggering an EDR

Example 11–14 shows a manually triggered EDR with its corresponding filter. The EDR is triggered using these lines.

Example 11–14 Manually Triggering an 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);
    ...
  }
```

You can further filter the resulting EDR by method or class name as shown in Example 11–15 (note the various ways of identifying this EDR):

Example 11–15 Filtering by Method and Class Name

```
  <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>
```

Checklist for Aspect 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 network-facing) 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 type 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 **edrjmslistener.jar** file 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 reside in the edr directory in **edrjmslistener.jar**.

Frequently Asked Questions About EDRs and EDR Filters

Question (Q): Is it possible to specify both exception and method name in the filter section?

Example 11–16 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?

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?

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>` element 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>
```

Generating Alarms

An alarm is a subset of an EDR. To generate an alarm, generate an EDR, using either aspects or programmatically, and define the ID and the descriptor of the alarm in the alarm descriptor.

The alarm ID, severity, description and other kinds of attributes are defined in the alarm descriptor, see "Understanding EDR Descriptors". For extensions, the alarm ID should be in the 500 000 to 999 999 range.

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

Triggering an Alarm Programmatically

Trigger an EDR as described in "Understanding Communication Service EDR Content". Then specify in the alarm descriptor the corresponding alarms. Example 11–17 shows you how to do this.

Example 11–17 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 in Example 11–18.

Example 11–18 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

Table 11–8 shows a list of the information provided in alarms.

Table 11–8 Alarm Information for Alarm Listeners, Also Stored in the Database

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.
alarm_info	The alarm information or description. Defined in the alarm descriptor.

Table 11–8 (Cont.) Alarm Information for Alarm Listeners, Also Stored in the Database

Field	Comment
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_STATE ▪ FIELD_EXCEPTION_NAME ▪ FIELD_ORIGINATING_ADDRESS ▪ FIELD_DESTINATION_ADDRESS ▪ FIELD_CONTAINER_TRANSACTION_ID ▪ FIELD_APP_INSTANCE_ID ▪ FIELD_FACADE ▪ FIELD_CORRELATOR ▪ FIELD_SESSION_ID ▪ FIELD_SERVER_NAME ▪ FIELD_URL ▪ FIELD_WEB_APP_NAME ▪ FIELD_REQUEST_CONTEXT ▪ FIELD_HTTP_METHOD ▪ FIELD_INTERCEPTOR_CHAIN ▪ FIELD_SUBSCRIBER_ID ▪ ExternalInvocatorFactory.SERVICE_CORRELATION_ID ▪ FIELD_BC_EDR_ID ▪ FIELD_BC_EDR_ID_3 ▪ FIELD_BC_ALARM_IDENTIFIER ▪ FIELD_BC_ALARM_INFO

Generating CDRs

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.

Understanding the Default CDRs

Services Gatekeeper generates a default set of CDRs which can be customized by re-configuring the CDR descriptor. You will probably need to configure CDR generation to meet the needs of your implementation.

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 network-facing interface after the method has returned back to the network. For application-triggered requests generate a CDR at the application-facing interface after the method has returned to the Network Tier SLSB.

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.

Triggering 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 "Triggering an EDR Programmatically" 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. Example 11–19 shows how to do this.

Example 11–19 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);
...
```

Example 11–20 shows the description in the CDR descriptor that matches this EDR.

Example 11–20 CDR Descriptor Descripton

```
<cdr>
    <filter>
      <method>
        <name>com.bea.wlcp.wlng.myEndOfRequestMethod</name>
        <class>com.bea.wlcp.wlng.myClass</class>
      </method>
    </filter>
```

```
</cdr>
```

Understanding CDR Content

In addition to the EDR fields, there are specific fields used only for CDRs. They are listed in Table 11–5.

Table 11–9 Fields in EdrConstants specific for CDRs.

Field in EdrConstants	Comment
FIELD_CDR_SESSION_ID	Session ID
FIELD_CDR_START_OF_USAGE	Start Time
FIELD_CDR_CONNECT_TIME	Connect Time
FIELD_CDR_END_OF_USAGE	End Time
FIELD_CDR_DURATION_OF_USAGE	Duration
FIELD_CDR_AMOUNT_OF_USAGE	Amount
FIELD_CDR_ORIGINATING_PARTY	Originating Party
FIELD_CDR_DESTINATION_PARTY	Same pattern applies as for send lists, see "Using Send Lists".
FIELD_CDR_CHARGING_INFO	Charging Information

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 the database
- RC: Renamed column in database

Table 11–10 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
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

Table 11–10 (Cont.) Content in database

Field	Comment	DB
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
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

Understanding the additional_info Database Column

The EDR populates the **additional_info** column of the database with all the custom key/value pairs found in the **EdrData** except the ones listed here.

Excluded keys (EdrConstants if not specified):

- FIELD_SERVICE_NAME
- FIELD_APP_INSTANCE_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_STATE
- FIELD_EXCEPTION_NAME
- FIELD_ORIGINATING_ADDRESS
- FIELD_DESTINATION_ADDRESS
- FIELD_CORRELATOR
- FIELD_APP_ACCOUNT_ID
- FIELD_SESSION_ID
- FIELD_TRANSACTION_ID
- FIELD_FACADE
- FIELD_URL
- FIELD_WEB_APP_NAME

- FIELD_REQUEST_CONTEXT
- FIELD_INTERCEPTOR_CHAIN
- FIELD_SUBSCRIBER_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 11–11 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

This is the **additional_info** field format:

```
key=value\n
```

similar to the Java properties file.

12 Using SLA Policies to Manage Subscribers

This chapter describes how you use Oracle Communications Services Gatekeeper Platform Development Studio to use policy-based control on your subscriber base.

There is an example Profile Provider in *Middleware_home*/**ocsg_pds/example**.

About Using Policies to Manage Subscribers

As a network operator, you can use Services Gatekeeper to evaluate the status of requests in terms of policy, or rules governing a variety of service characteristics and manage the normal request traffic policy evaluation flow using service contracts to evaluate requests and to generate subscriber budgets.

To do so, you 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 you or your network integrator can associate those service classes with subscriber URIs to create subscriber contracts. A single subscriber can be covered by multiple subscriber contracts, based on that individual subscriber's requirements.

Service Classes and the Subscriber SLA

Making subscriber personalization easy and offering superior subscriber data protection is key to growing and maintaining a loyal subscriber base.

The first step in adding subscriber-centric policy to Services Gatekeeper is to create a Subscriber SLA. This is an XML file based on the sub_sla_file.xsd schema.

The schema file can be found in the **wlng.jar** file located in the *Middleware_home*/**ocsg_pds/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 the discussion on managing application service providers in *Services Gatekeeper Concepts*. 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> element

The `<reference>` element 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

ServiceProviderGroupReference. In addition there are two additional reference types, **ServiceReference** and **MethodReference** that indicate specific service interfaces and methods, respectively, covered by those groups. In the Example 12–1 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 Example 12–1, 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**)

Example 12–1 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 `<references/>` element matches everything.

The <restriction> element

In addition to the `<reference>` element, service classes may have a `<restriction>` element. This element 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/>` element.

Example 12–2 The <restriction> element

