Contents

1. Document Roadmap
   Document Scope and Audience .................................................. 1-1
   Guide to This Document ......................................................... 1-1
   Terminology .............................................................................. 1-2
   Related Documentation .............................................................. 1-5

2. Overview of the Extension Toolkit
   The Eclipse Plug-in ................................................................. 2-1
   The Sample Traffic Path ............................................................ 2-2
   The Testing Tools Suite ............................................................. 2-2

3. Using the Eclipse Plug-in for Extension Toolkit
   About the Eclipse plug-in for the Extension Toolkit ....................... 3-1
   Configure Eclipse ....................................................................... 3-2
      Prerequisites ........................................................................... 3-2
      Basic configuration of Eclipse environment .............................. 3-2
      Configuration of the Eclipse plug-in for the Extension Toolkit ...... 3-2
   Using the Eclipse plug-in ............................................................ 3-4
      Generating a traffic path project .............................................. 3-4
      Adding a plug-in to a traffic path project ................................. 3-6
      Removing a plug-in from a traffic path project ......................... 3-7
      Creating a plug-in project for an existing Network Gatekeeper application-facing interface .............................................. 3-8
4. Description of a Generated Project

Generated project ................................................................. 4-2
Traffic path project .............................................................. 4-2
Plug-in project for an existing interface ................................. 4-3
Build files and targets for a traffic path project ....................... 4-3
Deployment build files ......................................................... 4-3
Traffic path build file ........................................................... 4-6
Plug-in build file ................................................................. 4-6
Generated classes ................................................................. 4-7
ManagedPlugin ................................................................. 4-7
ManagedPlugin Skeleton ...................................................... 4-8
PluginImpl .............................................................. 4-12
PluginImpl skeleton ............................................................ 4-12
RequestFactory Skeleton ...................................................... 4-13
Prepare and deploy the extension ............................................. 4-14

5. The Traffic Path Example

About the traffic path example ............................................... 5-2
Sequence diagrams .............................................................. 5-4
Application to Network ........................................................ 5-4
Network to Application ......................................................... 5-6
Web Service interface definition ............................................. 5-7
Interface: SendData ............................................................. 5-7
Interface: NotificationManager .............................................. 5-8
Interface: NotificationListener .............................................. 5-9
Web Services Layer ............................................................. 5-10
Service Capability Layer ....................................................... 5-10
Plug-in Layer ................................................................. 5-11
6. Policy

Overview ................................................................. 6-1
Policy Request Data ................................................... 6-2
Adding a New Rule ..................................................... 6-4
   Mapping PolicyRequest Data ....................................... 6-5
   Creating a New Rule File by Extending an Existing File: an Example ........................................... 6-8
      List the Current Services’ Rule Files ................................. 6-9
      Select the Service Whose Rule File You Wish to Extend ........................................... 6-9
      Add a New Extended Rule ........................................... 6-21
      Load the New Rule File ........................................... 6-21
   Using RequestContext Parameters Defined in Service Level Agreements ........................................... 6-22
Extending Service Level Agreements ..................................... 6-23
   Update the SLA Schema ........................................... 6-23
   Load the new SLA schema into the Policy Service ........................................... 6-24
Update and reload the rule files ........................................... 6-24
Update the appropriate SLAs ............................................. 6-25
Load the updated SLAs ..................................................... 6-25

7. Core and Core Utilities

Core and Core Utilities ................................................... 7-2
Core ............................................................................... 7-2
  Interface: ServiceDeployable ........................................... 7-2
  Interface: ServiceDeployableExt ....................................... 7-2
  Interface: ServiceContext and SLEEContext ...................... 7-3
Core utilities ................................................................. 7-3
  Class: InstanceFactory .................................................. 7-4
  Class: ClusterHelper .................................................... 7-4
  Service: Event Channel Service ....................................... 7-5
  Service: Time Service .................................................. 7-6

Other APIs ......................................................................... 7-7
Plug-in ................................................................. 7-7
  Interface: Plugin .......................................................... 7-7
  Class: ServiceType ....................................................... 7-7
  Interface: ContextMapperInfo ........................................ 7-8
  Interface: RequestContext ............................................... 7-9
  Class: ManagedPlugin ................................................... 7-9
  Class: AbstractManagedPlugin ....................................... 7-9
  Class: RequestFactory .................................................. 7-9
  Class: RequestInfo ....................................................... 7-12
Management ................................................................. 7-13
EDR ........................................................................... 7-13
Service Correlation ......................................................... 7-13
Interface: ExternalInvocation ............................................. 7-14
Class: ExternalInvocatorFactory ....................................... 7-14
Class: ServiceCorrelation ............................................... 7-14
Implementing the ExternalInvocation Interface .................. 7-14
Parameter Tunneling ...................................................... 7-16
Storage Services .......................................................... 7-16
  ConfigurationStore .................................................... 7-16
  Interfaces ............................................................... 7-17
  StorageService ........................................................ 7-20
  Store configuration file ............................................. 7-23
    <store> ............................................................. 7-25
    <db_table> ........................................................ 7-25
    <query> ............................................................. 7-26
    <provider-mapping> ................................................. 7-29
    <providers> ........................................................ 7-30
Using shared libraries ................................................. 7-30

8. Annotations, EDRs, Alarms, and CDRs
   About aspects and annotations .................................... 8-2
   How aspects are applied ........................................... 8-2
   Context aspect ....................................................... 8-3
   EDR Generation ....................................................... 8-7
     Exception scenarios ............................................... 8-8
     Adding data to the RequestContext ............................. 8-10
     Using translators ................................................ 8-10
     Trigger an EDR programmatically ............................... 8-12
   EDR Content .......................................................... 8-13
     Using send lists ................................................... 8-19
9. Making Traffic Paths Manageable

Overview ................................................................. 9-1
Create Standard JMX MBeans .............................................. 9-2
Create a Java Interface .................................................. 9-2
Implement the Interface .................................................. 9-3
Register the MBeans with the Runtime MBean Server ............... 9-6
Use the Configuration Store to Persist Values ......................... 9-7
10. Testing Your Traffic Path

Overview ................................................................. 10-1
How It Works ............................................................ 10-3
  Application-Initiated (MT) ........................................ 10-3
  Network-Initiated (MO) .......................................... 10-4
Setting up the Testing Tools ....................................... 10-5
  Prepare to generate the Web Services client stub ............ 10-7
  Prepare to generate the callback Web Service, if necessary .... 10-8
  Implement the TestAgentResult object ........................ 10-9
  Deploy the callback Web Services and the Listener Test Agent to a separate ear file . 10-10
  Create the test case .............................................. 10-12
  Implement the notification matching functionality in the test case .............. 10-13
  Implement the EDR matching functionality in the test case ...................... 10-13
  Edit the properties files, as necessary ........................ 10-15
Overview of Testing Tools Component Packages ............... 10-18

11. Adapting 2.2 Style Traffic Paths

Adapting a 2.2 Style Plug-in for a 3.0 Backwards Compatible Traffic Path ........ 11-1
Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface 11-3
  Merge the 2.2 Web Services implementation ....................... 11-4
  Implement the Managed Plugin class ............................. 11-5
Adapting a 2.2 Style Traffic Path Based on a non-3.0 Supported, Web Services-based, Application Facing Interface ......................... 11-8

12. SLA Parameters and Policy enforcement for new plug-ins

13. Checklist
CHAPTER 1

Document Roadmap

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

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

Document Scope and Audience

This document describes the WebLogic Network Gatekeeper Extension Toolkit, a framework for creating and testing new extension traffic paths and network plug-ins and adapting 2.2 style extensions. The intended audience of this document consists of system integrators and field engineers who need to extend the out-of-the-box functionality of WebLogic Network Gatekeeper.

Guide to This Document

The document contains the following chapters:

Chapter 1, “Document Roadmap”: This chapter

Chapter 2, “Overview of the Extension Toolkit”: A high level description of the capabilities of the Toolkit
Chapter 3, “Using the Eclipse Plug-in for Extension Toolkit”: Setting up the Extension Toolkit Eclipse plug-in and using the wizard to generate extension projects

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

Chapter 5, “The Traffic Path Example”: A description of the traffic path example supplied with the Extension Toolkit. This is a buildable and runnable sample

Chapter 6, “Policy”: Using and extending Policy mechanisms in extension traffic paths

Chapter 7, “Core and Core Utilities”: Accessing Network Gatekeeper’s Core functionality

Chapter 8, “Annotations, EDRs, Alarms, and CDRs”: Creating EDRs, CDRs, and Alarms

Chapter 9, “Making Traffic Paths Manageable”: Rendering extension traffic paths manageable by the Network Gatekeeper Console extension or other management tools

Chapter 10, “Testing Your Traffic Path”: Using the testing tools framework

Chapter 11, “Adapting 2.2 Style Traffic Paths”: Adapting 2.2 style extensions to run in the 3.0 environment

Chapter 12, “SLA Parameters and Policy enforcement for new plug-ins”: SLA parameters and policy parameters to use for plug-ins for Multimedia Messaging, Payment, and Terminal Location.

Chapter 13, “Checklist”: Checklist to be used when creating extensions.

Terminology

The following terms and acronyms are used in this document:

- Account—A registered application or service provider, associated with an SLA
- Account group—Multiple registered service providers or services which share a common SLA
- Administrative User—Someone who has privileges on the Network Gatekeeper management tool. This person has an administrative user name and password
- Alarm—The result of an unexpected event in the system, often requiring corrective action
- API—Application Programming Interface
- Application—A TCP/IP based, telecom-enabled program accessed from either a telephony terminal or a computer
• Application-facing Interface—The Application Services Provider facing interface

• Application Service Provider—An organization offering application services to end users through a telephony network

• AS—Application Server

• Application User—An Application Service Provider from the perspective of internal Network Gatekeeper administration. An Application User has a user name and password

• CBC—Content Based Charging

• End User—The ultimate consumer of the services that an application provides. An end user can be the same as the network subscriber, as in the case of a prepaid service or they can be a non-subscriber, as in the case of an automated mail-ordering application where the subscriber is the mail-order company and the end user is a customer to this company

• Enterprise Operator —See Service Provider

• Event—A trackable, expected occurrence in the system, of interest to the operator

• HA —High Availability

• HTML—Hypertext Markup Language

• IP—Internet Protocol

• JDBC—Java Database Connectivity, the Java API for database access

• Location Uncertainty Shape—A geometric shape surrounding a base point specified in terms of latitude and longitude. It is used in terminal location

• MAP—Mobile Application Part

• Mated Pair—Two physically distributed installations of WebLogic Network Gatekeeper nodes sharing a subset of data allowing for high availability between the nodes

• MM7—A multimedia messaging protocol specified by 3GPP

• MPP—Mobile Positioning Protocol

• Network Plug-in—The WebLogic Network Gatekeeper module that implements the interface to a network node or OSA/Parlay SCS through a specific protocol

• NS—Network Simulator

• OAM —Operation, Administration, and Maintenance
• Operator—The party that manages the Network Gatekeeper. Usually the network operator

• OSA—Open Service Access

• PAP—Push Access Protocol

• Plug-in—See Network Plug-in

• Plug-in Manager—The Network Gatekeeper module charged with routing an application-initiated request to the appropriate network plug-in

• Policy Engine—The Network Gatekeeper module charged with evaluating whether a particular request is acceptable under the rules

• Quotas—Access rule based on an aggregated number of invocations. See also Rates

• Rates—Access rule based on allowable invocations per time period. See also Quotas

• Rules—The customizable set of criteria - based on SLAs and operator-desired additions - according to which requests are evaluated

• SCF—Service Capability Function or Service Control Function, in the OSA/Parlay sense.

• SCS—Service Capability Server, in the OSA/Parlay sense. WebLogic Network Gatekeeper can interact with these on its network-facing interface

• Service Capability—Support for a specific kind of traffic within WebLogic Network Gatekeeper. Defined in terms of traffic paths

• Service Provider—See Application Service Provider

• SIP—Session Initiation Protocol

• SLA—Service Level Agreement

• SMPP—Short Message Peer-to-Peer Protocol

• SMS—Short Message Service

• SMSC—Short Message Service Centre

• SNMP—Simple Network Management Protocol

• SOAP—Simple Object Access Protocol

• SPA—Service Provider APIs

• SS7—Signalling System 7
• Subscriber—A person or organization that signs up for access to an application. The subscriber is charged for the application service usage. See End User

• SQL—Structured Query Language

• TCP—Transmission Control Protocol

• Traffic Path—The data flow of a particular request through WebLogic Network Gatekeeper. Different Service Capabilities use different traffic paths

• USSD—Unstructured Supplementary Service Data

• VAS—Value Added Service

• VLAN—Virtual Local Area Network

• VPN—Virtual Private Network

• WebLogic Network Gatekeeper Core—The container that holds the Core Utilities

• WebLogic Network Gatekeeper Core Utilities—A set of utilities common to all traffic paths

• WSDL—Web Services Definition Language

• XML—Extended Markup Language

Related Documentation

This extension toolkit developer’s guide is part of a set of documentation for Network Gatekeeper itself. These documents include:

• Architectural Overview

• System Administrator’s Guide

• Handling Alarms

• Installation Guide

• Integration Guidelines for Partner Relationship Management

• Managing Service Providers and Applications

• Statement of Compliance

• Application Development Guide
Document Roadmap

- SDK User Guide
- System Backup and Restoration Guide
- Licensing
- Traffic Path Reference
Overview of the Extension Toolkit

Networks change. Existing functionality is parsed in new ways to support new features. New nodes with new or modified abilities are added. Because of WebLogic Network Gatekeeper’s highly modular design, exposing these new features to partners is a straightforward proposition. The Network Gatekeeper Extension Toolkit is an environment in which much of the mechanics of creating extensions is taken care of, allowing the extension developer to focus on only those parts of the system that correspond directly to their specific needs. The Toolkit consists of three main parts:

- The Eclipse Plug-in
- The Sample Traffic Path
- The Testing Tools Suite

The Eclipse Plug-in

At the core of the Extension Toolkit is an Eclipse plug-in that creates projects based on the responses that the developer makes to an Extension Wizard. The developer supplies some basic naming information and the location of a WSDL for each application facing interface that the traffic path is meant to support, and the Wizard generates either a complete traffic path project, or a network plug-in only project. For more information on setting up the Eclipse Plug-in and running the Wizard, see Chapter 3, “Using the Eclipse Plug-in for Extension Toolkit.” To see an example of a generated project, see Chapter 4, “Description of a Generated Project.” To get an understanding of the Network Gatekeeper features with which your traffic path will interact, see

**The Sample Traffic Path**

To give you a concrete sense of the task of generating a new traffic path, the Extension Toolkit contains an entire Sample Traffic Path, which is buildable and runnable. Based on a very simple Web Service interface and an equally simple model of an underlying network protocol, this traffic path demonstrates the entire range of tasks that you will encounter in creating your own traffic path. For more information, see Chapter 5, “The Traffic Path Example.”

**The Testing Tools Suite**

Finally, to simplify the testing of your traffic path, the Extension Toolkit provides a simple testing tools framework, which provides an abstract base class, `WlngTestCase`, which contains mechanisms for running tests for both application and network-initiated traffic and for checking EDR/CDR generation; a component library that supports the functionality of the test case; a properties based model for loading configuration information at test run-time; and a set of fully functional test cases that work with the Sample Traffic Path. For more information, see Chapter 10, “Testing Your Traffic Path.”
CHAPTER 3

Using the Eclipse Plug-in for Extension Toolkit

This section describes using the Eclipse plug-in for the Extension Toolkit to generate extension projects:

- “About the Eclipse plug-in for the Extension Toolkit” on page 3-1
- “Configure Eclipse” on page 3-2
- “Using the Eclipse plug-in” on page 3-4
  - “Generating a traffic path project” on page 3-4
  - “Adding a plug-in to a traffic path project” on page 3-6
  - “Removing a plug-in from a traffic path project” on page 3-7
  - “Removing a plug-in from a traffic path project” on page 3-7
  - “Creating a plug-in project for an existing Network Gatekeeper application-facing interface” on page 3-8

About the Eclipse plug-in for the Extension Toolkit

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

- Traffic paths
- Network protocol plug-ins for existing application-facing interfaces
Configure Eclipse

Prerequisites

- Eclipse 3.3 must be installed
- Ant 1.6.5 must be installed

Basic configuration of Eclipse environment

To do the basic configuration of the Eclipse environment:

1. Make sure that Eclipse is started using the correct Java environment by running:
   
   ```
   BEAHOME\wlng300\server\bin\setWLSEnv.cmd
   
   or
   
   BEAHOME/wlng300/server/bin/setWLSEnv.sh
   ```

2. Start Eclipse

3. Open the Preferences window, **Window→Preferences**...

   a. In **Java→Installed JREs**, make sure that the JRE used is the JRE installed with the Network Gatekeeper. This is installed in `BEAHOME/<jdk version>/jre`

   b. In **Ant→runtime**, make sure Ant Home is set to the directory in which you have installed Ant.

Configuration of the Eclipse plug-in for the Extension Toolkit

To configure the Eclipse plug-in for the Extension Toolkit, starting in Eclipse:

1. Open the Preferences window, **Window→Preferences**...
2. In **WLNG Extension Toolkit** &gt; **Basic Configuration**, configure the following:

   **WLNG Home Directory** The directory of the Network Gatekeeper installation. This provides references to WebLogic Server APIs. In the default installation, this would be 
   
   `<beahome>/wlng300/`.

   **WLNG Extension Toolkit Installation Directory** The directory of the Network Gatekeeper Extension Toolkit installation. This provides references to Network Gatekeeper APIs, extension points and third party APIs. In the default installation, this would be 
   
   `<beahome>/wlng_et300`.

   **WLNG Domain Home Directory** The directory of the Network Gatekeeper domain. In the default installation, this would be 
   
   `<beahome>/user_projects/domains/wlng-access-network-domain`.

   **JDK Installation Directory** The JDK installation directory.

   **ANT Installation Directory** The ANT installation directory.

   Make sure the check-box **Turn on Plug-in Debug** is checked.

3. Open the Preferences window, **Window** &gt; **Preferences**...

4. In **WLNG Extension Toolkit** &gt; **WebLogic Server Deployment**, configure the following in order to support automatic deployment of your extension from within Eclipse:

   **WebLogic Server Administration URL** URL of the administration server in the Network Gatekeeper Domain.

   **Access Tier Deployment Target Name** The target name (cluster name) on which the access tier ear file will be deployed. In a default installation this is **WLNG_AT_Cluster**.

   **Network Tier Deployment Target Name** The target name (cluster name) on which the network tier ear file will be deployed. In a default installation this is **WLNG_NT_Cluster**.

   **Username** Administrator user name.

   **Password** Administrator password.

   **Note:** In the case of a combined installation, where the administration server, the access tier, and the network tier are executing in the same server, both target names should be the same as the administration server, for example **AdminServer**.
Using the Eclipse plug-in

Generating a traffic path project

A traffic path project is based on a WSDL file and a set of attributes given when running the Traffic Path Project wizard supplied by Network Gatekeeper Extension Toolkit.

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

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

To generate a traffic path project:

1. In Eclipse, choose File→New Project.
   - This opens the New Project window.

<table>
<thead>
<tr>
<th>In this window...</th>
<th>Perform the following action...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Wizard</td>
<td>Make sure WebLogic Network Gatekeeper-&gt;Traffic Path Project is selected.</td>
</tr>
<tr>
<td></td>
<td>Click Next to proceed. You may cancel the wizard at any time by clicking Exit. You may go back to a previous window by clicking Previous.</td>
</tr>
<tr>
<td>Create a traffic path</td>
<td>Enter a Project Name and choose a location for your project. Click Next to continue.</td>
</tr>
</tbody>
</table>

In this window... Perform the following action...
Define the traffic path

In the Configure Service WSDL Files group, add the WSDL files that define the Web Services that define the service to be implemented by the new traffic path.

In the Configure Callback WSDL Files group, for each WSDL file that includes the service definition for the new traffic path:

- Click Add, browse to the file, select it, and click OK.

In the Traffic Path Properties group, configure the following:

- **Traffic Path Identifier** - Ties together a collection of Web Services. Will be a part of the names of the generated war and jar files and the service type for the traffic path:
  - `<traffic path identifier>.war` and `<traffic path identifier>_callback.jar`

- **Java Class Package Name** - The package names to be used.

- **Web Services Context path** - The context path for the Web Service.

- **Support SOAP Attachment** Check this box if SOAP attachments will be used in the traffic path.

Click Next to continue.

To create a plug-in for the traffic path, click Next.

If only the Web Services and service capability parts of the traffic path should be generated, click Finish to start the code generation process.

---

<table>
<thead>
<tr>
<th>In this window...</th>
<th>Perform the following action...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define the traffic path</td>
<td>In the Configure Service WSDL Files group, add the WSDL files that define the Web Services that define the service to be implemented by the new traffic path. In the Configure Callback WSDL Files group, for each WSDL file that includes the service definition for the new traffic path: Click Add, browse to the file, select it, and click OK. In the Traffic Path Properties group, configure the following: <strong>Traffic Path Identifier</strong> - Ties together a collection of Web Services. Will be a part of the names of the generated war and jar files and the service type for the traffic path: <code>&lt;traffic path identifier&gt;.war</code> and <code>&lt;traffic path identifier&gt;_callback.jar</code> <strong>Java Class Package Name</strong> - The package names to be used. <strong>Web Services Context path</strong> - The context path for the Web Service. <strong>Support SOAP Attachment</strong> Check this box if SOAP attachments will be used in the traffic path. Click Next to continue. To create a plug-in for the traffic path, click Next. If only the Web Services and service capability parts of the traffic path should be generated, click Finish to start the code generation process.</td>
</tr>
</tbody>
</table>
Adding a plug-in to a traffic path project

To add a plug-in to a traffic path project:

1. In the Eclipse package explorer, right-click the project for the traffic path project, and choose Properties.
   
   This opens the Properties window for the traffic path project.
Removing a plug-in from a traffic path project

To remove a plug-in from a traffic path project:

1. In the Eclipse package explorer, right-click the project for the traffic path project, and choose Properties.
This opens the Properties Window for the traffic path project.

<table>
<thead>
<tr>
<th>In this window...</th>
<th>Perform the following action...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PluginConfiguration</strong></td>
<td>A list of plug-ins defined for the traffic path project is displayed. For each plug-in to be removed from the traffic path project: Select the plug-in to be removed and click <strong>Delete</strong>... The plug-in definitions are removed from the list. Click <strong>Apply</strong> to remove the plug-in part(s) from the traffic path project. <strong>Warning</strong>: This removes all parts of the project, including any manually edited or added files. Click <strong>Restore Defaults</strong> to restore the plug-in definition list.</td>
</tr>
</tbody>
</table>

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

**Creating a plug-in project for an existing Network Gatekeeper application-facing interface**

To create a plug-in project for an existing Network Gatekeeper application-facing interface:

1. Start Eclipse.

2. Create a new project by choosing **File**—>**New Project**. The **New Project** window opens.

3. In the **Select Wizard** page, choose **WebLogic Network Gatekeeper**—>**Plugin Only Project** and click **Next**.
4. This opens the **Create a plugin only project** wizard.

<table>
<thead>
<tr>
<th>In this window...</th>
<th>Perform the following action...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create a plugin only project</strong></td>
<td><strong>Project name</strong> The project name. From the <strong>Please select a traffic path</strong> drop-down list, select the application-facing interface which your plug-in will use. If the default project location should be used, check <strong>Use default location</strong>. If a custom location should be used for the project, <strong>uncheck</strong> the Use default location, and browse to the <strong>non-default location</strong>. Click <strong>Next</strong> to proceed. You may cancel the wizard at any time by clicking <strong>Exit</strong>. You may go back to a previous window by clicking <strong>Previous</strong>.</td>
</tr>
</tbody>
</table>

| **Define the traffic plugin** | A list of plug-ins to be defined for the plug-in project is displayed. For each plug-in to be created in the traffic path project: Click **Add...** This opens a pop-up window with the following fields: **Protocol** An identifier for the network protocol the plug-in implements. Used as a part of the generated jar files. **Package Name** The package name to be used. **Company Name** Company name. Click **OK**. The plug in definitions are added to the list of plug-ins. Click **Finish** to start the project generation for the plug-in(s). |
Description of a Generated Project

The section describes a project generated from the Eclipse plug-in:

- “Generated project” on page 4-2  
  - “Traffic path project” on page 4-2  
  - “Plug-in project for an existing interface” on page 4-3
- “Build files and targets for a traffic path project” on page 4-3  
  - “Deployment build files” on page 4-3  
  - “Traffic path build file” on page 4-6  
  - “Plug-in build file” on page 4-6
- “Generated classes” on page 4-7  
  - “ManagedPlugin” on page 4-7  
  - “PluginImpl” on page 4-12  
  - “RequestFactory Skeleton” on page 4-13
- “Prepare and deploy the extension” on page 4-14
Description of a Generated Project

Generated project

Traffic path project

When generating a traffic path project the following directory structure is created under the top-level directory. The base directory is the directory given in the Eclipse wizard, \(<Project Name>\)/\(<Traffic Path Identifier>\). It contains two subdirectories:

- traffic_path
- plugins

The directory traffic_path contains:

- build.properties, the properties file for the build file.
- build.xml, the build file for the non-plug-in parts of the project. Contains targets for regenerating the traffic path archive files (jar and war files) and a target to generate a new request factory skeleton.

The directory dist, which contains generated code, including the callback EJB (.jar) and the Web Service implementation (.war). These are deployed in the Access Tier in Network Gatekeeper.

The directory plugins contains one subdirectory for each plug-in belonging to the traffic path. The name of the subdirectory is the same as the name chosen for \(<Protocol>\) in the eclipse wizard for the plug-in.

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

- base.xml, definitions used by the build file.
- build.properties, build properties file for a plug-in.
- build.xml, the build file for the plug-in with targets to compile, jar and instrument (weave aspects into) the plug-in and a target to generate a new plug-in skeleton.

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

- 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.
- config, the directory that includes an instancemap that will be used by the InstanceFactory in the generated Managed Plug-in to create instances for the plug-in interface implementations.
Build files and targets for a traffic path project

- src, the directory that contains the generated plug-in skeleton.

Plug-in project for an existing interface

When generating a plug-in project the following directory structure is created under the top-level directory. The base directory depends on the type of traffic path the plug-in belongs to, such as, for example, SMS, or Multimedia Messaging.

The base directory contains the directory `plugins`, which contains subdirectories for each protocol that was added. The names of the directories are the same as the name chosen for the `<Protocol>` field in the Eclipse wizard.

Each of the protocol-specific subdirectories contains the following:

- `base.xml`, generated properties file for the plug-in build file.
- `build.properties`, build properties file for the plug-in.
- `build.xml`, the build file for the plug-in with targets to compile, jar and instrument (weave aspects into) the plug-in and a target to generate a new plug-in skeleton.

Each of the protocol-specific subdirectories also includes the directories:

- `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.
- `config`, the directory that includes an instancemap that will be used by the InstanceFactory in the generated Managed plug-in to create instances for the plug-in interface implementations.
- src, the directory that contains the generated plug-in skeleton.

Build files and targets for a traffic path project

Deployment build files

The build file for deployment of plug-ins and traffic paths contains the following targets:

- all: Calls the clean, unpackAll, copyAllExtensions, and updateConfig targets.
- packDeployAll: Calls the packAll and deployAll targets.
- packDeployAT: Calls the packAT and deployAT targets.
Description of a Generated Project

- **packDeployNT**: Calls the packNT and deployNT targets.

- **unpackAll**: Calls the unpackAT and unpackNT targets.

- **unpackAT**: Unzips the AT ear file from the WLNG domain into a staging directory.

- **unpackNT**: Unzips the NT ear file from the WLNG domain into a staging directory.

- **copyAllExtensions**: Calls the copyATExtensions and CopyNTExtensions targets.

- **copyATExtensions**: Copies the AT extension war and jar files to the correct place in staging directory.

- **copyNTExtensions**: Copies the NT extension jar files to the correct place in staging directory.

- **updateConfig**: Updates the NT ear META-INF/weblogic-extension.xml file with information about the plug-in jar files.

- **packAll**: Calls the packAT and packNT targets.

- **packAT**: Zips the AT part of the staging directory to an AT ear file and copies that file to the WLNG domain. The first time this is done the existing ear file is moved to wlng_at.ear.old. If that file exists, it is NOT overwritten, so that you always have a copy of the original ear file.

- **packNT**: Zips the NT part of the staging directory to an NT ear file and copies that file to the WLNG domain. The first time this is done the existing ear file is moved to wlng_at.ear.old. If that file exists, it is NOT overwritten, so that you always have a copy of the original ear file.

- **generate-mib**: Generates an SNMP MIB file based on the alarms defined in the NT ear APP-INF/classes/edr/alarm.xml file. Any alarms for an extension must have been added to alarm.xml in order for them to be added to the MIB file. The MIB file is named BEA-WLNG-MIB and is created in the directory $DOMAIN_HOME/snmp/BEA-WLNG-MIB. The NT ear file must have been unpacked using the target unpackNT prior to executing this target.

- **deployAll**: Calls the deployAT and deployNT targets.

- **deployAT**: Deploys the AT ear file on the server using the URL specified in the build.properties file.

- **deployNT**: Deploys the NT ear file on the server using the URL specified in the build.properties file.
Note: The targets attribute for the task wldeploy in the deployAT and deployNT targets must be updated if the domain configuration the target is not a Basic Weblogic Network Gatekeeper Domain with AT, NT and AdminServer in one single domain. Define the attribute targets to be the cluster name for the access tier and the network tier, respectively. Default values are WLNG_AT_Cluster and WLNG_NT_Cluster. These values may have been changed during domain configuration. See example below.

Listing 4-1 Targets that must be changed for the target deploy

```xml
<target name='deployAT'>
    <wldeploy
        action='deploy' verbose='true' debug='true'
        name='${at.deploy.name}'
        source='${wlng.at.ear}'
        user='${wlng.admin.username}' password='${wlng.admin.password}'
        adminurl='${wlng.admin.url}'
        targets='${at.deploy.target}' />
</target>

<target name='deployNT'>
    <wldeploy
        action='deploy' verbose='true' debug='true'
        name='${nt.deploy.name}'
        source='${wlng.nt.ear}'
        user='${wlng.admin.username}' password='${wlng.admin.password}'
        adminurl='${wlng.admin.url}'
        targets='${nt.deploy.target}' />
</target>
```

The at.deploy.target and nt.deploy.target needs to be updated. These attributes are defined in the file build.properties.
Description of a Generated Project

Traffic path build file
The build file for the traffic path contains the following targets:

- **dist**: Calls the **clean** and **wsGen** targets.
- **clean**: Deletes the **dist** directory.
- **wsGen**: Regenerates the application facing interface and includes the Request Factories from the **src** directory.

When building a **dist**, the following is created in \(<Project Name>\)/<Traffic Path>/dist:

- \(<Traffic Path Identifier>.war\), The web archive for the traffic path web service.
- \(<Traffic Path Identifier>_service.jar\), The service EJB archive which also includes the plug-in interface files.
- \(<Traffic Path Identifier>_callback.jar\), The callback EJB archive.
- \(<Traffic Path Identifier>_callback_client.jar\), The archive for client stubs used for the callback EJBs.

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

- **dist**: Calls the **clean**, **init**, **compile**, **jar** and **instrument-individual** targets.
- **clean**: Deletes the **build** and **dist** directories.
- **init**: Creates the **build** directory.
- **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-individual**: Weaves the aspects that should apply into the plug-in.
- **pluginSkelGen**: Regenerates the plug-in skeleton and puts it under the **dist** directory.

When building a **dist**, the following is created in \(<Project Name>\)/<Traffic Path Identifier>/plugins/<Protocol>/dist:

- \(j_<Traffic Path Identifier>_<protocol>.jar\), The archive for the plug-in.
**Generated classes**

The generated classes are listed below.

**Figure 4-1** Class diagram of the generated plug-in skeleton code (in blue) and its relations to other classes.

---

**ManagedPlugin**

The actual class name is `<Traffic path identifier>ManagedPlugin`. It extends the abstract class `com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin`. This class manages the life-cycle for the plug-in, including implementing the necessary interfaces that make the plug-in deployable in Network Gatekeeper. It is also responsible for registering the north interfaces with the Plug-in Manager. It implements all the necessary interfaces to make the plug-in deployable in Network Gatekeeper. At startup time it uses the InstanceFactory to create one instance of each plug-in if implementation 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 can be found under the `resource` directory.
ManagedPlugin Skeleton

Below is an example of a ManagedPlugin skeleton.

Listing 4-2  Example of ManagedPlugin implementation

```java
package com.bea.wlcp.wlng.smstest.protocoltype.managedplugin;
import com.bea.wlcp.wlng.api.plugin.PluginNorth;
import com.bea.wlcp.wlng.api.plugin.ServiceType;
import com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin;
import com.incomit.slee.ServiceContext;
import com.incomit.slee.ServiceDeploymentException;
import com.bea.wlcp.wlng.api.util.InstanceFactory;
import org.apache.log4j.Logger;
import com.bea.wlcp.wlng.smstest.plugin.SendSmsPlugin;
import com.bea.wlcp.wlng.smstest.plugin.ReceiveSmsPlugin;
import com.bea.wlcp.wlng.smstest.plugin.SmsNotificationManagerPlugin;
import com.bea.wlcp.wlng.smstest.protocoltype.north.SendSmsPluginImpl;
import com.bea.wlcp.wlng.smstest.protocoltype.north.ReceiveSmsPluginImpl;
import com.bea.wlcp.wlng.smstest.protocoltype.north.SmsNotificationManagerPluginImpl;
public class Px21SmsManagedPlugin extends AbstractManagedPlugin {

    /** The Log4J logger */
    private static final Logger logger =
        Logger.getLogger(Px21SmsManagedPlugin.class);
    /** Addressing schemes supported by the plugin. */
    private static final String[] SUPPORTED_SCHEMES = new String[]{"tel");

    private SendSmsPlugin northInterface1;
```
private ReceiveSmsPlugin northInterface2;
private SmsNotificationManagerPlugin northInterface3;

/**
 * Instantiates the plugin with "tel:" being the only supported addressing
 * scheme.
 */

public Px21SmsManagedPlugin() {
    super(SUPPORTED_SCHEMES);
}

/* (non-Javadoc)
 * @see com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin#doStarted()
 */
public void doStarted() throws ServiceDeploymentException {
    try {
        northInterface1 = (SendSmsPlugin)
            InstanceFactory.getInstance().getImplementation(SendSmsPluginImpl.class);
        northInterface2 = (ReceiveSmsPlugin)
            InstanceFactory.getInstance().getImplementation(ReceiveSmsPluginImpl.class);
        northInterface3 = (SmsNotificationManagerPlugin)
            InstanceFactory.getInstance().getImplementation(SmsNotificationManagerPluginImpl.class);

    } catch (Exception e) {
        rethrowServiceDeploymentException(e);
    }
}
/* (non-Javadoc)
 * @see com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin#doStopped()
 */

public void doStopped() throws ServiceDeploymentException {
  try {
    registerNorthInterface(SendSmsPlugin.class, northInterface1);
    registerNorthInterface(ReceiveSmsPlugin.class, northInterface2);
    registerNorthInterface(SmsNotificationManagerPlugin.class, northInterface3);
    active = true;
  } catch (Exception e) {
    rethrowServiceDeploymentException(e);
  }
}

/* (non-Javadoc)
 * @see com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin#doActivated()
 */

public void doActivated() throws ServiceDeploymentException {
  try {
    registerNorthInterface(SendSmsPlugin.class, northInterface1);
    registerNorthInterface(ReceiveSmsPlugin.class, northInterface2);
    registerNorthInterface(SmsNotificationManagerPlugin.class, northInterface3);
    active = true;
  } catch (Exception e) {
    rethrowServiceDeploymentException(e);
  }
}

/* (non-Javadoc)
 * @see com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin#doDeactivated()
 */

public void doDeactivated() throws ServiceDeploymentException {
}
try {
    active = false;
} catch (Exception e) {
    rethrowServiceDeploymentException(e);
}

/* (non-Javadoc)
 * @see com.bea.wlcp.wlng.api.plugin.common.AbstractManagedPlugin#isActive()
 */

public boolean isActive() {
    return active;
}

public ServiceType getServiceType() {
    return com.bea.wlcp.wlng.smstest.servicetype.Px21SmsServiceType.type;
}

/**
 * Rethrows a ServiceDeploymentException given another exception. The exception
 * is wrapped into a ServiceDeploymentException.
 *
 * @param e The exception to wrap
 * @throws ServiceDeploymentException
 */
Description of a Generated Project

private void rethrowServiceDeploymentException(Exception e) throws ServiceDeploymentException {
    throw new ServiceDeploymentException(e.toString(),
            svcCtx.getInstanceName(),
            0);
}

PluginImpl

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

The following files will be generated in the plug-in dist directory under plugin_skel/src:

- `<web service interface name>PluginImpl`: 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.

PluginImpl skeleton

Below is an example of a Plug-in skeleton.

Listing 4-3  Example of Plug-in interface implementation for ReceiveSms

```java
package com.bea.wlcp.wlng.smstest.protocoltype.north;
import com.bea.wlcp.wlng.api.edr.Edr;
import com.bea.wlcp.wlng.smstest.plugin.ReceiveSmsPlugin;

public class ReceiveSmsPluginImpl implements ReceiveSmsPlugin {

    /**
     */
```
* <B>getReceivedSms</B>*

```java
@Edr
public org.csapi.schema.parlayx.sms.receive.v2_2.local.GetReceivedSmsResponse
getReceivedSms(org.csapi.schema.parlayx.sms.receive.v2_2.local.GetReceivedSms
parameters)
    throws org.csapi.schema.parlayx.common.v2_1.PolicyException,
    org.csapi.schema.parlayx.common.v2_1.ServiceException{
    throw new RuntimeException("Not yet implemented");
}
```

---

**RequestFactory Skeleton**

The actual class name is <Traffic path 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.

In order to avoid unchecked exception to be thrown, each method should first check if the exception is declared and return, see example below:

```java
for(Class ec : method.getExceptionTypes()) {
    if(e.getClass().isAssignableFrom(ec)) {
        return e;
    }
}
```
If it is not a declared exception, perform a custom conversion to an exception declared for the method. For Parlay X, this is normally ServiceException and PolicyException.

The following files will be generated in the traffic_path 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.

![Class diagram of the generated request factory skeleton code (in blue) and its relations to other classes.](image)

**Prepare and deploy the extension**

Your extension must be injected into the ear files for Network Gatekeeper.

The following files must be injected into the access tier ear file, `wlng_at.ear`:

- `<Traffic Path Identifier>.war` The web archive for the traffic path web service.
- `<Traffic Path Identifier>_callback.jar` The callback EJB archive.

The following file must be injected to the network tier ear file, `wlng_nt.ear`:

- `j_<Traffic Path Identifier>_<protocol>_service.jar` The plug-in archive.

To prepare and deploy the Network Gatekeeper ear files with your extension files:
1. Run the Ant target \texttt{unpackAll} in \texttt{<Project Name>/build.xml}.

   The ear files are fetched from the Network Gatekeeper installation and exploded in the staging directories \texttt{<Project Name>/staging/temp_at} and \texttt{<Project Name>/staging/temp_nt}.

2. If a Web Service has been created, prepare \texttt{wlng_at.ear}.
   
   a. Copy \texttt{<Traffic Path Identifier>.war} and \texttt{<Traffic Path Identifier>_callback.jar} to \texttt{<Project Name>/staging/temp_at}.

   The ant target \texttt{copyATExtension} can be used for this.

   b. Edit \texttt{<Project Name>/staging/temp_at/META-INF/application.xml}, and add the following definitions:

   
   
   \begin{verbatim}
   Listing 4-4  Snippet for application.xml for wlng_at.ear
   
   <module>
     <ejb><Traffic Path Identifier>_callback.jar</ejb>
   </module>
   <module>
     <web>
       <web-uri><Traffic Path Identifier>.war</web-uri>
       <context-root><context></context-root>
     </web>
   </module>

   \end{verbatim}

   \textbf{Note:} The \texttt{context-root} in the snippet above overrides the \texttt{context-root} defined in \texttt{weblogic.xml} in the war file.

3. Prepare \texttt{wlng_nt.ear}:
   
   a. Copy \texttt{j_<Traffic Path Identifier>_<protocol>.jar} to \texttt{<Project Name>/staging/temp_nt}.