```
<restriction>
            <rate>
              <reqLimit>5</reqLimit>
              <timePeriod>1000</timePeriod>
            </rate>
            <quota>
              <qtaLimit>600</qtaLimit>
              <days>3</days>
              <limitExceedOK>true</limitExceedOK>
            </quota>
</restriction>
```

These elements function exactly as they do in the other SLAs in Services Gatekeeper. For more information on these elements, 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>` element 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/>` element. This element, 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 "Managing SLAs" in *Services Gatekeeper Extension Developer's Guide*. 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**.

The subscriber contract constructor throws 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 *Middleware_home*/**ocsg_pds/lib/wlng** directory. The documentation for the files is in the "All Classes" section of the *Services Gatekeeper Java API Reference*. An example implementation can be found in the *Middleware_home*/**ocsg_pds/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 *Middleware_home*/**ocsg_pds/example/profile_providers/resource** directory.

The Profile Provider must implement the Profile Provider SPI. The SPI defines three methods;

- **init**: 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**: 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, 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 **ProfileProviderImpl** has been created, the JAR file containing it must be added to the **app-inf/lib** directory in the **profile_providers.ear** file, which can be found in the *Middleware_home*/**ocsg/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 file is modified, it can be deployed in the normal manner. For more information on deploying EAR files see discussion on deployment model for communications services in *Services Gatekeeper System Administrator's Guide*.

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 Services Gatekeeper, see the discussion on managing and configuring budgets in *Services Gatekeeper System Administrator's Guide*.

When a request from an application arrives at 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?

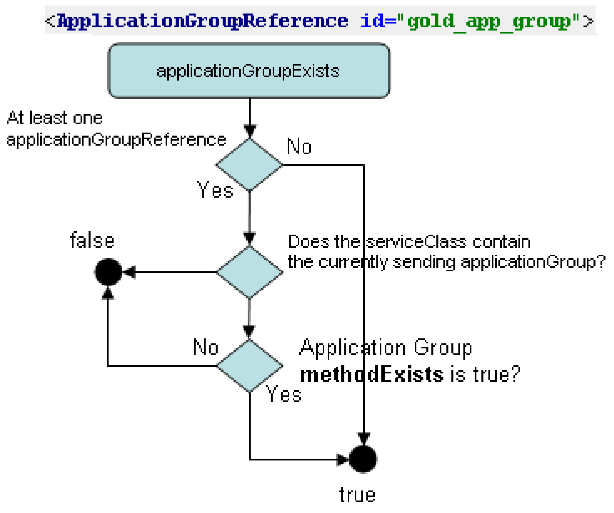
The first thing the interceptor must determine is whether one or more contracts exist that are relevant to the particular request that is being evaluated. The interceptor iterates through all the target URIs in the application request, and evaluates whether or not there are contracts in effect that it 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 Example 12–1.

  - Is there an **ApplicationGroupReference** and is it relevant? See Figure 12–1.

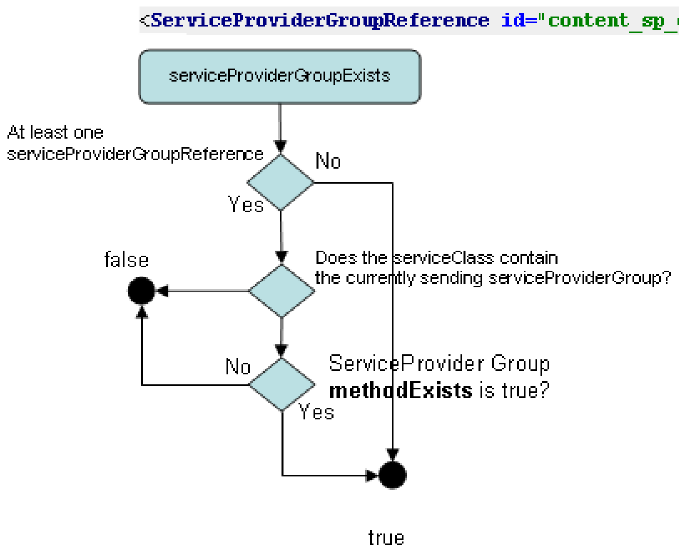
Figure 12–1 Application Group Reference Evaluation


<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 12–3.

- Is there a **ServiceProviderGroupReference** and is it relevant? See Figure 12–2.

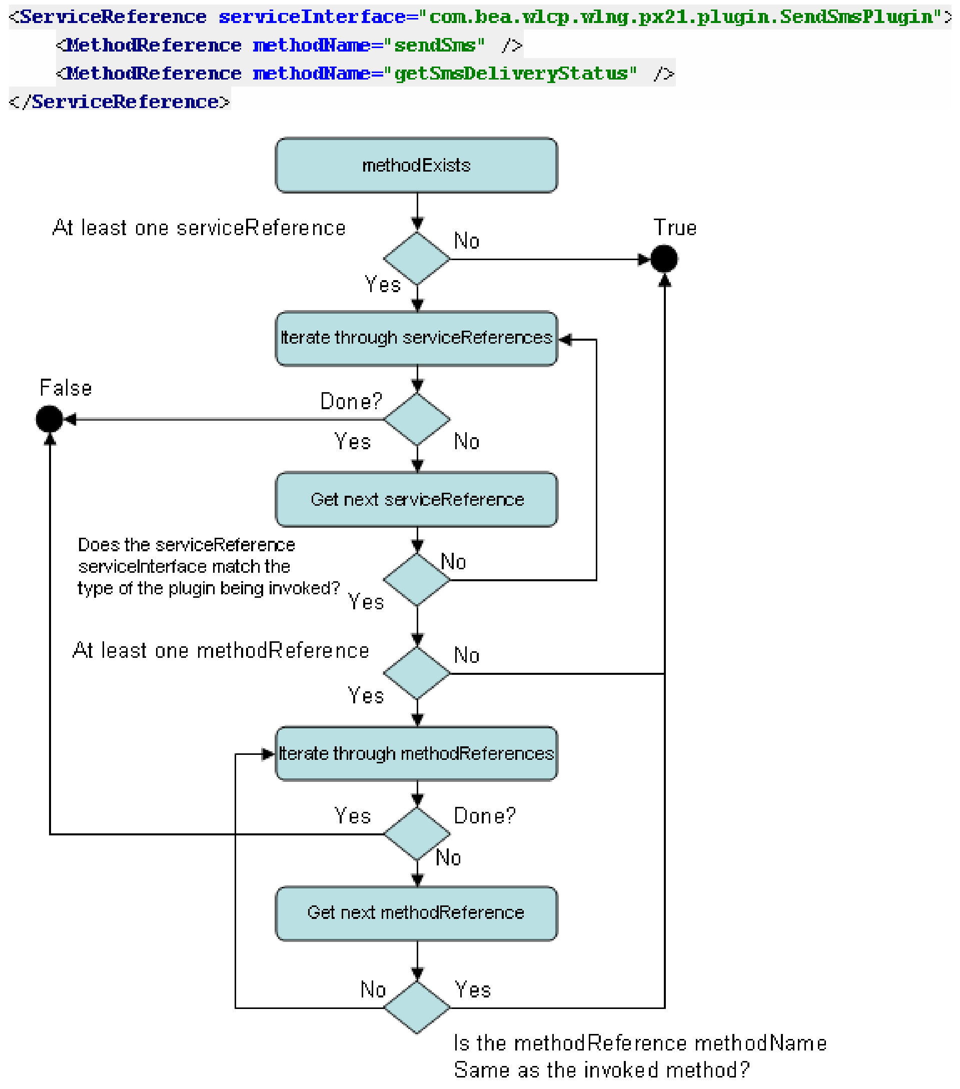
Figure 12–2 Service Provider Group Reference Evaluation


<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 12–3.

- Is there a Service Reference (and possibly a MethodReference) and are they relevant? See Figure 12–3.

Figure 12–3 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>` element set to `<restrictAll/>`. If so, the request is immediately denied, and processing on the request ceases.

If the `<restriction>` element is not set to `<restrictAll/>`, the decision flow is identical to the other budget evaluations that take place in 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.

13

Creating Custom Runtime SLAs

This chapter describes how to enforce custom service level agreements (SLAs) and explains the relationship between the custom SLAs, their XSDs, and the enforcement logic in Oracle Communications Services Gatekeeper.

Introduction

Custom service level agreements (SLAs), offer a mechanism to add custom SLA enforcement in addition to the SLA enforcement provided by default with 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 for 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 "Managing SLAs" in *Services Gatekeeper Extension Developer's Guide*.

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 **com.bea.wlcp.wlng.api.interceptor.Context** class.

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 "Using Service Interceptors to Manipulate Requests".

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 "Using Service Interceptors to Manipulate Requests".

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. 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" for more information. The custom SLA XSD is loaded in 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. See "Enforcement Logic" for more detailed information.

Custom SLA Schema and Example SLA

Example 13–1 is an example of a SLA schema that allows a set of content types to be defined.

Example 13–1 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>
```