   The ant target \texttt{copyNTExtension} can be used for this.
b. Run the ant target updateConfig to update the network tier ear file META-INF/weblogic-extension.xml file with information about the plug-in jar file.

c. Edit <Project Name>/staging/temp_nt/META-INF/application.xml, and add the following definition:

```
Listing 4-5   Snippet for application.xml for wlng_nt.ear

<module>
  <ejb><Traffic Path Identifier>_service.jar</ejb>
</module>
```

d. If the plug-in generates EDRs, edit <Project Name>/staging/temp_nt/APP-INF/classes/edr/edr.xml and add EDR filters.

e. If the plug-in generates alarms, edit <Project Name>/staging/temp_nt/APP-INF/classes/edr/alarms.xml and add alarm filters.

f. If the plug-in generates CDRs, edit <Project Name>/staging/temp_nt/APP-INF/classes/edr/cdr.xml and add CDR filters.

   **Note:** For more information on these filters, see Chapter 8, “Annotations, EDRs, Alarms, and CDRs.”

g. If the plug-in uses the storage service, add definitions for the storage service. Edit <Project Name>/staging/temp_nt/APP-INF/classes/wlng-cachestore-config-extens ons.xml.

4. Run Ant target packAll and deployAll in <Project Name>/build.xml.

The prepared ear files are generated and deployed in Network Gatekeeper. Now you must set up routing to the plug-in in the Plug-in Manager in Network Gatekeeper’s OAM.
CHAPTER 5

The Traffic Path Example

This section describes the traffic path example included in the Extension Toolkit. It serves as an introduction and overview of the example. The build files, code, and descriptor files are installed in the Extension Toolkit directory. In the default installation, this would be <beahome>/wing_et300/tp_example. This chapter covers the following:

- “About the traffic path example” on page 5-2
- “Sequence diagrams” on page 5-4
  - “Web Service interface definition” on page 5-7
  - “Web Services Layer” on page 5-10
  - “Service Capability Layer” on page 5-10
  - “Plug-in Layer” on page 5-11
    - “Plug-in Components” on page 5-11
    - “Network Interface” on page 5-12
    - “Plug-in Manager Registration” on page 5-13
    - “Operation and management” on page 5-13
    - “Storage” on page 5-13
    - “Policy Enforcement” on page 5-16
    - “EDR Generation” on page 5-18
    - “CDR Generation” on page 5-20
The Traffic Path Example

- “Alarm Generation” on page 5-21
- “Logging” on page 5-24
- “Statistics Generation” on page 5-24

Note: For a high-level overview of Traffic Paths in general, see “Introducing Traffic Paths” in the Architectural Overview, a separate document in the Network Gatekeeper set.

About the traffic path example

The Extension Toolkit Traffic Path Example is a fully functional and deployable traffic path. It should be noted however, that deployment is not a supported feature. The purpose of the Example Traffic Path is to be used as a source code reference during the development of a new traffic path. The Traffic Path Example demonstrates what best practices are in design and coding.

Below is a summary of the features provided with the traffic path example:

- Source code for the example traffic path:
  - The Web Service layer, which is responsible for transmitting application-initiated between an application and the Service Capability Layer.
  - The Service Capability layer, which is responsible for transmitting application-initiated and operations between the Web Services layer and the Plug-in layer. This is implemented as an EJB deployed in the Network Tier.
  - The Callback Service Capability layer, which is responsible for transmitting network-initiated operations between the Plug-in layer and the Web Services layer. This is implemented as an EJB in the Access Tier.
  - The Plug-in layer, which is responsible for transmitting application-initiated and network-initiated operations between the Service Capability Layers and the network.

- Build files

  The build file for the Traffic Path is located in the default installation in
  `<beahome>wlng_et300\tp_example\example\example\traffic_path`

  The build file for the plug-in is located in the default installation in
  `wlng_et300\example\tp_example\example\plugins\netex`

- SLA for the traffic path

- Policy Rules for the plug-in: The default Network Gatekeeper policy rules are used.
About the traffic path example

- EDR Listener: The EDR listener is a separate process that receives EDRs. This is a part of the Testing Tools. See Chapter 10, “Testing Your Traffic Path” for more information.

- Web Service Test Client: This is used for testing the application-facing interfaces by enabling the user to run application-initiated traffic. It is generated using the Testing Tools.

- Network Node Test Simulator: This provides a simulated network interface to which the traffic path can direct application-initiated traffic. It also provides functions for sending network-initiated traffic. This simulator is used as part of the Testing Tools.

- Source code for the traffic path:

  The Traffic Path Example interface consists of two Web Services for application-initiated requests, the SendData Web Service and the NotificationManager Web Service. The SendData Web Service is used to send data to the network, and the NotificationManager Web Service is used to start and stop subscriptions for network-triggered event notifications.

  WSDLs for these are located in the default installation at:

  - `<beahome>wlng_et300\tp_example\example\traffic_path\wsdl\service\example_data_send_service.wsdl`
  - `<beahome>wlng_et300\tp_example\example\traffic_path\wsdl\service\example_notification_manager_service.wsdl`

  There is also an interface that defines a Web Service to receive notifications from network-triggered requests, the NotificationListener interface. The Notification Listener interface defines the methods that Network Gatekeeper invokes on a Web Service that is implemented by an application in order to receive network-initiated traffic.

  WSDLs for this is located in the default installation at:

  - `<beahome>wlng_et300\tp_example\example\traffic_path\wsdl\callback`

  The Web Services implementations and the Service Capability EJBs are all generated from the WSDL using the build file located in the default installation at `<beahome>wlng_et300\tp_example\example\traffic_path`

  All plug-in specific build files and code is found under `<beahome>wlng_et300\tp_example\example\plugins\netex`

  The North interfaces of the plug-in are Java representations of the Web Services interfaces.
Sequence diagrams

Below are sequence diagrams describing the interaction between the modules in the example traffic path.

- Application to Network
- Network to Application

Application to Network

In Figure 5-1 an application uses SendData to send data through the traffic path. When the request reaches the plug-in it is translated according to the network protocol and forwarded to the network.

Figure 5-1  SendData

In Figure 5-2, the application sends a request to register for notifications from the network:
In Figure 5-3, the application de-registers for notifications from the network:
Network to Application

In Figure 5-4, the plug-in receives data from the network. It creates a Web Services based method call to notify the Web Service that the application has implemented that data has been received:
**Web Service interface definition**

**Interface: SendData**

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

**Operation: sendData**

Send data to the network.

Input message: sendDataMessage

<table>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>xsd:string</td>
<td>N</td>
<td>The data to be sent to the target device</td>
</tr>
<tr>
<td>address</td>
<td>xsd:anyURI</td>
<td>N</td>
<td>Address of the target device. Example: tel:4154011234</td>
</tr>
</tbody>
</table>

Output message: sendDataResponse
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>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>xsd:string</td>
<td>N</td>
<td>Service unique identifier provided to set up this notification.</td>
</tr>
<tr>
<td>endPoint</td>
<td>xsd:string</td>
<td>N</td>
<td>Endpoint address. Endpoint of the application to receive notifications. Example: <a href="http://www.hostname.com/NotificationService/services/Notification">http://www.hostname.com/NotificationService/services/Notification</a></td>
</tr>
<tr>
<td>address</td>
<td>xsd:anyUR</td>
<td>N</td>
<td>Service activation number. Example: tel:4154567890</td>
</tr>
</tbody>
</table>

Output message: invokeMessageResponse

<table>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Operation: stopEventNotification**

Stop the subscription of event notification from the network.

Input message: stopEventNotificationRequest

<table>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>xsd:string</td>
<td>N</td>
<td>Service unique identifier provided to set up this notification.</td>
</tr>
</tbody>
</table>

Output message: stopEventNotificationResponse

<table>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Interface: NotificationListener**

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

**Operation: notifyDataReception**

Method used for receiving a notification.

Input message: notifyDataReceptionRequest

<table>
<thead>
<tr>
<th>Part name</th>
<th>Part type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>xsd:string</td>
<td>N</td>
<td>Service unique identifier provided to set up this notification.</td>
</tr>
</tbody>
</table>
The Web Services layer code is generated largely based on the WSDLs that define the interfaces. In addition, routing information that is contained in the Web Service request, such as the address to which the data is being sent, must be bundled into an `AddressRequestInfo` object in order for the Plug-in Manager to be able to process it. To do this translation, a traffic path uses a Request Factory. For the Traffic Path Example Web Services there are two factories, `SendDataPluginFactory` and `NotificationManagerPluginFactory`. A Request Factory also contains functionality for converting the exception-types thrown by the plug-in into exception-types that can be thrown by the Web Service. For more on Request Factories, see RequestFactory Skeleton.

The Example Traffic Path’s Request Factories are located (in the default installation) in
<beahome>\wlng_et300\tp_example\example\traffic_path\src\com\acompany\example\plugin\.

### Service Capability Layer

The Service Capability layer is provides the entrance point of the request to the Network Tier. It is responsible for querying the Plug-in Manager for the appropriate plug-in for the request. It also initiates the transaction which brackets the request in the plug-in. The source code for this layer is generated from the application-facing WSDL interface files.
Plug-in Layer

The Plug-in layer is responsible for translating Web Service operations into Network Protocol operations. The Plug-in layer for the Traffic Path Example consists of an example protocol plug-in that translates requests from the example application-facing Web Services interfaces and the Example network interface. Both application-initiated and network triggered requests are supported. A notification handler that matches network-triggered requests with registered notifications is included. Applications register for notifications via the NotificationManager. The plug-in is responsible for registering itself with Network Gatekeeper plug-in manager. The code is also responsible for generating EDRs, CDRs, alarms, and implementing Policy Enforcement Points, although much of the work for this is done using aspects.

The source code for the application-facing (or North) interface of the Example Traffic Path is generated at build time from the application-facing WSDL interface file. The Network Interface for the Traffic Path Example consists of a proprietary interface named Netex.

For the purposes of registering with the Plug-in Manager, plug-ins have a name and a service type. The naming convention is Plugin_<web service interface part>_<_network protocol>

So, for example, in the Traffic Path Example, the plug-in is named Plugin_example_netex. These names are defined in the build properties file.

Plug-in Components

From a traffic point of view, the Plug-in consists of the following main components:

- **SendDataPlugin**: Translates the Java representation of the Web Services SendData call into the Java representation of the Netex protocol
- **InboundAdapter**: Translates the Java representation of the Netex protocol into the sample network interface
- **NotificationManagerPlugin**: Handles setting up a notification, including storing the notification information using the Storage Services
- **OutboundAdapter**: Translates the sample network interface into the Java representation of the Netex protocol
- **NotificationHandler**: Compares information from the Storage Service with data from the network request. If there is a match (that is, if a matching Notification Request has been made), the NotificationHandler plug-in component invokes the method
notifyDataEventReception on the Callback EJB. It uses a generated stub, the Service Callback Client for requests to the Callback EJB.

**Figure 5-5 Plug-in components.**

Java RMI is used between the plug-in and the Service EJB, and the plug-in and the callback EJB. This interface is generated and is the Java representation of the application-facing interface.

**Network Interface**

The network interface is designed only to be a simplified sample protocol especially for the Example plug-in. It does not have a real world counterpart. The name of this network protocol is Netex.

The Network Interface for the Traffic Path Example is defined using EJBs. The interface is synchronous and consists of:

- The inbound interface, used for submitting data to the network node. Application-initiated traffic.
- The outbound interface, used for receiving data from the network node. Network-triggered traffic.
Plug-in Manager Registration

The Example plug-in registers the following plug-in ID in the plug-in manager:

Plugin_example_netex

The following plug-in types are registered:

- com.accompany.example.plugin.SendDataPlugin
- com.accompany.example.plugin.NotificationManagerPlugin

The plug-in also registers its service types with the plug-in manager. The service type is included in the EDR information. The service type name for the Example plug-in is Example.

The plug-in supports the following address schemes: tel

Operation and management

Management methods can be classified into two categories:

- Cluster wide management methods, that are distributed to all the nodes in the cluster.
- Local management methods, that affect only the local node.

The cluster wide management methods are specified using a Configuration MBean and the local management methods are specified using a Runtime MBean.

The Management Object Name for the plug-in is: Plugin_example_netex. This name is displayed in Network Gatekeeper management console when the plug-in is deployed.

The plug-in exposes two management methods to the user, both attributes:

- NetworkNodeUrl: The address of the Network Node. This address is used by the plug-in when submitting data to the Network Node. This attribute is set for the local node only.
- NetworkNodeJNDIName: The JNDI name of the Network Node interface. This name is used by the plug-in when doing a lookup for the Network Node interface. This attribute is distributed to all service instances in the cluster.

Storage

The Storage Service provides extension developers with an efficient way of storing data that needs to be accessed on all cluster nodes. The Example Plug-in uses the Storage Service for storing Notification information. A new Notification information entry is added to the store when the startEventNotification method of the NotificationManagerPlugin plug-in component is invoked. A Notification information entry is removed from the store when the
The Traffic Path Example

stopEventNotification method is called. Notification information is read from the store by
the NotificationHandler plug-in component when the notifyDataReception method is to
be invoked on the Callback EJB.

The Example Plug-in uses a Write Through database store type for storing Notification
information since

- Notification information is only infrequently added.
- Notification information needs to be reliable.
- Notification information is never updated.
- Notification information needs to be persisted.

Because the store is configured to write to a database, a file named
wlng-cachestore-config-extensions.xml must be supplied. In the default installation is
found in
<beahome>\wlng_et300\tp_example\example\plugins\netex\config\cache_store\
It must be added to the wlng_nt.ear file in the APP-INF\classes directory.

The following is the Example wlng-cachestore-config-extensions.xml file. It specifies
information about database table name, column names, names of get and set methods to use to
get and set the data in the column, the name and type of cache to use.

Listing 5-1 Storage service configuration file

```xml
<?xml version="1.0"?>
<!DOCTYPE store-config SYSTEM "wlng-cachestore-config.dtd">
<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)"/>
  </db_table>
</store-config>
```

5-14 Developer's Guide - Extension Toolkit
<methods>
    <get_method name="getCorrelator"/>
    <set_method name="setCorrelator"/>
</methods>
</value_column>

</db_table>

<store cache_name="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>
</query>
The Traffic Path Example

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

</query>

</store-config>

---

Policy Enforcement

Policy is enforced with aspects; therefore no manual changes need to be made to the plug-in source code. Network Gatekeeper default policy rules are applied.

Example SLAs

Below are examples of SLAs. Note that the <scs> element contains the fully-qualified class name of the north interfaces of the plug-in.

Below is an example of a simple application group SLA.

Listing 5-2   Example application group SLA

```xml
<?xml version='1.0' encoding='UTF-8'?>
<Slas applicationGroupID='APPGroup1' xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xsi:noNamespaceSchemaLocation='file:./policy/sla_schema/app_sla_file.xsd'>

  <serviceContract>
    <startDate>2005-01-01</startDate>
    <endDate>2010-01-01</endDate>
    <scs>com.acompany.example.plugin.SendDataPlugin</scs>
  </serviceContract>

</Slas>
```
Below is an example of a simple service provider group SLA.

**Listing 5-3  Example service provider group SLA**

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<Sla serviceProviderGroupID='SPGroup1'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='file:./policy/sla_schema/sp_sla_file.xsd'>

  <serviceContract>
    <startDate>2001-01-01</startDate>
    <endDate>2010-01-01</endDate>
    <scs>com.acompany.example.plugin.SendDataPlugin</scs>
  </serviceContract>

  <serviceContract>
    <startDate>2001-01-01</startDate>
    <endDate>2010-01-01</endDate>
    <scs>com.acompany.example.plugin.SendDataPlugin</scs>
  </serviceContract>

</Sla>
```
The Traffic Path Example

<sabbrev force="true"></sabbrev>com.acompany.example.plugin.NotificationManagerPlugin</sabbrev>

</serviceContract>

</Sla>

EDR Generation

EDRs are generated by the plug-in using aspects. The methods in the plug-in that generate EDRs are annotated with special EDR annotations. A set of mandatory EDR fields must be provided, and additional EDR fields can be supplied.

Table 5-1  Methods that generate EDRs and their additional EDR fields.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Additional EDR Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendData</td>
<td>FIELD_DESTINATION_ADDRESS</td>
</tr>
<tr>
<td>startEventNotification</td>
<td>FIELD_DESTINATION_ADDRESS, FIELD_CORRELATOR</td>
</tr>
<tr>
<td>stopEventNotification</td>
<td>FIELD_DESTINATION_ADDRESS, FIELD_ORIGINATING_ADDRESS</td>
</tr>
<tr>
<td>deliver</td>
<td>FIELD_DESTINATION_ADDRESS</td>
</tr>
</tbody>
</table>

EDR filters for new EDRs must be added to the EDR filter. This is done as a part of the deployment procedure. The filter is located in Network Gatekeeper network tier ear file, wlng_nt.ear, in APP-INF/classes/edr/edr.xml

Listing 5-4  EDR filter for the plug-in

<edr id="11800" description="Example TP plugin OutboundAdapter deliver">
  <filter>
    <method>
      <name>void deliver</name>
    </method>
  </filter>
</edr>
<class>com.acompany.plugin.example.netex.notification.south.adapter.OutboundAdapter</class>

</method>
</filter>
</edr>

<edr id="11810" description="Example TP plugin SendDataPlugin sendData">
<filter>
<method>
  <name>SendDataResponse sendData</name>
</method>
</filter>
</edr>

<class>com.acompany.plugin.example.netex.send_data.north.SendDataPluginImpl</class>

</method>
</filter>
</edr>

<edr id="11820" description="Example TP plugin NotificationManagerPlugin startEventNotification">
<filter>
<method>
  <name>StartEventNotificationResponse startEventNotification</name>
</method>
</filter>
</edr>
The Traffic Path Example

<edr id="11825" description="Example TP plugin NotificationManagerPlugin stopEventNotification">
  <filter>
    <method>
      <name>StopEventNotificationResponse stopEventNotification</name>
    </method>
  </filter>
</edr>

<edr id="11830" description="Example TP plugin PluginSouth submit">
  <filter>
    <method>
      <name>void submit</name>
    </method>
  </filter>
</edr>

CDR Generation

A CDR is an EDR that has been processed through a mapping filter. This filter is located in Network Gatekeeper Network Tier ear file, wlng_nt.ear, in APP-INF\classes\edr\cdr.xml EDRs that are to be mapped to CDRs are defined in the cdr.xml file. The cdr.xml file needs to be updated with EDR-to-CDR mapping that is relevant to the Example Plug-in. This is done as a part of the deployment procedure.

A CDR is generated after the last method of the appropriate request returns.
Listing 5-5  Example of CDR mapping to be added to the cdr.xml file.

```xml
<cdr>
  <filter>
    <method>
      <name>void submit</name>
    </method>
  </filter>
</cdr>
```

**Alarm Generation**

An alarm is an EDR that has been processed through a mapping filter. This filter mapping filter is located in Network Gatekeeper Network Tier ear file, wlng_nt.ear, in APP-INF\classes\edr\alarms.xml.

EDRs that are to be mapped to alarms are defined in the alarms.xml file. The alarms.xml file needs to be updated with EDR-to-alarm mapping that is relevant to the Example Plug-in. This is done as a part of the deployment procedure.

The alarm filter that provides the first match in alarm.xml is used for triggering the alarm.

Listing 5-6  Example of an alarm-group to be added to the alarm.xml to enable alarm generation

```xml
<alarm-group id="299"
    name="example"
    description="ExampleTP alarm">

<alarm id="299000" severity="major"
```
The Traffic Path Example

<filter>
  <exception>
    <class>com.accompany.plugin.example.netex.send_data.north.SendDataPluginImpl</class>
    <method>SendDataResponse sendData</method>
  </exception>
  <filter>
  </filter>
</alarm>

<alarm id="299100" severity="major"
  description="Failed to deliver data">
  <filter>
    <exception>
      <class>com.accompany.plugin.example.netex.notification.south.adapter.OutboundAdapter</class>
      <method>void deliver</method>
    </exception>
  </filter>
</alarm>

<alarm id="299200" severity="major"
  description="Failed to startEventNotification">
  <filter>
  </filter>
</alarm>

<filter>
  <exception>
    <class>com.accompany.plugin.example.netex.notification_manager.north.NotificationManagerPluginImpl</class>
    <method>void notify</method>
  </exception>
  <filter>
  </filter>
</alarm>
<method>StartEventNotificationResponse startEventNotification</method>
</exception>
</filter>
</alarm>

<alarm id="299250" severity="major"
      description="Failed to stopEventNotification">
  <filter>
    <exception>
      <class>com.acompany.plugin.example.netex.notification_manager.north.Notificati
      onManagerPluginImpl</class>
      <method>StopEventNotificationResponse stopEventNotification</method>
    </exception>
  </filter>
</alarm>

<alarm id="299300" severity="major"
      description="Failed to submit in InboundAdapterImpl">
  <filter>
    <exception>
      <class>com.acompany.plugin.example.netex.send_data.south.adapter.InboundAdapte
      rImpl</class>
      <method>void submit</method>
    </exception>
  </filter>
</alarm>
Logging

Logging and tracing in Network Gatekeeper is performed using log4j. Basic tracing of method entry, exit and all exceptions that occur during traffic processing is added at build time by applying Tracing Aspects to the source code of the Traffic Path Example.

Statistics Generation

Statistics are generated using aspects. Because the generation of statistics is the basis for the Network Gatekeeper licensing model, the actual generation of statistics may not be manipulated, although new statistics types may be added. See Extending Statistics for more information.

Classes in the plug-in

All classes in the Example Plug-in belong to the package com.acompany.plugin.example.netex.

context.ContextTranslatorImpl

Implements ContextTranslator.

Helper class to populate a map with name/value pairs to put in the RequestContext.

managed_plugin.ManagedPluginImpl

Extends ManagedPlugin, implements ServiceDeployable.

Contains life-cycle management for the plug-in. Registers the North interfaces with the Plug-in Manager and instantiates helper services.

management.ConfigurationStoreHandler

Helper classes for managing the ConfigurationStore.

management.Management

Manages registration and de-registration of the MBean used for management of the plug-in.

management.MBean

Interface to the MBean used for management of the plug-in.
management.MBeanImpl
Implementation of the MBean interface for the plug-in.

notification.NotificationData
Holds NotificationData used to identify a notification registered by an application.

notification.NotificationHandler
Interface to the Notificationhandler.

notification.NotificationHandlerImpl
Matches a request from the network with a registered notification and calls the Service CallBack EJB with the NotificationData.

notification.south.adapter.OutboundAdapter
Implements PluginSouth. Forwards request from the network to NotificationHandler. Implements necessary method for the Context Aspect to rebuild the RequestContext.

notification.south.adapter.OutboundRegistry
Responsible for registering and accessing OutboundAdapter.

notification_manager.north.NotificationManagerPluginImpl
Implements NotificationManagerPlugin.

send_data.north.SendDataPluginImpl
implements SendDataPlugin
Implements the SendData north interface. It translates calls fro the EJB layer to the network adapter layer.

send_data.south.adapter.InboundAdapter
Definition of the network protocol interface for application-initiated requests.

send_data.south.adapter.InboundAdapterImpl
Implementation of the network protocol interface for application-initiated requests.
The Traffic Path Example

send_data.south.adapter.PluginSouthImpl
The last layer before requests leave the southern boundary of the plug-in and are handed off to the network node. Implements PluginSouth interface so that aspects can weave all methods in order to insert EDRs.

store.StoreHelper
Helper class for storing NotificationData using the Storage Service.

store.FilterImpl
Implements a filter for custom queries to the Storage Service.

org.netex.OutboundProxyBean
Implements the Outbound Proxy as an EJB SessionBean. It is used as a proxy to trigger network initiated events.
CHAPTER 6

Policy

For most installations of WebLogic Network Gatekeeper, the ability rapidly and accurately to evaluate the status of requests in terms of Policy, or rules governing a variety of service characteristics, is one of the most important features that the system offers. If you extend the Network Gatekeeper, particularly if you add a new traffic path, you may also need to make changes in the Policy system to cover new functionality that you have added. This chapter provides a very high level description of the process by which policy requests are processed and though which new rules can be added. It covers:

- **Overview**
- **Policy Request Data**
- **Adding a New Rule**
- **Using RequestContext Parameters Defined in Service Level Agreements**
- **Extending Service Level Agreements**

**Overview**

When an application service request arrives at the Network Plug-in, its parameters are automatically bundled by aspects into a Policy Request object and sent to Policy Decision Points, or PDPs, in the Policy Engine. There they are evaluated according to the rules. The rules in a Policy Decision Point are based both on data derived from some parts of the Application Group and Service Provider Group SLAs and on any other criteria that the operator chooses to employ. The rules themselves are written in the ILOG IRL language.
Figure 6-1 below shows a simplified version of the evaluation flow.

**Figure 6-1  Simplified policy flow**

---

**Policy Request Data**

A PolicyRequest object has a standard form. The values in the object must be mapped to the variables in the Policy Rule that will be used to evaluate them. The Policy Request object can contain subsets of this standard data:
• applicationID: The Application ID of the requesting party
• serviceProviderID: The Service Provider ID of the requesting party
• nodeID: Used internally by Network Gatekeeper - ignore
• serviceName: The name of the software module in which the policy request originates.
  Used in the rules to match to service contracts in the SLAs and to look-up any rules
  specific to the service.
• methodName: The name of the method which the request wishes to have executed. Access
  to this method is what is being evaluated.
• serviceCode: The service code provided by the application, which is written to CDRs for
  tracking purposes
• requesterID: An additional ID that may be provided by the application for tracking
  purposes. (dependent on the northbound interface being used)
All of the above are Strings.
• transactionID: Used internally by Network Gatekeeper - ignore
• noOfActiveSessions: Used internally by Network Gatekeeper - ignore
• timeStamp: The time the request was fed to the Policy Engine.
• reqCounter: The number of target addresses in the request. If only one target address is
  used in the request this value is set to 1. If using multiple target addresses in the request, it
  is the number of target addresses.
All of these are Longs.

In addition to these standard values, Policy Request objects contain all the parameters passed in
from the application in its initial request, as AdditionalParameters, an array of
AdditionalDataValue. An AdditionalDataValue consist of a name-value pair. The
following data types can be defined in an AdditionalDataValue object.

• intValue(int val): Integer values
• longValue(long val): Long values
• stringValue(String val): Strings.
• stringArrayValue(String[] val): Arrays of String values.
• booleanValue(boolean val): Boolean values.
The name of the name-value pair is defined in the `dataName` member variable in the `AdditionalData` object. See Listing 6-1

**Listing 6-1  Defining AdditionalData**

```java
AdditionalData adArray[] = new AdditionalData[1];
AdditionalDataValue targetAddressValue = new AdditionalDataValue();
AdditionalData adTargetAddressString = new AdditionalData();
targetAddressValue.stringValue(address);
adTargetAddressString.dataName = "targetAddress";
adTargetAddressString.dataValue = targetAddressValue;
adArray[0] = adTargetAddressString;
policyRequest.additionalParameters = adArray;
```

If any of the incoming parameters from the application are complex types, the objects are automatically examined and broken down into simple Java types. So, for example, the Parlay X 2.1 complex type `ChargingInformation` can contain a description, which is a string, a currency kind, which is also a string, an amount, which is a decimal number, and a code, which is a string. When the data is sent to the Policy Engine, it is broken down into a string value called `parameters.charging.currency`, another string value called `parameters.charging.code`, and so forth.

**Adding a New Rule**

New rules can be added to the Policy Service. The rule must have a name and a priority.
High priority rules are evaluated before low priority rules. There are a set of pre-defined priority levels, which are mapped to a numerical value:

- **minimum**, where the value is $-1 \times 10^9$
- **low**, where the value is $-1 \times 10^6$
- **high**, where the value is $1 \times 10^6$
- **maximum**, where the value is $1 \times 10^9$

Listing 6-2 shows the basic structure of a rule:

```
Listing 6-2   Skeleton of a rule

rule DenySubscriberNotExists
{
    priority = high;
    when
    {
        // fetch the policy request data and perform evaluations.
    }
    then
    {
        // Take action on
    }
};
```

**Mapping PolicyRequest Data**

In order to perform an evaluation, the data in the `PolicyRequest` object must be fetched by the rule in the Policy Engine and mapped to the equivalent variable name in the rule. The standard types of request data in the Policy Request are associated with variables of the same name in the
rules. Below is an example of a rule assigning the PolicyRequest member variable serviceName to the rule variable sname via the Policy Request object. The rule object pr is assigned to the PolicyRequest object.

**Listing 6-3   Policy Request data is fetched**

```
?pr: event PolicyRequest(?sname: serviceName);
```

If the Policy Engine has evaluated the request and made the decision to deny it, the Policy Engine’s representation of the PolicyRequest object (pr) must be retracted. Retracting the PolicyRequest object aborts further rule enforcement.

**Listing 6-4   Retract a request**

```
retract (?pr);
```

If the Policy Engine has evaluated the request and made the decision to allow it, the Policy Engine’s representation of the request (pr) must still be retracted, but in the last rule of the execution flow. For example, this could be achieved by adding a general finalizing allow rule that retracts the request. This rule should have priority minimum.

**Listing 6-5   General finalizing allow rule that retracts a request**

```
rule AllowServiceRequest
{
    priority = minimum;
    when
    {
        ?pr: event PolicyRequest();
    }
```


Data that is defined as AdditionalValues must fetched as shown in Listing 6-6. The Additional Value named targetAddress is stored in the variable addDataValue. The PolicyRequest object is pr.

**Listing 6-6  Fetching AdditionalValue data**

```java
bind ?addDataValue = ?pr.getAdditionalDataStringValue("targetAddress");
```

The particular signature of the fetching method depends on the type of data:

- `getAdditionalDataIntValue(...)`, for int values
- `getAdditionalDataLongValue(...)`, for long value.
- `getAdditionalDataStringValue(...)`, for String values
- `getAdditionalDataStringArrayValue(...)`, for arrays of String values
- `getAdditionalDataBooleanValue(...)`, for boolean values
- `getAdditionalDataShortValue(...)`, for short values
- `getAdditionalDataCharValue(...)`, for char values
- `getAdditionalDataFloatValue(...)`, for float values
Policy

- `getAdditionalDataDoubleValue(...)` for double values
- `getAdditionalDataIntArrayValue(...)` for arrays of int values.

If the data type is unknown, it can be determined by invoking the discriminator method on the `AdditionalDataValue` object.

**Listing 6-7  Determine the type of an AdditionalDataValue**

```java
bind ?type = ?pr.getAdditionalData.dataValue.discriminator().value();
```

Where `type` is one of the following:
- `AdditionalDataType._P_ADDITIONAL_INT`
- `AdditionalDataType._P_ADDITIONAL_LONG`
- `AdditionalDataType._P_ADDITIONAL_STRING`
- `AdditionalDataType._P_ADDITIONAL_STRING_ARRAY`
- `AdditionalDataType._P_ADDITIONAL_BOOLEAN`
- `AdditionalDataType._P_ADDITIONAL_SHORT`
- `AdditionalDataType._P_ADDITIONAL_CHAR`
- `AdditionalDataType._P_ADDITIONAL_FLOAT`
- `AdditionalDataType._P_ADDITIONAL_DOUBLE`
- `AdditionalDataType._P_ADDITIONAL_INT_ARRAY`

Creating a New Rule File by Extending an Existing File: an Example

The following shows an example of extending an existing rule file:

- List the Current Services’ Rule Files
- Select the Service Whose Rule File You Wish to Extend
- Add a New Extended Rule
List the Current Services' Rule Files

In the Network Gatekeeper Administration Console extension, select the PolicyMBean managed object and the listApplicationRuleFiles operation:

Figure 6-2  listApplicationRuleFiles

Select the Service Whose Rule File You Wish to Extend

Using the same managed object, select the viewApplicationRuleFile. Enter the name of the service whose rule file you wish to change. Here the ESPA_message_sender rule file is selected:
This produces the following rule listing:

Listing 6-8 The current ESPA_message_sender rule file

```java
//========================================================== -*- Java -*-
// Application rules
//==========================================================

import com.incomit.policy.*;
import com.incomit.policy.sla.*;
import java.util.Vector;
```
import java.net.*;

//General finalizing allow rule that will retract the request

rule AllowServiceRequest {
    priority = minimum;
    when {
        ?ci: ContextInformation(?pr: policyRequest);
    } then {
        retract (?ci);
        ?pr.allow();
    }
};

//Check method parameter values functions

function String verifyParameterValue(Vector ?allowedVals, String ?value, boolean ?allowValue) {
    bind ?dataValue = null;
    if(?value != null &&
        (!?allowedVals.contains(?value) && !?allowValue ||
          !?allowedVals.contains(?value) && ?allowValue)) {
        ?dataValue = ?value;
    }
    return ?dataValue;
}
    bind ?denyVal = null;
    bind ?addData = ?pr.getAdditionalData(?pn);
    if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_INT) {
        ?denyVal = verifyParameterValue(?pv, Integer.toString(?pr.getAdditionalDataIntValue(?pn)), ?av);
    } else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_LONG) {
        ?denyVal = verifyParameterValue(?pv, Long.toString(?pr.getAdditionalDataLongValue(?pn)), ?av);
    } else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_STRING) {
        ?denyVal = verifyParameterValue(?pv, ?pr.getAdditionalDataStringValue(?pn), ?av);
    } else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_STRING_ARRAY) {
        bind ?strArr = ?pr.getAdditionalDataStringArrayValue(?pn);
        bind ?i = 0;
        for(?i = 0; ?i < ?strArr.length; ?i++) {
            ?denyVal = verifyParameterValue(?pv, ?strArr[?i], ?av);
            if(?denyVal != null) {
                break;
            }
        }
    } else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_BOOLEAN) {
        ?denyVal = verifyParameterValue(?pv, Boolean.toString(?pr.getAdditionalDataBooleanValue(?pn)), ?av);
    }
Adding a New Rule

```java
} else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_SHORT) {
    ?denyVal = verifyParameterValue(?pv,
        Short.toString(?pr.getAdditionalDataShortValue(?pn)), ?av);
} else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_CHAR) {
    ?denyVal = verifyParameterValue(?pv,
        Character.toString(?pr.getAdditionalDataCharValue(?pn)), ?av);
} else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_FLOAT) {
    ?denyVal = verifyParameterValue(?pv,
        Float.toString(?pr.getAdditionalDataFloatValue(?pn)), ?av);
} else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_DOUBLE) {
    ?denyVal = verifyParameterValue(?pv,
        Double.toString(?pr.getAdditionalDataDoubleValue(?pn)), ?av);
} else if(?addData.dataValue.discriminator() == AdditionalDataType.P_ADDITIONAL_INT_ARRAY) {
    bind ?intArr = ?pr.getAdditionalDataIntArrayValue(?pn);
    bind ?i = 0;
    for(?i = 0; ?i < ?intArr.length; ?i++) {
        ?denyVal = verifyParameterValue(?pv, Integer.toString(?intArr[?i]), ?av);
        if(?denyVal != null) {
            break;
        }
    }
} else {
    //Should never happen, allow
}
```
return ?denyVal;
}

//------------------------------------------------------------------------
//Check method parameter values rule
rule CheckMethodParameterValues {
  priority = high;
  when {
    ?ci: ContextInformation(?contract: contract;
        ?contract != null;
        ((Contract)contract).params isknown;
        ?pr: policyRequest;
        ?sn: ?pr.serviceName;
        ?mn: ?pr.methodName);

    ?mp: MethodParameters(methodName equals ?mn;
        ?pn: parameterName;
        ?pv: parameterValues;
        ?av: acceptValues) in
        ((Contract)?contract).params.methodParametersList;

    } then {
      if(?pr.additionalDataExists(?pn)) {
        bind ?denyVal = verifyParameterValues(?pr, ?pn, ?pv, ?av);
        if(?denyVal != null) {
          retract (?ci);
          ?pr.deny("Value " + ?denyVal + " for parameter " + ?pn + " was denied by SLA!");
        }
      }
    }
Adding a New Rule

```java
AlarmUtilities.getInstance().generateAlarm("major", "Value for parameter " + ?pn + " was denied by SLA!");
}
}
}

//------------------------------------------------------------------------
//Deny the service request if no valid service contract is found.
//
rule DenyNoServiceContract {
    priority = high + 100;
    when {
        ?pr: PolicyRequest();
        not ServiceContract(scs equals ?pr.serviceName);
    } then {
        retract (?pr);
        ?pr.denyExt("No service contract found!", 2);
        AlarmUtilities.getInstance().generateAlarm("major", "Denied! No service contract for service: " + ?pr.serviceName + " found for application ID: " + ?pr.applicationID + " in SLA");
    }
};

//------------------------------------------------------------------------
//Deny the service request if the service contract is out of date.
//
rule DenyServiceContractOutOfValidationDate {
```
priority = high + 100;
when {
    ?pr: PolicyRequest(?sname: serviceName);
    ?sdh: PolicySlaDurationHelper();
    ?sc: ServiceContract(scs equals ?sname;
        ?sdate: startDate;
        ?edate: endDate);

evaluate (!?sdh.checkSlaDurationDate(?sdate,?edate));
} then {
    retract (?pr);
    ?pr.denyExt("ServiceContract out of date!", 4);
    AlarmUtilities.getInstance().generateAlarm("major", "Denied! The Servicecontract for " + ?pr.serviceName + " is out of date for application ID: " + ?pr.applicationID);
}
}

//Create the ContextInformation in working memory.
//The new object will contain the valid 'contract' element and PolicyRequest object.
//We are retracting the PolicyRequest event object to avoid more rules been triggered.
rule SaveValidOverrideDateWeekTime {
    priority = high + 51;
    when {
        ?pr: PolicyRequest();

Adding a New Rule

```prolog
?-sdh: PolicySlaDurationHelper();
?-or: com.incomit.policy.sla.Override();
   evaluate (?sdh.checkOverrideDurationDateWeekTime(?or.startDate,
        ?or.endDate, ?or.startTime, ?or.endTime, ?or.startDow, ?or.endDow));
) then {
   insert ContextInformation(?or.contract, ?pr);
   retract ?pr;
}
};

//Default rule to create ContextInformation
rule SaveValidServiceContract {
   priority = high + 50;
   when {
      ?pr: event PolicyRequest(?sname: serviceName);
      ?sdh: PolicySlaDurationHelper();
      ?sc: ServiceContract(scs equals ?sname);
   } then {
      insert ContextInformation(?sc.contract, ?pr);
      retract ?pr;
   }
};

//General method for checking method access.
//All requests to a blacklisted method will be denied.
rule DenySCSMетодNotAccessible {
   priority = high;
```
when {
    ?ci: ContextInformation(contract != null;
        ((Contract)contract).methodAccess is known;
        ?pr: policyRequest;
        ?mname: ?pr.methodName);

    ?bm: BlacklistedMethod(?methodName: methodName;
        ?methodName equals ?mname) in
}

    then {
        retract (?ci);
        ?pr.denyExt("Service method not allowed to be accessed!", 6);

        AlarmUtilities.getInstance().generateAlarm("major", "Denied! The method" + ?pr.methodName + " is not accessible for application ID: " + ?pr.applicationID);
    }
}

//START ESPA_message_sender specific rules //
//Check for content types that may not be allowed.
//Request
rule DenyContentTypeNotAllowed {
    priority = high;

    when {
        ?ci: ContextInformation(?pr: policyRequest;
Adding a New Rule

?serviceName: ?pr.serviceName;
?methodName: ?pr.methodName;
?methodName.equals("sendMessageReq");

?contentTypes: ?pr.getAdditionalDataStringArrayValue("Content-type");
?sc: ServiceContract(?scs: scs;
?scs equals ?serviceName;
contract isknown;
contract.allowedContentTypes isknown;
?allowedContentTypes: contract.allowedContentTypes);

evaluate
(!StringMatcherUtility.getInstance().areAllAllowed(?contentTypes,?allowedContentTypes));

) then {

retract (?ci);
?pr.denyExt("Non allowed content type.", 901);
AlarmUtilities.getInstance().generateAlarm("minor", "Denied content from application " + ?pr.applicationID + ".");
}

// Check maximum message length.
// Request
rule DenyMessageSizeNotAllowed {


Policy

priority = high;
when {
    ?ci: ContextInformation(?pr: policyRequest;
        ?serviceName: ?pr.serviceName;
        ?methodName: ?pr.methodName;
        ?methodName.equals("sendMessageReq");
        ?messageSize: ?pr.getAdditionalDataIntValue("Content-length");
    )
    ?sc: ServiceContract(?scs: scs;
        ?scs equals ?serviceName;
        contract isknown;
        contract.maxMessageSize isknown;
        ?maxMessageSize: contract.maxMessageSize);

evaluate (?messageSize > ?maxMessageSize);
}
}

retract (?ci);
?pr.denyExt("Message exceeds size limit", 903);
AlarmUtilities.getInstance().generateAlarm("minor", "Denied too large message from application " + ?pr.applicationID + ".");
}
};

//----------------------------------------------------------------------//
// END ESPA_message_sender specific rules   //
//----------------------------------------------------------------------//
Add a New Extended Rule

This example rule is trivial, and simply for demonstration purposes. Create a new file named “ESPA_message_sender_ext.ilr” by appending the following to the contents of the file in:Listing 6-8:

Listing 6-9 The new rule