Example 13–2 is an SLA that adheres to the schema in Example 13–1. It allows the content type Entertainment.

Example 13–2 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 "Using Service Interceptors to Manipulate Requests" for more information.

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.

14

Customizing SLA Behavior for a Service Provider or Application

This chapter explains how Oracle Communications Services Gatekeeper plug-ins can use the generic data in SLAs to customize the behavior of the plug-in.

Understanding How to Customize Behavior Based on SLAs

Plug-ins can specify different behavior for individual service providers or applications using the generic data specified in their service provider or application level SLAs. The plug-in uses generic data specified in the SLAs to do this.

This capability is useful when the data used by a plug-in should be different depending on which service provider or application that the request originates. For example, this can be used for information about parameters that corresponds to a certain group of applications. 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 `<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, you can retrieve the value associated with the **contextAttribute** with the **attributeName com.bea.wlcp.wlng.plugin.sms.testName1**:

```
String value1 =
(String)RequestContextManager.getCurrent().get("com.bea.wlcp.wlng.plugin.sms.testN
ame1");
```

15 Customizing Diameter AVPs

This chapter describes how to customize Diameter AVPs (Attribute-Value Pairs) in Oracle Communications Services Gatekeeper for:

- Parlay X 3.0 Payment Diameter communication service
- Credit Control Interceptor
- CDR Diameter listener

Understanding Customized Diameter AVPs

You can add or modify the Diameter AVPs that Services Gatekeeper sends to the network.

A set of standard AVPs are sent using Diameter, and you can add additional AVPs and modify them using configuration. Applications can also provide custom AVPs as tunnelled parameters, and they can receive returned AVPs using tunneled parameters.

This chapter starts with sections that describe how to configure custom AVPs for these entities:

- Configuring Customized AVPs for Parlay X 3.0 Payment/Diameter
- Configuring Customized AVPs for Credit Control Interceptor
- Configuring Customized AVPs for CDR Diameter Listener

Finally this chapter explains how to customize AVPs for any application in the "Dynamically Customizing AVPs for Applications" section.

Configuring Customized AVPs for Parlay X 3.0 Payment/Diameter

The Parlay X 3.0 Payment/Diameter communication service translates Parlay X requests or requests over the RESTful interfaces to Diameter calls. You can add and modify the Diameter AVPs that are sent in the Diameter request using a custom global SLA or a custom service provider SLA. See *Services Gatekeeper Extension Developer's Guide* for more information. Use the SLA type **payment_diameter_avp** when loading the SLA.

The custom SLA has the following structure:

```
<tns:paymentConfig>
   <tns:avpAttributeDefinitions>
      <avp:avpAttribute></avp:avpAttribute>
      ...
```

```
    </tns:avpAttributeDefinitions>

    <tns:avpTemplate>

        <avp:avpValue/>

        ...

    </tns:avpTemplate>

</tns:paymentConfig>
```

The `paymentConfig` element contains one instance of `avpAttributeDefinitions` and a sequence of `avtTemplate`. It has no attributes.

The `avpAttribute` element specifies the AVP attribute to use and defines its characteristics. One or more `avpAttribute` elements can be defined under `avpAttributeDefinitions`. The `avpAttribute` element has the following attributes:

- **code** Required. Defines the AVP attribute code.
- **vendorId** Required. Defines the Vendor ID for the AVP.
- **name** Required. Symbolic name to use when referring to the AVP attribute definition in the `avpValue` element.
- **type** Required. Type is String. Defines the data type of the AVP. Possible values are:
  - INTEGER32
  - INTEGER64
  - FLOAT32
  - FLOAT64
  - STRING
  - ADDRESS
  - GROUPED
  - BYTES

  If the type is GROUPED, the AVP attribute is a grouped attribute and a sequence of `avpAttribute` elements can be added as siblings to a `avpAttribute`.
- `flag` default is 64. Use one of the following values:
  - FLAG_NONE = 0x0 (0)
  - FLAG_VENDOR_SPECIFIC = 0x80 (128)
  - FLAG_MANDATORY =0x40 (64)
  - FLAG_END_TO_END_ENCRYPTION = 0x20 (32)

  You can combine these flags by adding their values together.

The value parts of the AVPs are defined as sequence of `avpValue` elements. The `avpValue` is a sibling to the `avpTemplate` element.

The `avpTemplate` element defines the optional `paramName` attribute. If a `ParamName` is not specified, the template that has the same name as the calling operation is used.

The `avpValue` element defines the value of an AVP. It has two attributes:

- `avpName` Required. Type is String. Points to the `avpAttribute` the value corresponds to.
- `defaultValue` Optional. Type is String. Defines the value of the AVP. When it is a grouped AVP (`type` is set to GROUPED) the value must be null, otherwise it must have a value.

Example 15–1 illustrates a custom AVP definition for Payment. It defines three AVP attributes and shows how to set the values for these.

Example 15–1 Custom AVP definition for Payment

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:paymentConfig xmlns:avp="http://ocsg.oracle/diameterAvp/xml"
xmlns:tns="http://ocsg.oracle/plugin/payment/diameter/xml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <tns:avpAttributeDefinitions>
    <avp:avpAttribute code="3001" vendorId="111" name="test-avp-1" type="String"
flag="0"></avp:avpAttribute>
    <avp:avpAttribute code="3002" vendorId="111" name="test-avp-2" type="Grouped"
flag="0">
            <avp:avpAttribute code="3003" vendorId="111" name="test-avp-3"
type="Integer32" flag="0"></avp:avpAttribute>
    </avp:avpAttribute>
  </tns:avpAttributeDefinitions>

  <!-- default template -->
  <tns:avpTemplate>
    <avp:avpValue avpName="test-avp-1" defaultValue="hello world."/>
    <avp:avpValue avpName="test-avp-2">
      <avp:avpValue avpName="test-avp-3" defaultValue="2"/>
    </avp:avpValue>
  </tns:avpTemplate>

  <!-- custom  template -->
  <tns:avpTemplate ParameterName="template1">
    <avp:avpValue avpName="test-avp-1" defaultValue="hello template 1"/>
    <avp:avpValue avpName="test-avp-2">
      <avp:avpValue avpName="test-avp-3" defaultValue="20"/>
    </avp:avpValue>
  </tns:avpTemplate>