```java
//------------------------------------------------------------------------
//Rule created for example purposes.
//
rule ExampleRule {
    priority = high;
    when {

        ?ci: ContextInformation(?pr: policyRequest;
            ?pr.applicationID.equals("SP1APP1"));
    } then {
        System.out.println("The Application ID is SP1APP1");
    }
};
```

Load the New Rule File

Using the Network Gatekeeper Management Console extension, load the new rule file:
Figure 6-4  Load new rule file

For the rule to be available, you must now restart Network Gatekeeper. To test the new rule extension, run traffic through the WAP Push traffic path from an application whose application ID is `SP1APP1`. You should see “The Application ID is SP1APP1” echoed to STDOUT.

Using RequestContext Parameters Defined in Service Level Agreements

It is possible to use generic data specified in service provider and application-level SLAs in a plug-in. This is useful when the a plug-in shall take a certain action or use a certain behavior based on which service provider or application the request originates from. For example, this can be used for information about parameters that corresponds to a certain group of applications. For instance a certain group might get the priority on their SMS set to LOW because they pay less. The priority might be a parameter that is sent down to the network which handles this.
In an SLA, a `<contextAttribute>` is defined as a name/value pair, where the name is defined in the tag `<attributeName>` and the value is specified in `<attributeValue>`.

A plug-in can retrieve the value specified in `<attributeValue>` using the name specified in `<attributeName>`. The value is retrieved using the RequestContext for the request:

```java
String attributeValue = (String) RequestContextManager.getCurrent().get("<attributeName>");
```

For example, the value associated with the contextAttribute with the attributeName `com.bea.wlcp.wlng.plugin.sms.testName1` is retrieved using:

```java
String value1 = (String) RequestContextManager.getCurrent().get("com.bea.wlcp.wlng.plugin.sms.testName1");
```

Extending Service Level Agreements

Service Level Agreements (SLAs) are XML files that contain some of the data which is enforced by the Policy Engine using the rules. To understand more about the types of SLAs, see “Managing Application Service Providers” in the Architectural Overview. Rules based on Service Level Agreements are created and loaded into the Policy Engine for these levels:

- Service Provider Group
- Application Group

There are separate Policy rules that enforce these SLAs, one on the service provider level and one on application level. To extend Service Level Agreements, the following steps must be taken:

1. “Update the SLA Schema” on page 6-23
2. “Load the new SLA schema into the Policy Service” on page 6-24
3. “Update and reload the rule files” on page 6-24
4. “Update the appropriate SLAs” on page 6-25
5. “Load the updated SLAs” on page 6-25

Update the SLA Schema

The SLA schema files are located in the domain\policy directory. In the default installation this would be `<beahome>\user_projects\domains\wlng-domain\policy\`
There are two pertinent schema files, based on the SLA type:

- `app_sla_file.xsd`: Defines the SLA schema for the application level SLAs.
- `sp_sla_file.xsd`: Defines the SLA schema for the service provider level SLAs.

Update the appropriate schema file by adding appropriate elements. For example, if the service provider SLA schema needs a new element to define a String value listed in the service provider SLA as `<additionalData>mydata</additionalData>`, update the `serviceContract` in the schema file with the following element:

```xml
<xs:element name="additionalData" minOccurs="0" maxOccurs="1" type="xs:int"/>
```

### Load the new SLA schema into the Policy Service

Load the updated SLA schema into the Policy Service using the Network Gatekeeper Console extension. The following methods in the Policy Service are used:

- `reloadApplicationXmlDriver`: Reloads application level SLA schemas.
- `reloadServiceProviderXmlDriver`: Reloads service provider level SLA schemas.

### Update and reload the rule files

In order to enforce the new data in the SLA, the rule files must also be updated. The data in the SLA is fetched from an object model the Policy Engine creates from the data defined in the SLAs. The data in an SLA is fetched from the Policy Rules by name. For example, if a tag in the SLA is `<additionalData>`, the data is fetched using the same name, as described below.

The rule gets the parameter `aParam` from the Policy Request object, and puts it in the local variable `pr`. The parameter `aParam` is compared with the data fetched from the SLA. The request is denied if the parameter given in the SLA is larger than the parameter provided in the Policy Request.

#### Listing 6-10  Get SLA data and compare with a parameter in a Policy Request

```java
rule denyAParamValueNotAllowed
{
    priority = high;
    when
```
Extending Service Level Agreements

{  
  ?pr: event PolicyRequest(?serviceName: serviceName;  
    ?aParam: aParam);  
  ?sc: ServiceContract(?scs: scs;  
    ?scs equals ?serviceName;  
    ?aParam > additionalData);  
}  

then  
{  
  retract (?pr);  
  ?pr.deny("The parameter is not allowed!");  
}

The updated rules must also be loaded into the Policy Engine: this is performed with the Network Gatekeeper Console extension using the following methods in the Policy Service:

**loadApplicationRules**: Reloads application level rules.

**loadServiceProviderRules** Reloads service provider level rules.

**loadNodeRules**: Reloads service provider traffic and total traffic rules.

**Update the appropriate SLAs**

If the schema is updated, the SLAs themselves must also be updated to contain any new elements and data. For example:

```xml
<additionalData>mydata</additionalData>
```

**Load the updated SLAs**

The SLAs holding the new parameters must be loaded into the Policy Engine. See “Enabling Service Providers and Applications” in the for information on loading a updated SLAs.
Policy
Core and Core Utilities

This chapter provides a high-level description of Network Gatekeeper Core and Core utilities. It also provides an overview of other parts of the API available for the use of extension developers:

- “Core and Core Utilities” on page 7-2
  - “Core” on page 7-2
  - “Core utilities” on page 7-3
- “Plug-in” on page 7-7
- “Management” on page 7-13
- “EDR” on page 7-13
- “Service Correlation” on page 7-13
- “Parameter Tunneling” on page 7-16
- “Storage Services” on page 7-16
  - “ConfigurationStore” on page 7-16
  - “StorageService” on page 7-20
- “Using shared libraries” on page 7-30

Note: The Javadocs for the Core API is available in the `<bea_home>/<et_wlng_home>/doc/javadoc` directory of the extension toolkit installation and on the Network Gatekeeper site at edocs.bea.com.
Core and Core Utilities

The Network Gatekeeper Core and Core Utilities provide the basic infrastructure by which an extension traffic path and the core functionality of Network Gatekeeper can communicate.

Core

In order for your service (generally the plug-in part of your traffic path) to interact with Network Gatekeeper it must be deployable in the context of Network Gatekeeper. Once it is deployable, it can have access to certain utility functions.

Interface: ServiceDeployable

Any service that is to be deployed in Network Gatekeeper must implement ServiceDeployable.

Note: Unless you are adapting a 2.2 style traffic path, the only place you need to implement this interface is in the skeleton AbstractManagedPlugin class generated in your plug-in by the Eclipse plugin.

This interface allows Network Gatekeeper Core to notify the service of its deployment status, or state. Network Gatekeeper Core uses the object implementing this interface to call the following methods on your service:

- `started()` - Called when the module has been started
- `activated()` - Called when the module has been activated (is in normal running state)
- `deactivated()` - Called when the module is to be deactivated.
- `stopped()` - Called when a module is to be stopped.
- `setServiceContext(…)` - Called to set the Service Context. An object representing the Service Context is passed in. This ServiceContext object is used to access the services made available by all other Core modules.

Interface: ServiceDeployableExt

If your service supports the suspend/resume states, you must also implement the ServiceDeployableExt - the Service Deployable Extension - interface.

The interface extends ServiceDeployable by adding following operations:
Core and Core Utilities

- `getNumberOfActiveSessions()` - Returns the number of active sessions the module holds.
- `resume()` - Causes the module to change its suspended state back to active state
- `suspend()` - Suspends the module

**Interface: ServiceContext and SLEEContext**

The `ServiceContext` object represents the Network Gatekeeper Core context for a module; that is, it provides an initial object for retrieving a number of Core and Core utility services. One of these `ServiceContext` objects is automatically set in any deployable module at startup. You also use the Service Context object to access a `SLEEContext` object by using its `getSLEEContext()` method. Your module can use `SLEEContext` to access a number of SLEE and other utility services.

Your service can use these objects to access the following SLEE and other utility services:

- The Configuration Store, which is used to persist read-mostly data, such as configuration information.
- The ID Manager, which is used to retrieve unique IDs.
- The Time Manager, which is used for accessing system time and setting up timers.

It also provides the following methods:

- `getSLEEEid()` - Returns the ID of the Network Gatekeeper server the module is executing in.
- `getSLEEName()` - Returns the name of the Network Gatekeeper server the module is executing in.
- `listSLEENames()` - Returns a list of Network Gatekeeper Servers in the cluster
- `getORB()` - Gets the ORB in Network Gatekeeper.

For details see Javadoc for Package `com.incomit.slee`: Interface SLEEContex

**Core utilities**

In addition to the general classes and methods that the SLEE Context and the Service Context make available, there are also some more specific utilities that they offer:
Class: InstanceFactory

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

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

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

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

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

2. If a mapping is provided and the target class has a public constructor or static singleton method, instantiate it.

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

Listing 7-1  Example instancemap file

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

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

Class: ClusterHelper

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

Helper class for getting the JNDI Context for the network and access tier.
For details see Javadoc for Package com.bea.wlcp.wlng.util.cluster Interface ClusterHelper.

**Service: Event Channel Service**

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

**Interface: SLEEEventChannel**

Use this interface to broadcast events to other instances of Network Gatekeeper, and to register listeners for events originating in them. It is used, for example, in propagating changes of cached data.

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

The following methods are available:

- `deactivateAllListeners()` - Deactivates all registered listeners.
- `publishEvent` - Publishes an event to all registered listeners.
- `publishEventToOneNode` - Publish an event to one Network Gatekeeper instance.
- `registerEventListener` - Register an EventListener.
- `unregisterEventListener` - Unregister an EventListener.

For details see Javadoc for Package com.incomit.slee.event_channel: Interface SLEEEventChannel.

**Interface: EventChannelListener**

This interface is used to receive events published using EventChannel.

The following method is available:

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

For details see Javadoc for Package com.incomit.slee.event_channel: Interface SLEEEventChannelListener.
Service: Time Service

The time service provides a mechanism for retrieving system time and setting up various timers. Periodic timers can generate certain events at regular intervals. For example, an application can read the contents of a database table containing subscription numbers, and use the results to submit a group SMS.

You should use a new object reference for each timer to be used. If the same reference is used and you want to cancel a timer, any of the timers using the same object reference could be cancelled.

Interface: SLEETimeManager

SLEETimeManager is used for managing system time and scheduling timers. The following methods are available:

- `cancelTimer(Object reference)` - Cancel a timer currently in the queue.
- `getSleeProcessUpTime()` - Get the time since last startup of the process (uptime).
- `getSleeUpTime()` - Get the time since last start (uptime).
- `getTime()` - Get the current system time.
- `scheduleTimer(boolean periodic, long time, SLEETimerListener listener, Object reference)` - Schedule a timer to trigger once or periodically.

Interface: SLEETimerListener

Use this interface to receive notifications from timers scheduled using SLEETimer. There is one method available:

- `processTimer(Object reference)` - Informs the timed object that its timer has expired.

It is possible to:

- Start and schedule a periodic timer
- Remove a timer
- Process timer events

For details see Javadoc for Package `com.incomit.slee.time`: Interface SLEETimeManager.
Other APIs

In addition to the Core and Core Utilities, Network Gatekeeper provides other APIs for the use of extension developers.

Plug-in

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

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

Interface: Plugin


Interface: PluginNorth

This interface must be implemented by your plug-in. It defines the entry-point for application-initiated requests and is one of the borders at which aspects are woven.

Interface: PluginNorthCallback

This interface must be implemented by any plug-in that expects network-originated traffic, either actual traffic or notifications. Defines the limit between the plug-in and the Access Tier in the case of network-initiated requests, for example an asynchronous call-back. It manages transporting asynchronous responses from a plug-in to an application.

Interface: PluginSouth

This interface must be implemented by your plug-in. Defines the south border of a plug-in, that is, the network-facing border. It contains methods used to rebuild the RequestContext for network-initiated requests, using information from the ContextMapperInfo object.

Class: ServiceType

This is an abstract utility class that each plug-in must implement. An object of this type is passed to the Plug-in Manager when the plug-in registers itself, so that the Plug-in Manager can query for service type.
Aspects takes care of making this service type available in the request thread of each plug-in. The service type is used by various services, including the EdrService.

Table 7-1 Existing ServiceTypes

<table>
<thead>
<tr>
<th>ServiceType</th>
<th>Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessServiceType</td>
<td>Access</td>
</tr>
<tr>
<td>ThirdPartyCallServiceType</td>
<td>Third-party Call</td>
</tr>
<tr>
<td>CallNotificationServiceType</td>
<td>Call Notification</td>
</tr>
<tr>
<td>SmsServiceType</td>
<td>Sms</td>
</tr>
<tr>
<td>MultimediaMessagingServiceType</td>
<td>Mms</td>
</tr>
<tr>
<td>PaymentMessagingServiceType</td>
<td>Payment</td>
</tr>
<tr>
<td>TerminalStatusServiceType</td>
<td>Terminal Status</td>
</tr>
<tr>
<td>TerminalLocationServiceType</td>
<td>Terminal Location</td>
</tr>
<tr>
<td>CallHandlingServiceType</td>
<td>Call Handling</td>
</tr>
<tr>
<td>AudioCallServiceType</td>
<td>Audio Call</td>
</tr>
<tr>
<td>CallHandlingServiceType</td>
<td>Call handling</td>
</tr>
<tr>
<td>PresenceServiceType</td>
<td>Presence</td>
</tr>
</tbody>
</table>

**Interface: ContextMapperInfo**

This interface defines a ContextMapperInfo object. When network-initiated traffic enters the plug-in from the network-facing (south) side, aspects take any annotated arguments from the network call that will be needed by the plug-in for correlation purposes and places them in this very short-lived object. Arguments are stored by key, defined when the annotation is set, that make it possible to retrieve a particular value. So if an argument is annotated with @MapperInfo(C), its value can be retrieved using the key “C”. Methods in the plug-in that need to retrieve these arguments in order to perform a mapping (for example, associating a notification with the session ID of the request that established it) can use this object. The PluginSouth interface includes one such method, resolveAppInstanceGroupId.
**Interface: RequestContext**

Defines a `RequestContext` object. A `RequestContext` object is available in all new traffic paths for both application-initiated and network-initiated requests. It contains contextual information about the request, including the service provider account ID, application account ID, and application instance group ID (login username) of the application that initiated either the request or the notification, as well as the session ID.

**Class: ManagedPlugin**

Allows the plug-in to register itself in the plug-in manager. See **Class: AbstractManagedPlugin**.

**Class: AbstractManagedPlugin**

Extends `ManagedPlugin`, implements `ServiceDeployable`. The Eclipse plug-in generates a skeleton of this class, which each plug-in must implement. It makes the plug-in deployable as a service in Network Gatekeeper, and assists in registering the plug-in with the Manager. See the `com.bea.wlcp.wlng.api.plugin.common` package JavaDoc for details.

**Class: RequestFactory**

The Request Factory is used to perform application-initiated request processing both before and after a request is processed in the plug-in. Each traffic path must have one implementation of the Request Factory per each application-facing interface, named according to the pattern: `<myinterfacename>.PluginFactory`. A skeleton for the factory is generated by the Eclipse plug-in.

The Request Factory has two main functions. It:

- Packages routing information contained in the request into a `RequestInfo` object that the Plug-in Manager uses to select an appropriate plug-in to process the request. See below for more information on `RequestInfo` objects.

**Note:** In order to support sendlists which target multiple plug-ins, the Request Factory implementation must support three methods that are not required for non-sendlist based plug-ins:

- `createRequestInfos`: allows the creation of multiple `RequestInfo` objects. Each instance of a `RequestInfo` object is matched to a plugin. For example if an SMS message request is sent to 3 addresses, the factory should create an array of 3 `AddressRequestInfo` objects.
Core and Core Utilities

- **createPartialRequest**: splits a request into multiple requests sent to different plug-ins
- **mergeResults**: merges the results reported back by multiple plug-ins into a single result.

For more information, see the Request Factory JavaDoc

**Note:** Plug-ins are invoked in sequence and if one of them fails the whole request is considered a failure. In this case, an exception is thrown and the transaction is rolled back.

- Translates any exceptions thrown in the plug-in (or the underlying network) into a form that can be sent back to the application.

Here is an overview of the traffic flow for an application initiated request:

1. Application request enters the Network Tier at the Service Capability SLSB.
2. The SC SLSB finds the appropriate Request Factory for processing the request by using a naming convention based on the Java class implementation. If the request is intended for a plug-in interface named `com.bea.wlcp.wlng.plugin.MyInterface`, the Request Factory must be implemented in a class named `com.bea.wlcp.wlng.plugin.MyInterfacePluginFactory`. If there is no factory implementation for a specific interface there is a fall-back to a default factory.
   **Note:** If the default factory is used, routing based on parameters like destination address will not be supported and exception conversion will not work.
3. The `validateRequest` method of the factory can be used for parameter validation.
4. The SLSB calls the `createRequestInfo` method of the factory. It is now up to the factory to create and return a `RequestInfo` object based on the parameters in the request.
5. The SLSB asks the Plug-in Manager to find a plug-in that matches the request.
6. The request is sent to the provided plug-in.
7. If the result is processed correctly, the result is returned to the application. If an exception occurs, the exception is filtered using the `convertEx` of the factory. This allows the factory to make sure that the exception can be received by the application. For example, if there is no available plug-in a `NoAvailablePluginException` is thrown. The factory can substitute a `MyServiceException` and add the appropriate details.

The plug-in manager will find a plug-in to use by filtering the list of all available plug-ins using a number of PluginFilters. Each filter may remove plug-ins from the list or rearrange the order of...
the list. Once all filters have been applied, the first one in the list executes the request. These filters are applied (in this order):

1. ActivePluginFilter: Removes plug-ins that are not active.
2. AddressPluginFilter: Removes plug-ins that do not have a matching route.
3. CustomPluginFilter: Removes plug-ins where the customMatchCriteria returns false.
4. Overridable filter: This is a filter loaded by the InstanceFactory for type com.bea.wlcp.wlng.plugin.filter.PluginFilter. By default, this is the RoundRobinPluginFilter which will rearrange the list so that the plug-in is selected using round-robin. This filter could be customized with any kind of behavior, by overriding an instancemap, see below.
5. PolicyPluginFilter: The filter pick the first plug-in from the list that is within the node SLA limits. This filter also enforces SLA throttling.

Listing 7-2 Interface for the PluginFilter.

```
public interface PluginFilter {

    /**
     * @param type Plug-in type
     * @param list List of all plug-ins that are currently available.
     * @param requestInfo The request information
     * @return A modified list where plug-ins that should not be used has been removed.
     * @throws PluginException
     */
    public List<PluginHolder> filter(Class type, List<PluginHolder> list,
                                        RequestInfo requestInfo)
                throws PluginException;
}
```
Listing 7-3  Example implementation that picks a plug-in at random.