</tns:paymentConfig>
```

Configuring Customized AVPs for Credit Control Interceptor

The Credit Control Interceptor sends Diameter requests based on the content of a request from an application. For each request, you can specify one or more AVPs to send in the request.

You specify the AVPs in an custom global SLA or custom service provider SLA. See *Services Gatekeeper Extension Developer's Guide* for more information. Use the SLA type **credit_control** when loading the SLA. For information about Credit Control Interceptors and SLAs, see "Implementing Diameter Ro Charging using Credit Control Interceptors" in *Services Gatekeeper System Administrator's Guide*.

The AVPs are defined in the `avpAttributeDefinitions` elements in the credit control SLA. Define this as a sibling to the `CCInterceptions` element.

The SLA has the following structure:

```
<CCInterceptions>
   <avpAttributeDefinitions>
      <avpAttribute></avp:avpAttribute>
      ...
   </avpAttributeDefinitions>
    <CCInterception>
      <SubscriptionId>...</SubscriptionId>
      <OCSGChargeDescription>...</OCSGChargeDescription>
      <ServiceContextId>...</ServiceContextId>
      <Amount>...</Amount>
      <Currency>...</Currency>
      <ServiceIdentifier>...</ServiceIdentifier>
      <CallingPartyAddress>...</CallingPartyAddress>
      <CalledPartyAddress>...</CalledPartyAddress>
      <AsynchronousCommit></AsynchronousCommit>
      <customizedAvpValues>...</customizedAvpValues>
    </CCInterception>
    <CCInterception>
       ...
    </CCInterception>
</CCInterceptions>
```

The `CCInterceptions` element contains a sequence of `CCInterception` and `avpAttributeDefinitions` elements. It has no attributes.

The `CCInterception` element defines the data to set in an AVP and it specifies which method and interface the request that the AVPs are valid for.

It has the following attributes:

- `interfaceName` Required. Type is String. The Java representation of the Parlay X interface name. For example, **com.bea.wlcp.wlng.px21.plugin.SendSmsPlugin**.
- `methodName` Required. Type is String. The method name. For example, **sendSms**.
- `pluginId` Optional. Type is String. The Id of the plug-in instance the AVP is defined for and serves as a default definition. If it is not present, it is valid for all plug-in instances. If it is present, it overrides the default AVP definition.

The `avpAttribute` element is identical to the `avpAttribute` element for the Payment plug-in, see "Configuring Customized AVPs for Parlay X 3.0 Payment/Diameter".

Configuring Customized AVPs for CDR Diameter Listener

The CDR Diameter listener converts CDRs emitted by Services Gatekeeper to Diameter Requests.

The mapping from a CDR to Diameter AVPs is defined in the XML file **mapping.xml**. The file is located in the EAR files **cdr_to_diameter-single.ear** and **cdr_to_diameter.ear**, in the **APP-INF\classes** directory.

You can define two types of AVPs: dynamic and static.

Dynamic AVPs take fields in the CDR and defines which AVP it is mapped to. The Static AVPs define static values.

The XML file has the following structure:

```
<mappings>

   <mapping/>

   ...

   <mapping/>

</mappings>
```

The `mappings` element contains a sequence of `mapping`. It has no attributes

The mapping element has the following attributes:

- `edr` Optional. Type is String. Defines the EDR ID to convert.
- `avp` Required. Type is String. Defines the AVP attribute name part, for example Called-Party-Address.
- `avpType` Required. Type is String. Defines the data type of the AVP. Possible values are:
  - INTEGER32
  - INTEGER64
  - FLOAT32
  - FLOAT64
  - STRING
  - ADDRESS
  - GROUPED
  - BYTES
- `vendorId` Optional. Type is int. Defines the vendor-ID included in the Diameter request. For example, Oracle's vendor-ID is 111.
- `mandatory` Optional. Type is Boolean. Defines if the AVP is mandatory or not.
- `vendorSpecific` Optional. Type is Boolean. Specifies if the AVP is specific for the vendor or if it is a standard AVP.
- `endToEndEncryption` Optional.Type is Boolean. Specifies if end-to-end encryption shall be used for the request.
- `avpCode` Required. Type is int. Specifies the numeric value for the AVP attribute part.
- `mapper` Optional. Type is String. Specifies the class that performs the mapping. Use the fully qualified class name, including he package name.
- `avpValue` Optional. Type is String. Specifies the value-part of the AVP.

The following rules apply:

- If the specified `edr` attribute exists, the value in the EDR is forwarded in the Diameter ACR (account request).
- f the value in the EDR is an array, all entries are sent as comma-separated values.
- If there is no EDR attribute that matches **edr**, the value set in `avpValue` is used.

To define a dynamic EDR to AVP mapping for the time stamp, use the following XML:

```
<mapping edr="Timestamp" avp="Event-Timestamp" mandatory="true"
avpCode="55" avpType="INTEGER32"
mapper="com.bea.wlcp.wlng.cdrdiameter.xmlmapper.avpmapper.TimeStampAVPMapp
er"/>
```

To define a static AVP, use the following XML:

```
<mapping edr="CustomizedAVP1" avp="Ocsg-Customized-1" mandatory="false"
avpCode="3001" vendorId="111" avpType="STRING" avpValue="test"/>
```

Dynamically Customizing AVPs for Applications

Applications can define customized AVPs to be sent in Diameter requests by using tunneled parameters (x parameters) that define the AVPs to be added. The AVPs need to be configured in the SLAs using the same key name as used in the tunneled parameter. AVPs are forwarded in the responses to requests from the SOAP and RESTful interfaces for requests to the Payment communication service.

See *Services Gatekeeper Communication Service Reference Guide* for descriptions of the tunneled parameters used by individual communication services.

The parameters are defined using key-value pairs encapsulated by the tag `<xparams>`. The xparams tag can include one or more `<param>` tags. Each `<param>` tag has a **key** attribute that identifies the parameter and a `value` attribute that defines the value of the parameter. Set the value for the key in the tunneled parameter to the value of the **paramName** attribute in the SLA to use a configured AVP.

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

You can block requests that contain tunneled parameters that have not been configured as allowed. Filtering is on a global, not application, level. See "About Filtering Tunneled Parameters" in *Services Gatekeeper System Administrator's Guide* for more information on configuring which tunneling parameters are allowed or blocked in Services Gatekeeper.

Example 15–2 Retrieving the Value of the Tunneled Parameter 'aParameterName'

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

A parameter tunneled from the application is overridden, if the same parameter is defined in the `<contextAttribute>` SLA element. This behavior, however, is defined per plug-in.

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

The key-value pairs for SOAP-based tunneled parameters are defined this way:

```
<soapenv:Header>
...
```

```
<xparams>
    <param key="key1" value="value1" />
    <param key="key2" value="value2" />
</xparams>
...
</soapenv:Header>
```

When you use the SOAP interfaces, the AVPs are returned in the SOAP header in a tunneled parameter with the attribute `key` set to **AVP_LIST** and the attribute `value` set to an XML encoded string representing the AVP. An example is:

`<param key="`**AVP_LIST**`" value="`*AVP_list_in_XML*`" />`

When you use the RESTful interfaces, the values should be encoded in the HTTP header this way:

```
X-Plugin-Param-Keys:template1,template2
```

```
X-Plugin-Param-Values:10,10
```

The key and the value for the tunnelled parameters are ordered so the first occurrence in **X-Plugin-Param-Keys** is for the first occurrence of **X-Plugin-Param-Values**, and so on.

When the XML is returned in a SOAP header, values are escaped. For example the **<** character is converted to **&lt;**.

When you use the RESTful interfaces, the AVPs are returned in the HTTP response header in an tunneled parameter with the attribute **X-Plugin-Param-Keys** set to **AVP_LIST** and the attribute **X-Plugin-Param-Values** set to an XML encoded string representing the AVP.

The key-value pairs for REST-based tunneled parameter (xparam) request headers are defined this way:

```
X-Param-Keys: key1,key2
X-Param-Values: value1,value2
```

The key-value pairs for REST-based response headers are defined this way:

```
X-Plugin-Param-Keys: key1,key2
X-Plugin-Param-Values: value1,value2
```

This example shows an AVP list expressed in XML:

```
<Avp-List>

 <Session-Id>;1280993750;3</Session-Id>

 <Origin-Host>ocag.oracle.com</Origin-Host>

 <Origin-Realm>oracle.com</Origin-Realm>

 <Result-Code>2001</Result-Code>

 <CC-Request-Type>4</CC-Request-Type>

 <CC-Request-Number>0</CC-Request-Number>

</Avp-List>
```

When the XML is returned in the HTTP header, all **<CR>** (carriage return) and **<LF>** (line feed) characters are removed.

16 Creating EDR Listeners

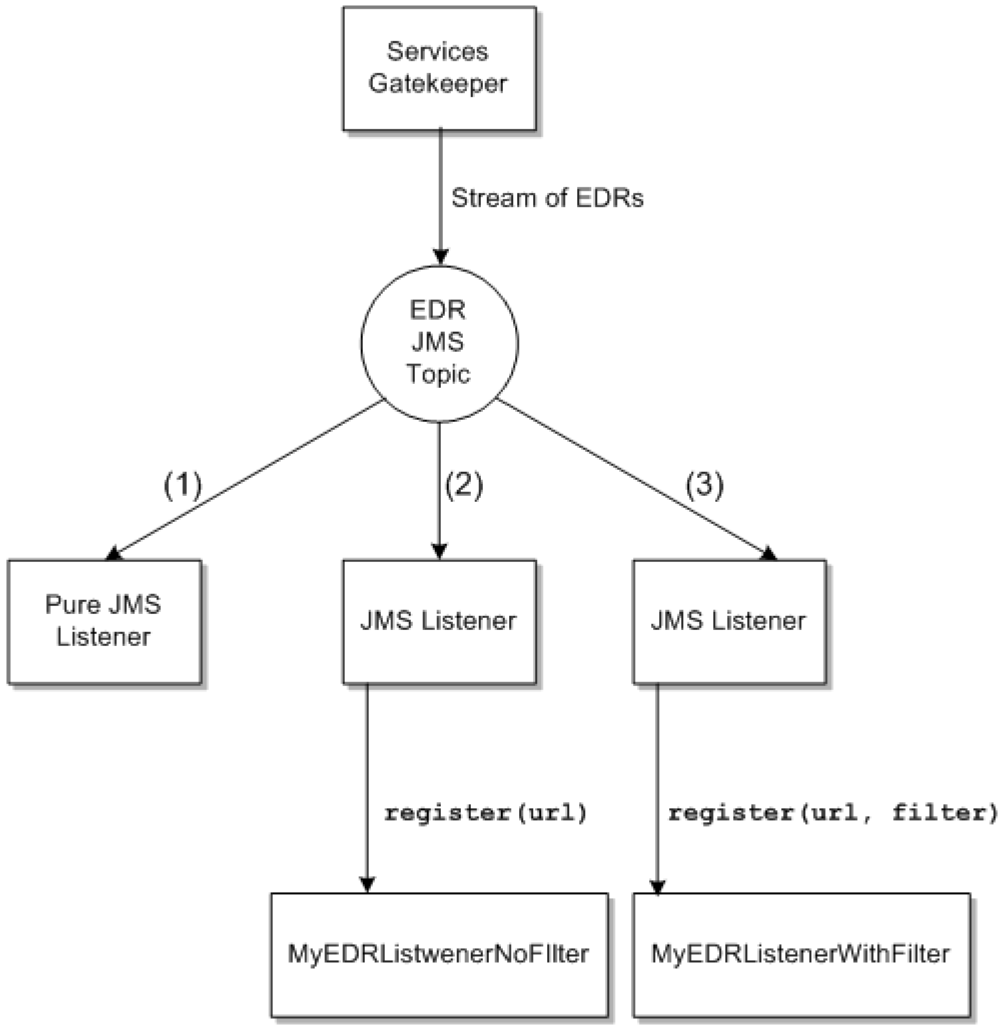
This chapter describes how to create an external event data record (EDR) listener in Oracle Communications Services Gatekeeper.

Understanding External EDR Listeners

External EDR listeners are Java Message Service (JMS) topic subscribers.

The diagram below illustrates three different ways of listening for EDRs as a JMS listener.

Figure 16–1 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 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

Example 16–1 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

Example 16–2 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

Example 16–3 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 Services Gatekeeper is also updated in the edrjmslistener.jar file.

Understanding an 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 in *Middleware_home*/**ocsg_pds/lib/wlng/edrjmslistener.jar**.

Class JMSListener

Table 16–1 lists the important **JMSListener** methods. See the "All Classes" section of *Services Gatekeeper Java API Reference* for details on **JMSListener**.

Table 16–1 JMSListener Methods

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 is a subclass of **ConfigDescriptor** that identifies the type of EDR: one of **EdrConfigDescriptor**, **AlarmConfigDescriptor** or **CdrConfigDescriptor**.

Understanding the Helper JMSListener Helper Classes

Table 16–2 shows the JMSListener helper classes and their important methods. See the "All Classes" section of *Services Gatekeeper Java API Reference* for details on these **JMSListener** helper classes.

Table 16–2 JMSLisytener Helper Classes

JMSListener Helper Class	Description
EdrFilterFactory	Creates the default filter used by Services Gatekeeper to filter the EDRs using the **edr.xml**, **alarm.xml**, or **cdr.xml** file in the **edrjmslistener.jar** file, depending on EDR listener alternative used.
EdrData	Does one of the following: ■ Contains all the values that EDRs (alarm and CDR) have. ■ Gets the value associated with the specified key. ■ Gets the values associated with the specified key.
ConfigDescriptor	The parent class of **EdrConfigDescriptor**, **AlarmConfigDescriptor** and **CdrConfigDescriptor**.
EdrConfigDescriptor	Does one of the following: ■ Contains the data specified in the descriptors in the **edr.xml** configuration file: the identifier and the description. ■ Returns the identifier of the EDR. ■ Returns the description of the EDR.
AlarmConfigDescriptor	Contains the data specified in the descriptors in the **alarm.xml** configuration file: the identifier, the severity and the description of the alarm.
CdrConfigDescriptor	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 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**.

17

Making Communication Services Manageable

This chapter explains how to make new communications services manageable by Communications Services Gatekeeper.

Understanding Communication Service Management

Once you have created an extension communication service, you must make it manageable by Services Gatekeeper. You do this by exposing the operations and management (OAM) functions, such as read/write attributes and or operations, in a way that allows them to be accessed and manipulated by the Services Gatekeeper Administration Console extension, or other management tools.

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 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. This makes the functionality available to the Distributed Services layer, management tools like the Services Gatekeeper Administration Console. Finally, because configuration information needs to be persisted, you store the values you set using the Services Gatekeeper 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 Fusion Middleware Developing Manageable Applications With JMX for Oracle WebLogic Server* at:

`http://docs.oracle.com/cd/E24329_01/web.1211/e24416/designapp.htm`

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".

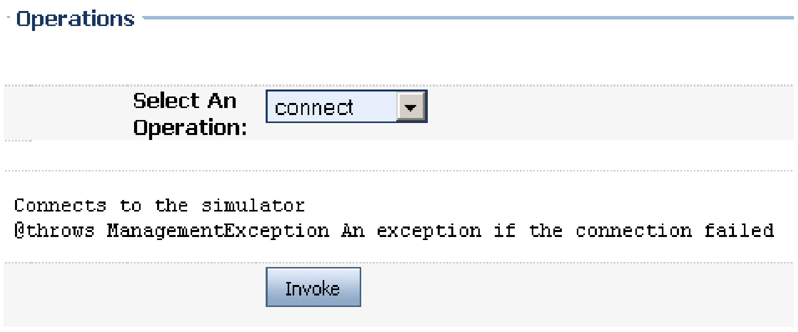
Create an MBean Interface

You must first create an interface file that describes the 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). Also create 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 user interface 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 is rendered in the console as a description of an attribute or operation. It renders exactly as in the JavaDoc. For example:

```
/**
   * Connects to the simulator
   * @throws ManagementException An exception if the connection failed
   */
  public void connect() throws ManagementException;
```

  Renders 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. For eample:

```
@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. For 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;
```

Name the interface *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, and 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. 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 17–1.

Table 17–1 MBean ObjectName

The ObjectName convention for extensions	Description
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#6.0,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 17–2.

Table 17–2 MBean ObjectName with hierarchy

The ObjectName convention for extensions	Description
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
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#6.0,InstanceName= Plugin_px21_short_
messaging_smpp,
Type=com.bea.wlcp.wlng.plugin.sms.smpp.management.SmsMBean,Level1InstanceName=Hear
tBeatManager,Level1Type=com.bea.wlcp.wlng.heartbeat.management.HeartbeatMBean

com.bea.wlcp.wlng:AppName= wlng_nt_multimedia_messaging_px21#6.0,InstanceName
Plugin_px21_multimedia_messaging_mm7, Type=
com.bea.wlcp.wlng.plugin.multimediamessaging.mm7.management.MessagingManagementMBe
an,Level1InstanceName=HeartBeatManager,Level1Type=com.bea.wlcp.wlng.heartbeat.mana
gement.HeartbeatMBean
```

Use the Configuration Store to Persist Values

The 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.

The configuration store supports only Boolean, Integer, Long, and String values.

18 Extending the ATE and PTE for Your Communication Services

The chapter describes how to generate and build virtual communication services, clients, and simulators for the Application Test Environment (ATE) and the Platform Test Environment (PTE) in Oracle Communications Services Gatekeeper.

For complementary information, see "Adding and Testing Custom Client Modules" in *Services Gatekeeper Platform Test Environment User's Guide*.

Understanding ATE and PTE Extensions

You use the ATE to create virtual communication services, and interact with and extend to the network simulator. The ATE uses these extension points:

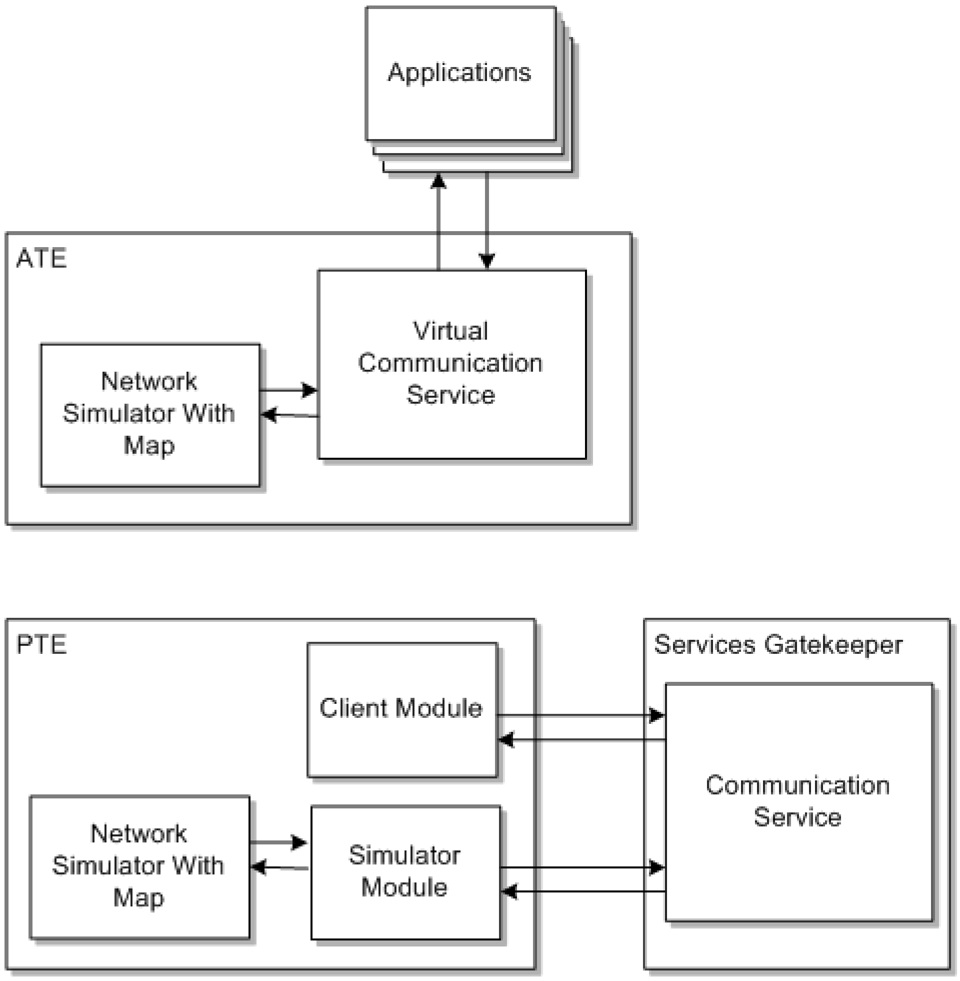
- Virtual communication services
- The network simulator part of the ATE

You use the PTE to create client modules that act as applications, and simulator modules that act as network elements and present results and statistics in the PTE user interface. The PTE uses these extension points:

- Clients
- Network protocol simulators
- The network simulator part of the PTE

Figure 18–1 shows how the PTE and ATE work with Services Gatekeeper

Figure 18–1 Extensions to the ATE and PTE



After you create a communication service, you can:

- Create a module that simulates the communication service and deploys it to the ATE. This provides application developers access to an Application Test Environment that simulates the behavior of the communication service with which you have extended Services Gatekeeper.
- Use the PTE to test the communication service from an application perspective. You do this by creating a module that acts as a client application to the new communication service and deploys it to the PTE.
- Use the PTE to test the communication service from a network perspective. You do this by creating a module that acts as a simulator to the new communication service and deploys it to the PTE.

The base for a virtual communication service project, a client project, and a network protocol simulator project is one or more WSDL files defining the application-facing interfaces exposed by the ATE or used by the PTE.

You use Platform Development Studio Eclipse tools to generate source code, deployment descriptors, and build files for modules that use these extension points.

Your implementation can use a set of interfaces to interact with the statistics and presentation facilities provided by the ATE and the PTE. You can interact with the network simulator map and add new elements to the map. Refer to the “All Classes” section of the *Services Gatekeeper Java API Reference* for the interface documentation..

Generating a Custom Module Project Using the PDS Wizard

See "Generating a Platform Test Environment Custom Module" for instructions about using the PDS Wizard to create a custom PTE module.