```java
public class ExamplePluginFilter implements PluginFilter {

    @Override
    public List<PluginHolder> filter(Class type, List<PluginHolder> list,
                                      RequestInfo requestInfo) {

        if (list.size() > 0) {
            int random = getRandomInt();
            int index = random % list.size();
            PluginHolder ph = list.remove(index);
            list.add(0, ph);
        }
        return list;
    }
}
```

The instancemap in _g_slee_resource.jar_ in _wlng_nt.ear_ must be updated to map to the filter implementation.

**Class: RequestInfo**

The object created by the RequestFactory to hold information from the application-initiated request. There are four sub-classes of RequestInfo that can be used depending on the request:

- AddressRequestInfo, if the request contains an address.
- CorrelatorRequestInfo, if the request contains a correlator.
- RegistrationIdentifierRequestInfo, if the request contains a registration identifier.
- RequestIdentifierRequestInfo, if the request contains a request identifier.
Management
Base classes and annotations for giving the Network Gatekeeper Management Console or other JMX tools management access to extension traffic paths. See Chapter 9, “Making Traffic Paths Manageable” for more information. Also see the JavaDoc for the packages: com.bea.wlcp.wlng.api.management.*

EDR
See Chapter 8, “Annotations, EDRs, Alarms, and CDRs.”. Also see the JavaDoc for the packages com.bea.wlcp.wlng.api.edr.*

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

- The Service Correlation ID is optional.
- The Service Correlation ID is captured in the CDRs and EDRs generated from WLNG.
- The Service Correlation ID is propagated as a String.
- The Service Correlation ID is propagated to/from the application in the SOAP header.
- The Service Correlation ID functionality is available only in 3.0 style traffic paths.

The SCID itself is provided either by the application or by an external mechanism that the traffic path implementor must provide (see Interface: ExternalInvocation). Network Gatekeeper does not check whether or not it is unique. The SCID is stored in WLS Work Context, so that it can be accessed by both the Access Tier and the Network Tier. The Service Correlation class registers itself as a RequestContextListener. When application-initiated request traffic enters the plug-in, the Service Correlation service takes the SCID from the Work Context and places it in the RequestContext object, where it will be available to the EDR service. When network-initiated request traffic is leaving the plug-in, the Service Correlation service takes the
SCID from the RequestContext object and places it in the Work Context, where it can be retrieved by the SOAP Handler and passed along to the application.

Interface: ExternalInvocation

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

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

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

Class: ExternalInvokerFactory

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

```
com.bea.wlcp.wlng.api.ExternalInvocation=myPackageStructure.myImplClass
```

where myImplClass is the ExternalInvocation implementation. The instancemap should be stored in the WLNG_NT.ear under APP-INF\classes.

Class: ServiceCorrelation

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

Implementing the ExternalInvocation Interface

There are four basic steps in creating a custom service correlation:
1. Create a jar file that includes your code. For example:

**Listing 7-4   Sample Custom Service Correlation**

```java
package myPackageStructure;
import com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocation;
import com.bea.wlcp.wlng.api.servicecorrelation.ExternalInvocationException;

public class MyImplClass implements ExternalInvocation {
    public MyImplClass() {
    }

    public String pushServiceCorrelationID(String scID, String serviceName, String methodName, String spID, String appID, String appInstGrp) throws ExternalInvocationException {
        // your code here
        return scID;
    }

    public String getServiceCorrelationID(String scID, String serviceName, String methodName, String spID, String appID, String appInstGrp) throws ExternalInvocationException {
        // your code here
        return scID;
    }
}
```

2. Create the instancemap. See Class: `ExternalInvokerFactory`.
3. Put the instancemap file in the WLNG_NT.ear under APP-INF\classes. This makes your custom service correlation available to all traffic paths.

4. Put your jar file in the WLNG_NT.ear under APP-INF\lib.

**Parameter Tunneling**

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

The application sends the tunneled parameters in the SOAP header of a Web Services request. The tunneled parameter can be retrieved in a plug-in by the key. The parameter is fetched from the RequestContext, using the method `getXParam(String key)`. If a value for the key can not be found, null is returned.

**Listing 7-5  Get the value of the tunneled parameter ‘aParameterName’**.

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

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

**Storage Services**

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

- “ConfigurationStore” on page 7-16
- “StorageService” on page 7-20

**ConfigurationStore**

The Network Gatekeeper Core exposes a `ConfigurationStore` Java API that traffic paths can use to store simple configuration parameters instead of using JDBC and caching algorithms in each module.

All data stored in a `ConfigurationStore` is 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 Network Gatekeeper server (local). The domain wide store type replicates all data changes to all servers in the cluster, while the local store type keeps a different view of the parameters on different servers and data changes affect only the view for that particular server.
- Parameters stored in a ConfigurationStore are persisted to database.
- Data in all ConfigurationStores are also cached in memory.
- Only one instance of each named ConfigurationStore is cached in memory per server.
- Updates to a cluster wide named ConfigurationStore is reflected in all cluster nodes.
- The named ConfigurationStore only supports parameters of type Boolean, Integer, Long and String.

Interfaces

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

The entry point to configuration stores is through ServiceContext using the following method:

```java
public ConfigurationStore getConfigurationStore(String name, int storeType)
        throws ConfigurationException;
```

The ConfigurationStore service exposes an interface with the following features:

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

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

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

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

Listing 7-6 is an example of using the Configuration Store.

Listing 7-6   Example of a ConfigurationStoreHelper

```java
import com.incomit.slee.ServiceContext;
import com.bea.wlcp.wlng.api.storage.ConfigurationException;
import com.bea.wlcp.wlng.api.storage.ConfigurationStore;

public class ConfigurationStoreHelper {
    private static final String STORE_NAME = "my_store";
    private static final String MY_STRING_KEY = "my_string_key";
    private static final String MY_LONG_KEY = "my_long_key";
    private final ConfigurationStore myStore;
    private final ServiceContext serviceContext;

    private ConfigurationStoreHelper(ServiceContext sc) {
        serviceContext = sc;
    }

    private void init() throws ConfigurationException {
        myStore = serviceContext.getConfigurationStore(STORE_NAME,
```
ConfigurationStore.STORE_TYPE_SHARED);
myStore.initialize(MY_STRING_KEY, "default value");
myStore.initialize(MY_LONG_KEY, new Long(0));
}

public ConfigurationStoreHelper getInstance(ServiceContext sc)
throws ConfigurationException {
    if(m_instance == null) {
        m_instance = new ConfigurationStoreHelper(sc);
        m_instance.init();
    }
    return m_instance;
}

public String getMyString() throws ConfigurationException {
    return myStore.getString(MY_STRING_KEY);
}

public Long getMyLong() throws ConfigurationException {
    return myStore.getLong(MY_LONG_KEY);
}

public void setMyString(String value) throws ConfigurationException {
    myStore.setString(MY_STRING_KEY, value);
}

public void setMyLong(Long value) throws ConfigurationException {
The Configuration Store uses a database table, `slee_configuration`, to store the data.

**Note:** This table should only be modified through the ConfigurationStore APIs to avoid stale cache data.

<table>
<thead>
<tr>
<th>Table 7-2  Slee_configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column name</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>instance</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>parameter_key</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>value_type</td>
</tr>
</tbody>
</table>

The instance column holds the service name if the store type is shared, or the service instance name (unique for a service running in a particular server) if the store type is local.

**StorageService**

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

It provides mechanisms for:

- **Store initialization**

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

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

- Named queries

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

- Store listener

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

- Cluster locking

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

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

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

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

The storage service has a configuration file that defines the configuration for stores and the relationship between the cluster-wide store and the database table that backs the store. In the
configuration file it is also possible to define named queries towards the database table. The configuration file is shared between all extensions. The file `wlng-cache-store-config-extensions.xml` must be created and located in the network-tier EAR file, `wlng.nt.ear`, in the directory `APP-INF/ classes`. For details about the store configuration file, see the corresponding xsd: `wlng-cache-store-config.xsd`.

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

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

```java
Store<String, NotificationData> store = StoreFactory.getInstance().getStore(String.class, NotificationData.class);
try {
    notificationData = store.put(address.toString(), notificationData);
} finally {
    store.release();
}
```

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

**Listing 7-8  Retrieving a store by name**

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

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

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

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

**Listing 7-9  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">
```

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

Each Store has three sections:

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

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

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

The attribute db_table_name identifies the database definition to use.

store contains the following elements:

- identifier, which holds one classes elements. This element defines the classes for the key and the value that defines the store. The class for the key is defined in the attribute key-class and the class for the value part is defined in the attribute value-class. If a named store is used, the name is given in the element name.

- index, defines an index on the cache and one or more get methods. The methods maps to an index on the corresponding columns in the table and potentially a cache index if supported by the provider in use.

<db_table>

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

- key_column, which has the attributes name and data_type. The attribute name specifies the column name for the key and the attribute data_type specifies the SQL data type for the key.

- bucket_column, which has the attribute name. This attribute specifies the name of the column for the value part of the store. By default this is a BLOB. There is an optional attribute data_type, that can be used if other data types are used. This must be a Java to SQL supported data type mapping and corresponds to the data type in the value part of the store.

- value_column, which is used if attributes in the value part of the store should be stored in a separate column. The attribute name defines the name of the column and the data_type specifies the SQL data type for the column. value_column has the sub-element methods, which encloses the elements get_method and set_method. The sub-element methods
defines the names of the set and get methods for the data stored in `value_column` and the set and get methods for the attribute of the object in the store.

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

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

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

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

Listing 7-10  Example of a named query

```xml
<query name="com.bea.wlcp.wlng.plugin.example.netex.Query">
  <sql>
    <![CDATA[
      SELECT * FROM example_store_notification WHERE correlator = ?
    ]]>  
  </sql>
  <filter-class>com.acompany.plugin.example.netex.store.FilterImpl</filter-class>
</query>
```
Listing 7-11   Example of using the named query using a filter

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

Listing 7-12   Example of a filter implementation

```java
public class FilterImpl implements Filter {

    /**
     * The query parameters.
     */
    private Serializable[] parameters;

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

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

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

        return false;
    }

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

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

        return compareValue == parameters[0];
    }

    return false;
}

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

         this.parameters = parameters;
 }

<provider-mapping>

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

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

The <provider-name> entry references the type of store being used, see “<providers>” on page 7-30.

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

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

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

Using shared libraries

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

If there are such parts these should preferably not be packaged into the plug-in jar but instead be stored in the APP-INF/lib directory of wlng_nt.ear. All jars in this directory are be available for all plug-ins at runtime.
Annotations, EDRs, Alarms, and CDRs

The following section describes aspects and generation of EDRs, alarms, CDRs, and statistics:

- “About aspects and annotations” on page 8-2
- “How aspects are applied” on page 8-2
- “Context aspect” on page 8-3
- “EDR Generation” on page 8-7
  - “Exception scenarios” on page 8-8
  - “Adding data to the RequestContext” on page 8-10
  - “Trigger an EDR programmatically” on page 8-12
  - “EDR Content” on page 8-13
  - “RequestContext and EDR” on page 8-20
- “Categorizing EDRs” on page 8-21
  - “The EDR descriptor” on page 8-22
- “Check-list for EDR generation” on page 8-33
- “Alarm generation” on page 8-37
  - “Trigger an alarm programmatically” on page 8-37
  - “Alarm content” on page 8-38
Annotations, EDRs, Alarms, and CDRs

- “CDR generation” on page 8-42
  - “Triggering a CDR” on page 8-42
  - “Trigger a CDR programmatically” on page 8-42
  - “CDR content” on page 8-43
  - “Out-of-the box (OOTB) CDR support” on page 8-48
- “Extending Statistics” on page 8-49

About aspects and annotations

Aspects (pointcuts, advise, etc.) are written based on AspectJ 1.5.3 based annotation style. Aspects allow developers to manage cross-cutting concerns in their code in a straightforward and coherent way. 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 compiled Java packages. Minimal reflection is used at runtime to make aspect-based decisions. Different aspect types are applicable to different Network 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.

Here traffic flow is defined to only include plug-in implementations. Traffic aspects are subdivided into two categories:

- those that are always applied and
- those that are controlled using annotations.

Only statistics aspects are always applied in order to enforce licensing. Traffic aspect are applied to North and South boundaries of a plug-in as well as to the internal processing of the plug-in. Annotations are used to control the aspects that are not always applied for each plug-in. These annotations are defined as part of the functional areas that a given set of aspect implements. They allow the plug-in to communicate with the aspects as well as to customize their behavior.
**Context aspect**

The Context aspect is woven at compile time, using PluginSouth as a marker.

*Figure 8-1  Example of classes woven by aspects*

While requests coming from the north interface have a valid context (with attributes like Service provider account ID, application Account ID, and so on) any events triggered by the network and entering a plug-in’s south interface does not have a valid context.

The Context aspect solves this problem by rebuilding the context as soon as a south interface method is invoked: after this aspect is executed, a valid context will be available for any subsequent usages, for example in the EDR aspect. All methods inside a class implementing the interface PluginSouth are woven by the context aspect.

The Context aspect requires the following in order to correctly weave the south interface methods and be able to rebuild the context:

- Each ManagedPlugin must explicitly register its north and south interfaces.
- Each south interface must implement the `resolveAppInstanceGroupId()` and `prepareRequestContext()` methods of the PluginSouth interface.
- North interfaces must implement PluginNorth and south interfaces must implement PluginSouth (both of these interfaces extends Plugin).

The following rules apply for methods in classes that implements PluginNorth:
• The default behavior is that EDRs are triggered only for exceptions and callbacks to EJBs in 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 implements PluginSouth:

• Methods that performs 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 group ID using some plug-in specific lookup.

• Must implement resolveAppInstanceGroupId(ContextMapperInfo info) in PluginSouth and return the application instance group ID that corresponds to the original request to the network.

How to do this is plug-in-specific, but normally a network triggered request is tied to an application instance group in a cache that is managed by the plug-in.

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 group, is known since the request originated from an application.

2. The plug-in puts the application instance group ID and the ID for the request into a cache.

3. At a later stage, when a response to the original requests arrives to the plug-in, the method resolveAppInstanceGroupId() is called by aspects.

4. In this method, the plug-in must perform a lookup in the cache of the application instance group related to that request and return the application instance group ID to the aspect.

5. The aspect puts the application instance group ID in the RequestContext.

6. The method in the plug-in receives the request from the network and the RequestContext contains the application instance group ID.
In the example below the method deliver(...) is a request from the underlying network. The destinationAddress is annotated to be available by the aspect that handles network-triggered requests associated with this request represented by constant C.

NotificationHandler handles the cache for notifications and supplies all necessary parameters to the cache.

Listing 8-1  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 group 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 group ID is returned to the aspect and the execution flow continues in the plug-in, with a RequestContext that contains the application instance group ID, session ID and so on.
Listing 8-2  Rebuilding RequestContext

protected static final String C = "destinationAddress";

public String resolveAppInstanceGroupId(ContextMapperInfo info) {

    String destinationAddress = (String) info.getArgument(C);
    NotificationData notificationData = null;
    try {
        notificationData = StoreHelper.getInstance().getNotificationData(destinationAddress);
    } catch (StorageException e) {
        return null;
    }

    if (notificationData == null) {
        return null;
    }

    return notificationData.getAppInstanceGroupId();
}

Below are the steps you have to do to make your plug-in compliant with the Context aspect:

- Make sure to register all your PluginSouth objects within the ManagedPlugin before registering your plug-in in the PluginManager.
- Make sure to implement the resolveAppInstanceGroupId() method for each PluginSouth instance.
- Annotate each parameter in the south object methods that you need to have when aspect calls back the resolveAppInstanceGroupId() or the prepareRequestContext() methods. All
the annotated parameters will be available in the ContextMapperInfo parameter. The aspect need to have them annotated to be able to store them into the ContextMapperInfo object.

**EDR Generation**

EDRs are generated in the two following ways:

- automatically using aspects at given points in the traffic execution flow in a plug-in.
- manually anywhere in the code using the EdrService.

EDRs should be generated at the plug-in boundaries (north and south), using with 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 shall be in the range 500 000 to 999 999.

EDRs are generated automatically by an aspect in the following locations in the plug-in:

- Before and after any method annotated with @Edr
- Before and after any callback to an EJB
- After any exception is thrown

**Note:** Note that aspects are not applied outside the plug-in.

<table>
<thead>
<tr>
<th>Trigger</th>
<th>When</th>
<th>Modifiers restrictions</th>
<th>What is woven</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>before executing</td>
<td>public method only</td>
<td>only in methods annotated with @Edr</td>
</tr>
<tr>
<td>method</td>
<td>after executing</td>
<td>public method only</td>
<td>only in methods annotated with @Edr</td>
</tr>
<tr>
<td>method-call</td>
<td>before calling</td>
<td>any method</td>
<td>only for method call to a class implementing the PluginNorthCallback interface (EJB callback)</td>
</tr>
</tbody>
</table>
The following values are always available in the EDR when they are generated from an aspect:

- class name
- method name
- direction of the request (south, north)
- position (before, after)
- interface (north, south)
- source (method, exception)

### Exception scenarios

Exceptions are automatically woven by the aspect.

Some limitations applies:

- The aspect will catch only exceptions that are thrown out of a plug-in method.
- The aspect will not catch an exception that is thrown out by a library and caught by the plug-in.
- If the same exception is re-thrown several times, the aspect will only trigger an EDR once, for the first instance of the exception.

The diagram illustrates typical scenarios when a library (or core service) throws an exception in the plug-in.
Figure 8-2 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-throw 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 exception A but another exception, named B. In this case, because B is not the same instance as A, the aspect will trigger another EDR after the method in stage 1.
Adding data to the RequestContext

In addition to the default values, an EDR also contains all the values put into the RequestContext using the putEdr() method.

Listing 8-3  Example to add values to and EDR using RequestContext

```java
...  
RequestContext ctx = RequestContextManager.getCurrent();
// this value will be part of any EDRs generated in the current request
ctx.putEdr("address", "tel:1234");
// this value will NOT be part of any EDRs since ctx.put(...) is used
ctx.put("foo", "bar");
...
```

Note:  Common key names are defined in the class com.bea.wlcp.wlmg.api.edr.EdrConstants.

Using translators

When a parameter is a more complex object, it is possible to specify a translator that will take care of extracting the relevant information from this parameter.

The annotation is @ContextTranslate.

For example, the following method declares:

- The first (and only) parameter should be translated using the specified translator
  ACContextTranslator
- The returned object should also be translated using the specified translator
  ACContextTranslator

Listing 8-4  Using a translator

...
@Edr
   
   public @ContextTranslate(ACContextTranslator.class) PlayTextMessageResponse
   playTextMessage(@ContextTranslate(ACContextTranslator.class) PlayTextMessage
   parameters) {
   
   ...  
   return response;
   }
   ...

The Translator is a class implementing the ContextTranslator interface.

Listing 8-5  Example of a Translator

public class ACContextTranslator implements ContextTranslator {
   public void translate(Object param, ContextInfo info) {
      if(param instanceof PlayTextMessage) {
      PlayTextMessage msg = (PlayTextMessage) param;
      info.put("address", msg.getAddress().toString());
      } else if(param instanceof PlayTextMessageResponse) {
      PlayTextMessageResponse response = (PlayTextMessageResponse) param;
      info.put("correlator", response.getResult());
      } ...
   }
}

The ContextTranslator class specified in the @ContextTranslate annotation is automatically
instantiated by the aspect when needed. It is however possible to explicitly register it using the
ContextTranslatorManager.
Annotations, EDRs, Alarms, and CDRs

Listing 8-6  Example of registering a context translator

```java
ContextTranslatorManager.register(ACContextTranslator.class.getName(), new ACContextTranslator());
```

Below is a summary of annotations to use.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ContextKey</td>
<td>Annotation</td>
<td>Specifies that an argument must be put into the current RequestContext under the name provided in this annotation.</td>
</tr>
<tr>
<td>@ContextTranslate</td>
<td>Annotation</td>
<td>Same as @ContextKey but for complex argument that need to be translated using a translator (implementing the ContextTranslator interface).</td>
</tr>
<tr>
<td>ContextTranslator</td>
<td>Interface</td>
<td>Interface used by static translators to translate complex object.</td>
</tr>
</tbody>
</table>

**Trigger an EDR programmatically**

Network Gatekeeper triggers EDRs automatically in all plug-ins where aspects have been applied. It is also possible to trigger EDRs explicitly. In this case, you will have to manually create and trigger the EDR by following these steps:

1. Create an EdrData object
2. Trigger the EDR using the EdrService instance

Below is an example of how to trigger an EDR from inside a plug-in.

Listing 8-7  Trigger an EDR programmatically

```java
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
    // the WLNG 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
    EdrManager.getInstance().logEdr(data);

    ...
}

EDR Content

The following table describes the content of an EDR. It describes which values are mandatory, 
who is responsible for providing these values and other information.

Legends:

- A: Automatically provided by the WLNG
- H: Provided if the EdrDataHelper createData API is used to create the EdrData (which is 
  the recommended way)
- M: Provided manually in the EdrData
Annotations, EDRs, Alarms, and CDRs

- X: Provided in the edr.xml file.
- C: Custom filter. Use the element `<attribute>` to specify a custom filter.

**Note:** EDRs triggered by aspects will have all the mandatory fields provided by the aspect.

### Table 8-3 EDR content

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Filter tag name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdrId</td>
<td>To get the ID, use <code>getIdentifier()</code> in <code>EdrConfigDescriptor</code>. This value is provided in edr.xml file</td>
<td>C</td>
</tr>
</tbody>
</table>
|       | Provider INSIDE plug-in: X  
       | Provider OUTSIDE plug-in: X  
       | Mandatory: Yes               |
| ServiceName | The name (or type) of the service. Fields in `EdrConstants`: FIELD_SERVICE_NAME | C               |
|       | Provider INSIDE plug-in: H  
       | Provider OUTSIDE plug-in: M  
       | Mandatory: Yes               |
| ServerName | The name of the Network Gatekeeper server. Fields in `EdrConstants`: FIELD_SERVER_NAME | C               |
|       | Provider INSIDE plug-in: H  
       | Provider OUTSIDE plug-in: H  
       | Mandatory: Yes               |
### Table 8-3 EDR content

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Filter tag name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestamp</td>
<td>The time at which the EDR was triggered (in ms since midnight, January 1, 1970 UTC)</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: Yes</td>
<td></td>
</tr>
<tr>
<td>ContainerTransaction Id</td>
<td>The WebLogic Server transaction ID, if available.</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_CONTAINER_TRANSACTION_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Name of the class that triggered the EDR.</td>
<td>&lt;class&gt;</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_CLASS_NAME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: Yes</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Name of the method that triggered the EDR.</td>
<td>&lt;name&gt; inside &lt;method&gt; or &lt;method&gt; inside &lt;exception&gt;</td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: Yes</td>
<td></td>
</tr>
</tbody>
</table>
## Annotations, EDRs, Alarms, and CDRs

### Table 8-3 EDR content

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Filter tag name</th>
<th>Source</th>
<th>Direction</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Indicates the type of source that triggered the EDR.</td>
<td>&lt;method&gt; or</td>
<td>INSIDE</td>
<td></td>
<td>INSIDE</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_SOURCE</td>
<td>&lt;exception&gt;</td>
<td>OUTSIDE</td>
<td></td>
<td>OUTSIDE</td>
</tr>
<tr>
<td></td>
<td>Values in EdrConstants: VALUE_SOURCE_METHOD,</td>
<td></td>
<td></td>
<td>VALUE_DIRECTION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VALUE_SOURCE_EXCEPTION</td>
<td></td>
<td></td>
<td>SOUTH</td>
<td>BEFORE</td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
<td></td>
<td>NORTH</td>
<td>AFTER</td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direction</td>
<td>Direction of the request.</td>
<td>&lt;direction&gt;</td>
<td>INSIDE</td>
<td></td>
<td>INSIDE</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_DIRECTION</td>
<td></td>
<td>OUTSIDE</td>
<td></td>
<td>OUTSIDE</td>
</tr>
<tr>
<td></td>
<td>Values in EdrConstants: VALUE_DIRECTION_SOUTH,</td>
<td></td>
<td></td>
<td>VALUE_DIRECTION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VALUE_DIRECTION_NORTH</td>
<td></td>
<td></td>
<td>SOUTH</td>
<td>BEFORE</td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
<td></td>
<td>NORTH</td>
<td>AFTER</td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>Position of the EDR relative to the method that triggered the EDR.</td>
<td>&lt;position&gt;</td>
<td>INSIDE</td>
<td></td>
<td>INSIDE</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_POSITION</td>
<td></td>
<td>OUTSIDE</td>
<td></td>
<td>OUTSIDE</td>
</tr>
<tr>
<td></td>
<td>Values in EdrConstants: VALUE_POSITION_BEFORE,</td>
<td></td>
<td></td>
<td>VALUE_POSITION</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VALUE_POSITION_AFTER</td>
<td></td>
<td></td>
<td>BEFORE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-3 EDR content

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Filter tag name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Interface where the EDR is triggered.</td>
<td>&lt;interface&gt;</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_INTERFACE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Values in EdrConstants: VALUE_INTERFACE_NORTH,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VALUE_INTERFACE_SOUTH, VALUE_INTERFACE_OTHER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>Exception</td>
<td>Name of the exception that triggered the EDR.</td>
<td>&lt;name&gt; inside</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_EXCEPTION_NAME</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>SessionId</td>
<td>Session ID.</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_SESSION_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>ServiceProviderId</td>
<td>Service provider account ID.</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_SP_ACCOUNT_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8-3 EDR content

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Filter tag name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationId</td>
<td>Application account ID.</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_APP_ACCOUNT_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>AppInstanceId</td>
<td>Application instance group ID.</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_APP_INSTANCE_GROUP_ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
<tr>
<td>OrigAddress</td>
<td>The originating address with scheme included (for example “tel:1234”).</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Fields in EdrConstants: FIELD_ORIGINATING_ADDRESS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider INSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Provider OUTSIDE plug-in: M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mandatory: No</td>
<td></td>
</tr>
</tbody>
</table>
Using send lists

If more than one address need to be stored in the DestAddress field, use the following pattern. Both patterns described below can be used.

Listing 8-8 Pattern to store one single or multiple addresses in field destination directly on EdrData.

```java
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 the array of addresses. The EdrDataHelper will automatically take care of converting this array into the EdrData.

Listing 8-9  Pattern to store one single or multiple addresses in field destination using RequestContext.

RequestContext ctx = RequestContextManager.getCurrent();
// If there is only one address
ctx.putEdr(EdrConstants.FIELD_DESTINATION_ADDRESS, address);
// If there are multiple addresses
URI[] addresses = ...;
ctx.putEdr(EdrConstants.FIELD_DESTINATION_ADDRESS, Arrays.asList(addresses));

RequestContext and EDR

The following diagram shows how and where information for the EDR is added to the RequestContext and how it finally ends up in the additional info column of the alarm and CDR databases.

Figure 8-3  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 the one that are put into the ContextInfo object.

Any information put in the RequestContext parameter of the PluginSouth.prepareRequestContext() method.

When an EDR is created, the EdrDataHelper (which is the recommended way to create the EDR) will populate the EdrData with all the key/value pairs found in the RequestContext.

When the EdrService writes the alarm or CDR additional information content into the database, it will use all the EdrData key/value pairs EXCEPT a set of well-known keys that are either not relevant or already included in other columns of the database, see “Alarm content” on page 8-38 and “CDR content” on page 8-43.

Categorizing EDRs

Only one type of EDRs exists: alarms and CDRs are subsets of these EDRs. In order to categorize the flow of EDRs as either a pure EDRS, alarms or CDRs, the new EDR service uses 3 xml configuration files:

- edr.xml: this file contains descriptors that describe pure EDRs.
- alarm.xml: this file contains descriptors that describe EDRs that should be considered alarms.
- cdr.xml: this file contains descriptors that describe EDRs that should be considered CDRs.

These XML configuration files are located in 'APP-INF/classes/edr/' as well as the validation file edr-config.xsd.
The EDR descriptor

Each XML configuration file contains a list of EDR descriptors that define an EDR as a pure-EDR, as an alarm or as a CDR.

Table 8-4 EDR descriptors.

<table>
<thead>
<tr>
<th>File</th>
<th>Descriptor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edr.xml</td>
<td>&lt;edr...&gt;</td>
<td>Define which EDRs are pure EDRs</td>
</tr>
<tr>
<td>alarm.xml</td>
<td>&lt;alarm...&gt;</td>
<td>Define which EDRs are alarms</td>
</tr>
<tr>
<td>cdr.xml</td>
<td>&lt;edr...&gt;</td>
<td>Define which EDRs are CDRs</td>
</tr>
</tbody>
</table>

The descriptor is composed of two parts:

- The `<filter>` element: this is the filter
- The `<data>` element: this part is used to attach additional data with the EDR if it is matched by the `<filter>` part

The following table describes the elements allowed in the `<filter>` part:

Table 8-5 Elements allowed in `<filter>` part of an EDR descriptor.

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Min occurs</th>
<th>Max occurs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;method&gt;</code></td>
<td></td>
<td>0</td>
<td>unbounded</td>
<td>Filter EDR triggered by a method</td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td></td>
<td>0</td>
<td>unbounded</td>
<td>Name of the method that triggered the EDR</td>
</tr>
<tr>
<td><code>&lt;class&gt;</code></td>
<td></td>
<td>0</td>
<td>unbounded</td>
<td>Name of the class that triggered the EDR</td>
</tr>
<tr>
<td><code>&lt;direction&gt;</code></td>
<td></td>
<td>0</td>
<td>2</td>
<td>Direction of the request</td>
</tr>
<tr>
<td><code>&lt;interface&gt;</code></td>
<td></td>
<td>0</td>
<td>3</td>
<td>Interface where the EDR has been triggered</td>
</tr>
<tr>
<td><code>&lt;position&gt;</code></td>
<td></td>
<td>0</td>
<td>2</td>
<td>Position relative to the method that triggered the EDR</td>
</tr>
<tr>
<td><code>&lt;exception&gt;</code></td>
<td></td>
<td>0</td>
<td>unbounded</td>
<td>Filter EDR triggered by an exception</td>
</tr>
</tbody>
</table>
The following table describes the values allowed for each element of the `<filter>` part:

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Min occurs</th>
<th>Max occurs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;name&gt;</code></td>
<td>0</td>
<td>unbounded</td>
<td></td>
<td>Name of the exception that triggered the EDR</td>
</tr>
<tr>
<td><code>&lt;class&gt;</code></td>
<td>0</td>
<td>unbounded</td>
<td></td>
<td>Name of the class where the exception was thrown</td>
</tr>
<tr>
<td><code>&lt;method&gt;</code></td>
<td>0</td>
<td>unbounded</td>
<td></td>
<td>Name of the method where the exception was thrown</td>
</tr>
<tr>
<td><code>&lt;direction&gt;</code></td>
<td>0</td>
<td>2</td>
<td></td>
<td>Direction of the request</td>
</tr>
<tr>
<td><code>&lt;interface&gt;</code></td>
<td>0</td>
<td>3</td>
<td></td>
<td>Interface where the EDR has been triggered</td>
</tr>
<tr>
<td><code>&lt;position&gt;</code></td>
<td>0</td>
<td>2</td>
<td></td>
<td>Position relative to the method that triggered the EDR</td>
</tr>
<tr>
<td><code>&lt;attribute&gt;</code></td>
<td>0</td>
<td>unbounded</td>
<td></td>
<td>Filter EDR by looking at custom attribute</td>
</tr>
<tr>
<td><code>&lt;key&gt;</code></td>
<td>1</td>
<td>1</td>
<td></td>
<td>Name of the key</td>
</tr>
<tr>
<td><code>&lt;value&gt;</code></td>
<td>1</td>
<td>1</td>
<td></td>
<td>Value</td>
</tr>
</tbody>
</table>

The following table describes the values allowed for each element of the `<filter>` part:

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Allowed values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;method&gt;</code></td>
<td><code>&lt;name&gt;</code></td>
<td>“return type name of method([args])”</td>
<td>Method name. The arguments can be omitted with the parenthesis. See Special characters below.</td>
</tr>
<tr>
<td><code>&lt;class&gt;</code></td>
<td></td>
<td>“full name of class”</td>
<td>Fully qualified class name. See Special characters below.</td>
</tr>
<tr>
<td><code>&lt;direction&gt;</code></td>
<td></td>
<td>“south”, “north”</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-5 Elements allowed in `<filter>` part of an EDR descriptor.

Table 8-6 Values allowed in each element of the `<filter>` part.
**Annotations, EDRs, Alarms, and CDRs**

**Table 8-6 Values allowed in each element of the <filter> part.**

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Allowed values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;interface&gt;</td>
<td></td>
<td>“north”, “south”, “other”</td>
<td></td>
</tr>
<tr>
<td>&lt;position&gt;</td>
<td></td>
<td>“before”, “after”</td>
<td></td>
</tr>
<tr>
<td>&lt;exception&gt;</td>
<td>&lt;name&gt;</td>
<td>“fullnameofexceptionclass”</td>
<td>Fully qualified exception class name. See Special characters below.</td>
</tr>
<tr>
<td>&lt;class&gt;</td>
<td></td>
<td>“fullnameofclass”</td>
<td>Fully qualified class name where the exception was triggered. See Special characters below.</td>
</tr>
<tr>
<td>&lt;method&gt;</td>
<td></td>
<td>“returntype nameofmethod([args])”</td>
<td>Method name. The arguments can be omitted with the parenthesis See Special characters below.</td>
</tr>
<tr>
<td>&lt;direction&gt;</td>
<td></td>
<td>“south”, “north”</td>
<td></td>
</tr>
<tr>
<td>&lt;interface&gt;</td>
<td></td>
<td>“north”, “south”, “other”</td>
<td></td>
</tr>
<tr>
<td>&lt;position&gt;</td>
<td></td>
<td>“before”, “after”</td>
<td></td>
</tr>
<tr>
<td>&lt;attribute&gt;</td>
<td>&lt;key&gt;</td>
<td>“astring”</td>
<td></td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td></td>
<td>“astring”</td>
<td></td>
</tr>
</tbody>
</table>

**Special characters**

The filter uses special characters to indicate more precisely how to match certain values.

Using * at the end of a method, class or exception name matches all names that match the string specified prior to the * (that is, what the string starts with).

**Note:** The usage of any of these characters disables the caching of the filter containing them. It is strongly encouraged to use other way of matching if possible to avoid performance hit.
Values provided

The exact value in these fields depends on who triggered the EDR. If the aspect triggered the EDR, then the name of the method (with return type and parameters) or the fully qualified name of the class/exception is indicated. If the EDR is manually triggered from the code, it is up to the implementer to decide what name to use. Here are some examples of fully qualified method/class names as specified by the aspect:

Example methods:

```java
SendSmsResponse sendSms(SendSms)
```
Annotations, EDRs, Alarms, and CDRs

```java
void receivedMobileOriginatedSMS(NotificationInfo, boolean, SmsMessageState, String, SmsNotificationRemote)
TpAppMultiPartyCallBack reportNotification(TpMultiPartyCallIdentifier, TpCallLegIdentifier[], TpCallNotificationInfo, int)

Example Class:
com.bea.wlcp.wlng.plugin.sms.smpp.SMPPManagedPluginImpl

Boolean semantic of the filters
The following diagram shows briefly how the filter works:

- The EdrConfigSource elements are the following: <method>, <exception> or <attribute>. They are combined using OR.
- The filter elements of each EdrConfigSource are combined using AND. However, if the same filter is available more than once (e.g. multiple class names), they are combined with OR.

Figure 8-4 Filter mechanism
Example filters

Example 1: filter

The following filter will categorize EDRs as pure EDRs with an id of 1000 when the following conditions are met:

- The class where the method triggered the EDR is `com.bea.wlcp.wlng.plugin.AudioCallPlugin` or any subclass of it.
- AND the request is southbound (direction = south)
- AND the interface where the EDR was trigger is north
- AND the EDR has been triggered after the method has been executed (position = after)

Listing 8-10  Example 1: filter

```xml
<edr id="1000" description="...">
  <filter>
    <method>
      <class>com.bea.wlcp.wlng.plugin.AudioCallPlugin</class>
      <direction>south</direction>
      <interface>north</interface>
      <position>after</position>
    </method>
  </filter>
</edr>
```

Example 2: Alarm filter

The following filter will categorize EDRs into alarms when the following conditions are met:

- The exception is the class `com.bea.wlcp.wlng.plugin.PluginException` or a subclass of it.
Annotations, EDRs, Alarms, and CDRs

- 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 alarm.xml file has a <alarm-group> element that is used to group alarms by service/source: this group id and each individual alarm id is used to generate the OID of SNMP traps.

**Listing 8-11  Example 3: filter**

```xml
<alarm-group id="104" name="parlayX" description="Parlay X alarms">
  <alarm id="1000" severity="minor" description="Parlay X exception">
    <filter>
      <exception>
        <name>com.bea.wlcp.wlng.plugin.PluginException</name>
        <name>org.csapi*</name>
      </exception>
    </filter>
  </alarm>
</alarm-group>
```

**Example 3: Alarm filter**

The following filter will categorize EDRs into alarms when the following conditions are met:

- The exception is the class com.bea.wlcp.wlng.plugin.PluginException or a subclass of it
- OR the name of the exception starts with “org.csapi”. String matching in used.
- AND the exception was triggered in a class whose name starts with com.bea.wlcp.wlng.plugin
- AND the request is northbound (direction = north) when the exception was triggered

If the filter determines that the EDR is an alarm, the following attributes are available to the alarm listener (they are defined in the <data> part):

- identifier = 123
Categorizing EDRs

- source = wlng_nt1

**Listing 8-12  Example 3: filter**

```xml
<alarm id="1000" severity="minor" description="Parlay X exception">
  <filter>
    <exception>
      <name>com.bea.wlcp.wlng.plugin.PluginException</name>
      <name>org.csapi*</name>
      <class>com.bea.wlcp.wlng.plugin*</class>
      <direction>north</direction>
    </exception>
  </filter>
  <data>
    <attribute key="identifier" value="123"/>
    <attribute key="source" value="wlng_nt1"/>
  </data>
</alarm>
```

**Example 4: filter**

The following filter (for example purposes only) will categorize EDRs into 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.
Annotations, EDRs, Alarms, and CDRs

- OR the name of the exception that triggered the EDR starts with `com.bea.wlcp.wlng.bar` AND the exception was triggered in a plug-in north interface
- OR the name of the exception that triggered the EDR starts with `com.bea.wlcp.wlng.plugin.exceptionA` AND the exception was triggered in a class whose name starts with `com.bea.wlcp.wlng.plugin.classD` AND the exception was triggered in a method whose name starts with `void com.bea.wlcp.wlng.plugin.methodA` AND the exception was triggered in a plug-in north interface
- OR the EDR contains an attribute with key `attribute_a` and value `value_a`
- OR the EDR contains an attribute with key `attribute_b` and value `value_b`

Listing 8-13  Example 4: filter

```xml
<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>
  </filter>
</edr>
```
Example 5: filter with corresponding code for manually triggering a matching EDR

The following example shows a manually triggered EDR with its corresponding filter. The EDR is triggered using these lines.

Listing 8-14  Example 5: Trigger the EDR

```java
// 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 the WLNG 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);
```
Annotations, EDRs, Alarms, and CDRs

```java
    data.setValue(EdrConstants.FIELD_METHOD_NAME, "myMethodName");
    data.setValue("myKey", "myValue");

    // Log the EDR
    EdrManager.getInstance().logEdr(data);
    ...
    }

This EDR can be filtered using the following filter (note the various way of identifying this EDR):

Listing 8-15  Example: Filter 5

```xml
<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>
  </filter>
</edr>

Check-list for EDR generation

Below is a list of steps you have to do to make your plug-in able to take advantage of the aspect EDR:

- Make sure to register all your PluginNorth (and south) objects within the ManagedPlugin before registering your plug-in in the PluginManager.
- Annotate all the methods you want to be woven using the @Edr annotation.
- Annotate the specific arguments you want to see in the EDR for each annotated methods. Use either @ContextKey or @ContextTranslate depending on the kind of argument.
- Add to the edr.xml file all the EDR you are triggering, either manually or with the @Edr annotation. This is the only way to customize alarms and CDRs.
- If external EDR listeners, CDR, and alarms are used, the file edrjmslistener.jar needs to be updated on all the listeners. Replace the existing edr.xml, alarm.xml, and cdr.xml file with the updated file in the edr directory in edrjmslistener.jar.

Frequently Asked Questions about EDRs and EDR filters

**Question:** Is it possible to specify both exception and method name in the filter section?

**Listing 8-16  Example: method name and exception in a filter.**

```xml
<filter>
  <method>
    <name>internalSendSms</name>
  </method>
</filter>
```
Annotations, EDRs, Alarms, and CDRs

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 only have one of the methods to generate the EDR and map that edr to an alarm?