Understanding the Generated Project

The PDS (Eclipse) Wizard generates:

- A build file for the project: **build.xml**
- A deployment descriptor: **pte-extensions.xml**
- Depending on the type of project you generate, the project may also include:
  - Interface classes for a Virtual communication service.
  - Deployment class for a Virtual communication service.
  - Application client classes.
  - Network protocol simulator skeleton class.

The directory structure is described below:

```
<Eclipse_project>
+- build.xml
+- pte-extensions.xml
+- <Identifier given in Ecplise Wizard>
|  +- clients
|  +- simulators
|  +- vcs
|  |  +- <Identifier given in Eclipse Wizard>ModuleVCS.java
|  |  +- <Interface Name>Impl.java // One per interface
                                   // defined in the
                                   // Service WSDL files.
```

Understanding the Generated Project Build File

A generated Apache Ant build file is created in the directory:

Eclipse_project/**build.xml**

Where *Eclipse_project* is the directory where the project was generated by the PDS Wizard.

The build file defines the following targets:

- **generate**

  Generates source code.
- **compile**

  Compiles the generated source code.
- **jar**

  Packages the modules in JAR files.
- **clean**

Removes all generated artifacts.

- **dist**

  Generates the source code, compiles it, and generates JAR files that you can deploy.

Understanding the Generated Project Deployment Descriptor

A deployment descriptor is created when the project is generated. The deployment descriptor file name is **pte-extensions.xml**. It is created in the *Eclipse_project* directory, where *Eclipse_project* is the directory where the project is generated by the PDS Wizard.

The deployment descriptor describes how the virtual communication service is deployed in the Application Test Environment.

The deployment descriptor is an XML file with the following structure:

```
<module>
   <data>
      <parameter>
      </parameter>
   </data>
</module>
```

The **module** element has these attributes:

- **name**

  The name of the module given in the **Name** field in the PDS Wizard. The suffix **VCS** is added for virtual communication services. The suffix **Simulator** is added for simulators.

- **type**

  The type of module. **vcs** for a Virtual Communications Service module, **client** for a client module, and **sim** for a simulator module.

- **class**

  The fully qualified class name for the module. The first part of the package name is the name given in the **Package name** field in the PDS wizard. The name of the class is the name given in the **Name** field in the PDS wizard.

  For simulators, the last part of the package name is **.simulators.**. The class name has the suffix **Simulator**.

  For virtual communication services, the last part of the package name is **.vcs**. The class name has the suffix **VCS**.

- **version**

  The version of the module, given in the **Version** field in the PDS Wizard.

- **depends**

  The name of the module that this module depends on. In most cases, it is the Session module. Not used for Virtual Communications Services.

- **uiPane**l

  Describes in which panel in the user interface of the ATE or the PTE the module presents it’s user interface. For PTE clients is **clients**, for PTE simulators it is **simulators**, and for ATE virtual communication services it is **vcs**.

- `uiTabs`

  Describes in which tab in the GUI the module is presented. A comma-separated list describes the hierarchy.

The `<data>` element encapsulates zero or more parameter elements.

The `<parameter>` element describes fields in the user interface. It has the following attributes:

- **name**

  The label of the parameter in the user interface. Mandatory.

- **class**

  The class that defines the parameters in the user interface. Reflection is used to present the fields in the user interface. The parameters are presented as a hierarchy of description-only fields, and the parameters that have simple data types are presented with an input field. Optional.

- **default**

  The default value for the input field.

- **occurs**

  The number of occurrences of the parameter. Default value is 1.

Example 18–1 illustrates a deployment descriptor example for a client module.

Example 18–1 Example of a Client Module Deployment Descriptor

```
<module name="SendSmsModule"
        type="client"
        class="my.company.sm.clients.SendSmsModule"
        version="1.0"
        depends="session"
        uiPanel="client"
        uiTabs="Other,ate_pte_sm,sendSms"
        >

  <data>
    <parameter name="Parameters"
               class="my.company.sm.clients.SendSmsModuleData"
               occurs="1">

      <parameter name="facade"/>

      <parameter name="url"
                 default="http://${at.host}:${at.port}/SendSmsModule"/>

      <parameter name="vcsUrl"
                 default="http://${localhost}:13444/jaxws/SendSmsModule"/>

      <parameter name="restUrl"
                 default="http://${at.host}:${at.port}/rest/SendSmsModule"/>

      <parameter name="restVcsUrl"
                 default="http://${localhost}:13444/rest/SendSmsModule"/>

    </parameter>
  </data>
</module>
```

Example 18–2 illustrates a deployment descriptor example for a simulator module.

Example 18–2 Example of a Simulator Module Deployment Descriptor

```
<module name="Ate_pte_smSimulator"
        type="sim"
        class="my.company.sm.simulators.Ate_pte_smSimulator"
        version="1.0"
        uiPanel="simulator"
        uiTabs="ate_pte_sm"
        >

  <data>
    <parameter name="Parameters"
               class="parameterClassName"
               occurs="1">
    </parameter>
    -->
  </data>
</module>
```

Example 18–3 illustrates a deployment descriptor example for a virtual communication service.

Example 18–3 Example of a Virtual Communications Service Module Deployment Descriptor

```
<module name="Ate_pte_smVCS"
        type="vcs"
        class="my.company.sm.vcs.Ate_pte_smVCS"
        version="1.0"
        >
  </module>
```

Building and Deploying the Module

Run the Ant target **dist** to create a deployable module for the ATE or the PTE.

The deployable module is a JAR file named *Name***.jar**, where *Name* is the name of module given in the **Project Name** field in the PDS Wizard. The file is created in the **dist** subdirectory in the Eclipse project directory.

Deploying the module:

- ATE: Copy the deployable module to *SDK_Home***lib/modules**, where *SDK_Home* is the installation directory for the SDK.
- PTE: Copy the deployable module to *Service_Gatekeeper_Home***/ocsg_pds**/lib/modules, where *Service_Gatekeeper_Home* is the installation directory for Services Gatekeeper

Restart the ATE or PTE after deploying the new module.

Virtual Communication Service Module for the ATE

Skeletons of Java classes for a Virtual Communications Service for the ATE are created by the PDS Wizard. The classes are generated in the *Project_home***/src/***Package_hierarchy***/vcs** directory.

The class *Module_name*`VCS` deploys all the port implementations for the virtual communication service. *Module_name* is given in the PDS Wizard.

The class implements the **oracle.ocsg.pte.api.vcs.VCSModule** interface.

The methods are:

- **getName()**

  Returns the name of the module as a String. The name was given in the **Name** field in the PDS Wizard.

- **initialize(...)**

  Initializes the module. If the module is exposing MBeans, register them here.

- **start(...)**

  Deploys the module in the web services container. All generated implementation classes are deployed.

- **stop(...)**

  Undeploys the module from the web services container.

A separate class is generated for each port defined in the WSDL that the project defines. The classes are named *Port_name***Impl**, where *Port_name* is defined in the WSDL.

Each of the implementation classes defines the web service using the annotation `@WebService`. Example 18–4 gives an example of the `@WebService` annotation.

Example 18–4 Example of an @WebService Annotation

```
@WebService(name = "SendSms", targetNamespace =
"http://www.csapi.org/wsdl/parlayx/sms/send/v2_2/interface")
```

Each implementation class also defines a handler chain using the annotation **@HandlerChain**. This handler chain is necessary to leverage the security and SLA enforcement in the ATE and PTE. Example 18–5 illustrates the **@HandlerChain** annotation.

Example 18–5 Handler Chain Definition

```
@HandlerChain(file = "/vcs/VcsHandlerChains.xml")
```

Each method defined in the WSDL has a skeleton implementation. Each method is defined with the annotations:

- **@WebMethod**
- **@WebResult**
- **@RequestWrapper**
- **@ResponseWrapper**

Each method has an empty implementation where you add the custom code for the virtual communication service.

Client Module for the PTE

If you generated the project by using WSDL files, Platform Development Studio Wizard creates skeletons of Java classes for a client module for the Platform Test Environment.

The classes are generated in the *Project_home*/**src**/*Package_hierarchy*/**clients** directory.

For each method defined in the WDSL file the following classes are generated:

- *Method_Name***Module**
- **ResourceEndpoint**
- *Method_Name***ModuleData**

The class *Method_Name*`Module` deploys all the port implementations for the virtual communication service. *Method_Name* is given in the WSDL file.

The class extends o**racle.ocsg.pte.api.module.AbstractRestClientModule** and implements **oracle.ocsg.pte.api.module.CustomStatelessModule**.

The following methods are defined:

- execute(...)
- **prepare(...)**

The **execute(...)** method is called when you click the **Send** button for the client method in the Platform Test Environment GUI or, once, at the beginning of a duration test. The skeleton for the method retrieves the data to use in the method call from the **CustomModuleContext**. This context is passed in as a parameter to **execute(...)**. **CustomModule** is cast to the corresponding *Method_Name***ModuleData** object for the method.

The **prepare(...)** method fetches the URL from the field in the GUI and checks if the client shall use the SOAP interface or the RESTful interface by calling **isRestFacade()** on the **CustomModuleContext**. If the RESTful interface is used, the method **getRestClient(...)** defined by *Method_Name***ModuleData** object is fetched.

If the client shall use the SOAP facade, **JAXWSServiceFactory** is used to create the Web Service, a port is derived from the service and the local stub is set on the **RequestContext**.

The class **ResourceEndpoint** is instantiated if you are using the RESTful facade in the client. This class defines the methods:

- **getHttpMethod()**
- **getResourceURI()**

The method **getHttpMethod** shall return the HTTP request type as a `String`; POST or GET.

The method **getresourceURI()** shall return the URI to the RESTful method as a **String**.

The class *Method_Name***Module** is used to hold the data about the request.

Simulator Module for the PTE

Skeletons of a Java classes for a simulator module for the Platform Test Environment are created by the PDS Wizard. The class is generated in the directory *Project_home*/**src**/*Package_hierarchy*/**simulators**.

The class is named *Project_Name***Simulator**, where *Project_Name* is fetched from the **Name** field in the PDS Wizard.

The class implements the **oracle.ocsg.pte.api.module.CustomStatefulModule** interface. It defines these methods:

- **prepare(....)**
- **start(...)**
- **stop(...)**

All the methods sends in the `CustomModuleContext` as a parameter.

When the **prepare(....)** method is called, you can set up anything that is necessary for the simulator.

The **start(...)** method is called when you click the **Start** button in the GUI for the simulator.

The s**top(...)** method is called when you click the **Stop** button in the GUI for the simulator.

Virtual Communication Service Example

An example of a virtual communication service is provided in the *Middleware_home***/ocsg_pds/example/pte_vcs** directory.

Client Module Example

An example of a client module is provided in the *Middleware_home***/ocsg_pds/example/pte_module** directory.

Simulator Module Example

An example of a simulator module is provided in the *Middleware_home***/ocsg_pds/example/pte_module** directory.

Stateless and Stateful Modules

A stateless module implements the **oracle.ocsg.pte.api.module.CustomStatelessModule** interface. This interface defines the methods **prepare(...)** and **execute(...)**.

A stateful module implements the **oracle.ocsg.pte.api.module.CustomStatefulModule** interface. This interface defines the methods **prepare(...)**, **start(...)**, and **stop(...)**.

All these methods provide **oracle.ocsg.pte.api.module.CustomModuleContext** as a parameter.

Presenting Results

You present results in the Platform Test Environment GUI by implementing the **oracle.ocsg.pte.api.module.CustomResultsProvider<T>** interface.

The results are presented in a table with columns and rows. Specify the name, or title, of each column by returning the names when **java.lang.String[] getResultsColumns()** is called.

When **java.lang.Object getResultsContent(int column, T object)** is called, return the value of the object in the column with index column. The index is the same as the position in the results from **java.lang.String[] getResultsColumns()**.

The method **java.util.Map<java.lang.String,java.lang.Object> getResultsDetails(T object)** can be used to return additional result data than returned by **getResultsContent(...)**. Return null if there are no additional details.

When **java.util.List<T> getResultsObjects()** is called, you return a list of objects to be presented. Each object is presented in its' own row.

When **void clearResults()** is called, clear all result objects.

Presenting Statistics

You present statistics in the Platform Test Environment GUI by implementing the **oracle.ocsg.pte.api.module.CustomStatisticsProvider** interface.

To clear the statistics, call **void clearStatistics()**.

To return the statistics as a **java.util.Map<java.lang.String,java.lang.String>**, call **getStatistics()**. Each key in the map represents a specific statistics counter and the value is the value of the statistic counter.

Interacting With the Network Simulator Map

You use the simulator module and the virtual communication service to interact with the network simulator map and elements in the map.

Use **oracle.ocsg.pte.api.network.Network** to interact with the network simulator map.

Use **oracle.ocsg.pte.api.network.factory.NetworkFactory** to get a handle to the network simulator map:

```
oracle.ocsg.pte.api.network network =
NetworkFactory.getInstance().getNetwork();
```

Define a new network element by defining a class that extends the abstract class **oracle.ocsg.pte.api.network.element.AbstractNetworkElement**.

Create the network element using a class that implements the **oracle.ocsg.pte.api.network.factory.NetworkElementFactory<T extends NetworkElement>** interface. For example:

```
public class ExampleTruckFactory implements
NetworkElementFactory<ExampleTruck>
```

Register the network element with the network simulator map:

```
NetworkFactory.getInstance().register(ExampleTruck.class, new
ExampleTruckFactory());
```

Define the message that can be sent to and from a network element by defining a class that extends the abstract class **oracle.ocsg.pte.api.network.message.AbstractNetworkMessage**. Register the message, For example:

```
NetworkFactory.getInstance().register(ExampleMessage.class, new
ExampleMessageFactory());
```

There are a set of classes that describes messages that already defined. The classes are defined in the package **oracle.ocsg.pte.api.network.message**. The messages include SMS, MMS, and MLP update messages.

To send a message to the network use **sendMessage(NetworkSource source**, NetworkMessage message) in the **oracle.ocsg.pte.api.network.Network** interface. For example:

```
final ExampleMessage msg = new ExampleMessage(null, address, data);
```

```
NetworkFactory.getInstance().getNetwork().sendMessage(this, msg);
```

The network element class receives a call to **public boolean processMessage(NetworkMessage message** defined in the **oracle.ocsg.pte.api.network.element.NetworkElement** interface. For example:

```
public boolean processMessage(NetworkMessage message) throws Exception {
    if(message instanceof SmsMessage) {
      final SmsMessage sms = (SmsMessage) message;
      if(AbstractNetworkMessage.isAddressesMatching(sms.getDestinationAddress(),
getAddress())) {
        storeIncomingMessage(message);
        return true;
      }
    }
    return false;
  }
```

When the message is processed, the type of the message is checked and there is a check to see if the message is addressed to the network element by comparing the destination address in the message with the address of the element.

If you move around a network element on the map, the coordinates for it is updated. Use the methods **public double getLatitude()** and **public double getLongitude()** defined by the **NetworkElement** interface to get the location of the element.