Listing 8-17  Re-throwing an exception

myMethodA()throws MyException{
    myMethodB();
}

myMethodB()throws MyException{
    myMethodC();
}

myMethodC()throws MyException{
    ...
    //on error
Frequently Asked Questions about EDRs and EDR filters

```java
throw new MyException("Exception text..");
}
```

**Answer**

In this case, only the first exception will be caught by aspect. Or more precisely, they will all be caught by aspect but will only trigger an EDR for the first one, 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 aspect detect the following exception?**

Listing 8-18  Example exception

```java
try{
    throw new ReceiverConnectionFailureException(message);
}catch(ReceiverConnectionFailureException connfail){
    //EDR-ALARM-MAPPING
}
```

**Answer**

This exception will NOT be detected by the 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: EDRs for exceptions will also work for private methods, or?**

**Answer**

Yes, it will work for any methods.

**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?**
Can an alarm be generated if a request to with more than 12 destination addresses? How can information be added to the alarm about how many addresses that where included in the request?

It is possible to specify some info in the alarm.xml file with something like

```xml
<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 shown below.

```java
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 only relevant information.

Q: Is it possible to specify classname in the filtering section?

Answer

Yes, use the <class> tag inside <method> or <exception> in the filter.

```xml
<filter>
    <exception>
        <class>com.y.y.z.MyClass</class>
        <name>com.x.y.z.MyException</name>
    </exception>
</filter>
```
Alarm generation

An alarm is a subset of an EDR. To generate an alarm, generate an EDR, either using one generated in aspects or programmatically and define the ID, and the descriptor of the alarm in alarm.xml.

The alarm ID, severity, description and other kind of attributes are defined in the alarm.xml file, see “The EDR descriptor” on page 8-22. For extensions, the alarm ID shall be in the range 500 000 to 999 999.

Note: The alarm filter that provides the first match in alarm.xml 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.xml file.
- Programmatically trigger an EDR and add its descriptor in both the alarm.xml file and edr.xml. Make sure the ID of the alarm is unique and that the descriptor is the same as in edr.xml.

In order for the alarm to be displayed in the Alarms tab in the Network Gatekeeper Administration Console, or if external alarm listeners are used, the file $DOMAIN_HOME/edr/edrjmslistener.jar needs to be updated on all the servers in the network tier cluster. Replace the existing alarm.xml file with the updated file in the edr directory in edrjmslistener.jar.

Trigger an alarm programmatically

Trigger an EDR as described in “EDR Content” on page 8-13. Then specify in the alarm.xml file the descriptor necessary for this EDR to be considered an alarm.

**Listing 8-19  Example code to trigger an alarm**

```java
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");
```
Annotations, EDRs, Alarms, and CDRs

data.setValue("myAdditionalInformation", ...);
EdrManager.getInstance().logEdr(data);
...

The corresponding descriptor, in the alarm.xml file, that matches this EDR is shown below.

Listing 8-20  Alarm descriptor

<alarm id="2006"
       severity="major"
       description="Sample alarm">
  <filter>
    <method>
      <name>com.bea.wlcp.wlng.myMethod</name>
      <class>com.bea.wlcp.wlng.myClass</class>
    </method>
  </filter>
</alarm>

Alarm content

Below is a list of which alarm information is provided in alarms.
Table 8-8  Alarm information for alarm listeners, also stored in DB

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>alarm_id</td>
<td>Unique ID for the alarm. Automatically provided by the EdrService.</td>
</tr>
<tr>
<td>source</td>
<td>Service name emitting the alarm. Automatically provided by the EdrService.</td>
</tr>
<tr>
<td>timestamp</td>
<td>Timestamp in milliseconds since midnight, January 1, 1970 UTC. Automatically provided by the EdrService.</td>
</tr>
<tr>
<td>severity</td>
<td>Severity level. Defined in the alarm.xml configuration file.</td>
</tr>
<tr>
<td>identifier</td>
<td>The alarm identifier. Defined in the alarm.xml configuration file. The column in the database will always contain the identifier defined in the alarm.xml file. The identifier value for the backwards compatible alarm listener can be different. If a backwards compatible plug-in triggers an alarm, the identifier specified in the plug-in code will be used in the backwards compatible alarm listener. If an enhanced plug-in is triggering an alarm, the backwards compatible alarm listener will receive the identifier coming from the alarm.xml file, as expected.</td>
</tr>
</tbody>
</table>
Annotations, EDRs, Alarms, and CDRs

Table 8-8 Alarm information for alarm listeners, also stored in DB

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>alarm_info</td>
<td>The alarm information or description.</td>
</tr>
<tr>
<td></td>
<td>Defined in the alarm.xml configuration file.</td>
</tr>
<tr>
<td></td>
<td>If the alarm is triggered from a backwards compatible plug-in, the description will be as it was in Network Gatekeeper 2.2.</td>
</tr>
<tr>
<td></td>
<td>For enhanced plug-ins, the description will come from the alarm.xml file.</td>
</tr>
</tbody>
</table>

Alarm generation

The format of this field has changed in Network Gatekeeper 3.0. Each entry is now formatted as:
key=value

Similar to the Java properties file.

All the custom key/value pairs found in the EdrData except these are present (EdrConstants if not specified):
• FIELD_TIMESTAMP
• FIELD_SERVICE_NAME
• FIELD_CLASS_NAME
• FIELD_METHOD_NAME
• FIELD_SOURCE
• FIELD_DIRECTION
• FIELD_POSITION
• FIELD_INTERFACE
• FIELD_EXCEPTION_NAME
• FIELD_ORIGINATING_ADDRESS
• FIELD_DESTINATION_ADDRESS
• FIELD_CONTAINER_TRANSACTION_ID
• FIELD_CORRELATOR
• FIELD_SESSION_ID
• FIELD_SERVER_NAME
• ExternalInvokerFactory.SERVICE_CORRELATION_ID
• FIELD_BC_EDR_ID
• FIELD_BC_EDR_ID_3
• FIELD_BC_ALARM_IDENTIFIER
• FIELD_BC_ALARM_INFO

Table 8-8 Alarm information for alarm listeners, also stored in DB

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>additional_info</td>
<td>Automatically provided by the EdrService.</td>
</tr>
<tr>
<td></td>
<td>Not valid for backwards compatible alarm listeners.</td>
</tr>
</tbody>
</table>
|            | The format of this field has changed in Network Gatekeeper 3.0. Each entry is now formatted as: key=value
|            | key=value
|            | 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_INTERFACE                                        |
|            | • FIELD_EXCEPTION_NAME                                   |
|            | • FIELD_ORIGINATING_ADDRESS                              |
|            | • FIELD_DESTINATION_ADDRESS                              |
|            | • FIELD_CONTAINER_TRANSACTION_ID                         |
|            | • FIELD_CORRELATOR                                       |
|            | • FIELD_SESSION_ID                                       |
|            | • FIELD_SERVER_NAME                                     |
|            | • ExternalInvokerFactory.SERVICE_CORRELATION_ID         |
|            | • FIELD_BC_EDR_ID                                        |
|            | • FIELD_BC_EDR_ID_3                                      |
|            | • FIELD_BC_ALARM_IDENTIFIER                              |
|            | • FIELD_BC_ALARM_INFO                                    |
CDR generation

A CDR is a subset of an EDR. To generate an CDR, generate an EDR and define the ID of the EDR in cdr.xml.

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 descriptor to the cdr.xml file.
- Programatically trigger an EDR and add its descriptor in the cdr.xml file.

Trigger a CDR programmatically

If none of the existing EDR is appropriate for a CDR, you can programmatically trigger an EDR that will become a CDR. See the section, “Trigger an EDR programmatically” on page 8-12 for information on how to create and trigger an EDR. Specify in the cdr.xml file the descriptor necessary for this EDR to be considered a CDR.

Listing 8-21  Example, triggering a CDR

```java
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, ...);
...
EdrManager.getInstance().logEdr(data);
...```

Annotations, EDRs, Alarms, and CDRs
The descriptor, in the cdr.xml file, that matches this EDR is shown below.

Listing 8-22  Filter to match the EDR

```xml
<cdr>
  <filter>
    <method>
      <name>com.bea.wlcp.wlng.myEndOfRequestMethod</name>
      <class>com.bea.wlcp.wlng.myClass</class>
    </method>
  </filter>
</cdr>
```

**CDR content**

In addition to the EDR fields, the following table lists the specific fields used only for CDRs.

<table>
<thead>
<tr>
<th>Field in EdrConstants</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD_CDR_SESSION_ID</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_START_OF_USAGE</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_CONNECT_TIME</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_END_OF_USAGE</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_DURATION_OF_USAGE</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_AMOUNT_OF_USAGE</td>
<td></td>
</tr>
</tbody>
</table>
The CDR content is aligned toward the 3GPP Charging Applications specifications. The consequence is that the database schema is changed to fit these changes and to accommodate future extensions.

Legends:

- NU: Not used
- NC: New column in DB
- RC: Renamed column in DB

### Table 8-9 Fields in EdrConstants specific for CDRs.

<table>
<thead>
<tr>
<th>Field in EdrConstants</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD_CDR_ORIGINATING_PARTY</td>
<td></td>
</tr>
<tr>
<td>FIELD_CDR_DESTINATION_PARTY</td>
<td>Same pattern applies as for send lists, see “Using send lists” on page 8-19.</td>
</tr>
<tr>
<td>FIELD_CDR_CHARGING_INFO</td>
<td>In Network Gatekeeper 3.0, this field is a string a string. In backwards-compatible traffic paths, it is converted back into a long as required.</td>
</tr>
</tbody>
</table>

### Table 8-10 Content in database

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
<th>3.0</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_id</td>
<td>Unique id for the CDR. Provided automatically by the EDR service.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>service_name</td>
<td>name of the service Provided automatically by the EDR service.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>service_provider</td>
<td>the service provider account ID Provided automatically by the EDR service.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>application_id</td>
<td>the application account ID (was user_id in 2.2)</td>
<td>x</td>
<td>RC</td>
</tr>
<tr>
<td>application_instance_grp_id</td>
<td>the application instance group ID.</td>
<td>x</td>
<td>NC</td>
</tr>
</tbody>
</table>
### Table 8-10 Content in database

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
<th>Comment</th>
<th>3.0</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>container_transaction_id</td>
<td>id of the current user transaction Provided automatically by the EDR service.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>server_name</td>
<td>name of the server that generated the CDR. Provided automatically by the EDR service.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td>in ms since midnight, January 1, 1970 UTC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>service_correlation_id</td>
<td>Service Correlation ID. Provided automatically by the EDR service.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>charging_session_id</td>
<td>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.</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start_of_usage</td>
<td>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.</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>connect_time</td>
<td>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.</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end_of_usage</td>
<td>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</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 8-10  Content in database

<table>
<thead>
<tr>
<th>Field</th>
<th>Comment</th>
<th>3.0</th>
<th>DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration_of_usage</td>
<td>The total time the service capability module used the network services (in ms)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amount_of_usage</td>
<td>Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>originating_party</td>
<td>The originating party address with scheme included (e.g. “tel:1234”)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>destination_party</td>
<td>the originating party address with scheme included (e.g. “tel:1234”). Additional addresses are stored in the additional_info field.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>charging_info</td>
<td>The charging service code from the application.</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Plug-in specific. Plug-in needs to put the value into the RequestContext of the request that will trigger the CDR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>additional_info</td>
<td>Additional information provided by the plug-in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>revenue_share_percentage</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>party_to_charge</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slee_instance</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>network_transaction_id</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>network_plugin_id</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transaction_part_number</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion_status</td>
<td>Not used.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Additional_info column

The EDR populates the additional_info column of the DB with all the custom key/value pairs found in the EdrData except the ones listed below.

Excluded keys (EdrConstants if not specified):

- FIELD_SERVICE_NAME
- FIELD_APP_INSTANCE_GROUP_ID
- FIELD_SP_ACCOUNT_ID
- FIELD_CONTAINER_TRANSACTION_ID
- FIELD_SERVER_NAME
- FIELD_TIMESTAMP
- ExternalInvocatorFactory.SERVICE_CORRELATION_ID
- FIELD_CDR_SESSION_ID
- FIELD_CDR_START_OF_USAGE
- FIELD_CDR_CONNECT_TIME
- FIELD_CDR_END_OF_USAGE
- FIELD_CDR_DURATION_OF_USAGE
- FIELD_CDR_AMOUNT_OF_USAGE
- FIELD_CDR_ORIGINATING_PARTY
- FIELD_CDR_DESTINATION_PARTY
- FIELD_CDR_CHARGING_INFO
- FIELD_CLASS_NAME
- FIELD_METHOD_NAME
- FIELD_SOURCE
- FIELD_DIRECTION
- FIELD_POSITION
Annotations, EDRs, Alarms, and CDRs

- FIELD_INTERFACE
- FIELD_EXCEPTION_NAME
- FIELD_ORIGINATING_ADDRESS
- FIELD_DESTINATION_ADDRESS
- FIELD_CORRELATOR
- FIELD_APP_ACCOUNT_ID
- FIELD_SESSION_ID
- FIELD_BC_EDR_ID
- FIELD_BC_EDR_ID_3
- FIELD_BC_ALARM_IDENTIFIER
- FIELD_BC_ALARM_INFO

Two keys not present in the EdrData are added to additional_info.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>destinationParty</td>
<td>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</td>
</tr>
<tr>
<td>oldInfo</td>
<td>Any backwards compatible additional info is available</td>
</tr>
</tbody>
</table>

The format of the additional_info field is formatted as:

```
key=value
```

similar to the Java properties file.

**Out-of-the-box (OOTB) CDR support**

It is difficult to come up with a CDR generation scheme which will fulfill the requirements of all customers. Network Gatekeeper generates a default set of CDRs which can be customized by re-configuring cdr.xml.
The guiding principle for when to generate CDRs is:

- Generate a CDR when we are 100% sure that we have completely handled the service request

In other words, after the last method, in a potential sequence of method calls, returns.

This means that for network-triggered requests will generate a CDR at the south interface after the method has returned back to the network. For application-triggered requests we generate a CDR at the north interface after the method has returned to the Network Tier SLSB.

**Extending Statistics**

Aspects are also used to generate statistics. To add a new statistic type to your traffic path requires two steps:

1. You must add a new statistic type, using the `addStatisticType` operation in the Management Console. For more information, see “Managing and Configuring Statistics and Transaction Licenses” in the *System Administration Guide*.

2. The statistics aspect is automatically applied to all public methods at `PluginNorth`. By default extension traffic paths generate information identified with the transaction type `TRANSACTION_TYPE_EXTENSION`. To generate more specific types, annotate your code with `@Statistics(id=<My_Statistics_Type>)`

   For extensions, the statistics ID shall be in the range 1000 to 2250.
Annotations, EDRs, Alarms, and CDRs
Making Traffic Paths Manageable

Once you have created your extension traffic path, any OAM functions that you have designed - read/write attributes and/or operations - must be exposed in a way that allows them to be accessed and manipulated, either through the Network Gatekeeper Console extension, or through other management tools. The following chapter provides a description of the mechanism that Network Gatekeeper uses to accomplish this.

Overview

WebLogic Network Gatekeeper uses the Java Management Extensions (JMX) 1.2 standard, as it is implemented in JDK 1.5. The JMX model consists of three layers, Instrumentation, Agent, and Distributed Services. As a traffic path developer, you work in the Instrumentation layer. You create managed beans (MBeans) that expose your traffic path management functionality as a management interface. These MBeans are then registered with the Agent, the Runtime MBean Server in the WebLogic Server instance, which makes the functionality available to the Distributed Services layer, management tools like the Network Gatekeeper Management Console. Finally, because configuration information needs to be persisted, you store the values you set using Network Gatekeeper’s Configuration Store, which provides a write-through database cache. In addition to persisting the configuration information, the cache also provides cluster-wide access to the data, updating a cluster-wide store whenever there is a change in globally relevant configuration data.

For more information on the JMX model in general in relation to WebLogic Server, see Developing Manageable Applications with JMX.
Create Standard JMX MBeans

Creating standard MBeans is a two step process.

1. Create a Java Interface
2. Implement the Interface

Create a Java Interface

The first thing you need to do is to create an interface file that describes getter and setter methods for each class attribute that is to be exposed through JMX (getter only for read-only attributes; setter only for write-only) and a wrapper operation for each class method to be exposed. The attribute names should be the case-sensitive names that you wish to see displayed in the UI of the Console extension. See Listing 9-1

Listing 9-1 A snippet of a Sample MBean Interface

```java
/**
 * A simple MBean for exposing the management information
 **/
public interface TestMBean{
    @Attr("<<Put a meaningful description for the attribute here>>")
    public int RepeatIndicator = 0;
    @Attr("<<Put a meaningful description for the attribute here>>")
    public int ResponseRequested = 0;
    //setters and getters
    public int getRepeatIndicator();
    public void setRepeatIndicator(int value);
    public void setResponseRequested(int value);
    public int getResponseRequested();
    //Operations
```
Create Standard JMX MBeans

@Description("<<Put a description for the operation here>>")
public String listOps (@Param("Parameter name") int a,
                           @Param("Parameter name ") int b,
                           @Param("Parameter name") String c);

The file should be named <ServiceName>MBean.java. The interface for the Sample Traffic Path provided with the Extension Toolkit is named MBean.java. Note the @Attr, @Description, and the @Parm annotations. These strings define what is shown in the UI of the Network Gatekeeper Console extension to describe, respectively, the attributes, operations, and parameter names.

Implement the Interface

Once you have defined the interface, it must be implemented. See Listing 9-2

Listing 9-2 A Snippet of a Sample Implementation

```java
/**
 * Bean implementation
 **/

class TestMBeanImpl extends WLNGMBeanDelegate implements TestMBean {

    public TestMBeanImpl() throws NotCompliantMBeanException {
        super(TestMBean.class);
    }

    public void setRepeatIndicator(int value){
        //set attribute in the config store
```
Making Traffic Paths Manageable

/**
 * Optional, only if you want to send notification to a listener on
 * attribute change
 **/
 super.sendNotification(new AttributeChangeNotification(this,
                  sequenceNumber, timeStamp, "msg", "attributeName",
                  "attributeType", oldValue, newValue))
}

/**
 * Get the repeat indicator value used by Parlay.
 * @return Repeat indicator value.
 */

public int getRepeatIndicator(){
    // get the attribute from the config store
}

/**
 * Specifies if a response is required from the call
 * user interaction service.
 * @param value The requested flag
 */

public void setResponseRequested(int value){
    // set the value in config store
}
/**
 * Get the response requested flag.
 * @return The response requested flag
 */

public int getResponseRequested(){
    // Get the value from config store
}

/**
 * list the operation
 * @return result
 */

public String listOps(int a, int b, String c){
    // String result;
    //provide implementation
    return result;
}

Note that this class is named TestMBeanImpl. You must name your class
<ServiceName>MBeanImpl.java, based on the interface name. Notice also that it extends
WLNGMBeanDelegate. This class takes care of setting up notifications 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
Sample Traffic Path provided with the Extension Toolkit is named MBeanImpl.java.
Register the MBeans with the Runtime MBean Server

Once the MBean is written, it must be registered with the Runtime MBean Server in the local WLS instance.

Listing 9-3   Registering the MBean

```java
import com.bea.wlcp.wlng.api.management.MBeanManager;
...

MBeanManager.registerMBean(mBeanImpl, ObjectType, displayName);
```

where `mBeanImpl` is the instance of the `BeanImplementation` class (TestMBeanImpl)

- `mBeanInterface` is the interface class (TestMBean.class)
- `ObjectType` is a string (name of the MBean) `com.bea.wlcp.wlng.TestMBean`
- `displayName` is a String (the name of the service)

The convention is that this class is named `<ServiceName>Management.java`. In the Example Traffic Path code this class is merely named `Management.java`. To unregister the MBean:

Listing 9-4   Unregistering the MBean

```java
import com.bea.wlcp.wlng.api.management.MBeanManager;
...

MBeanManager.unregister(ObjectType, displayName);
```

- `ObjectType` is a string (the fully qualified name of the MBean) `com.bea.wlcp.wlng.TestMBean`
Use the Configuration Store to Persist Values

The Network 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 Sample Traffic Path provides a handler class, ConfigurationStoreHandler.java, that gives an example of both usages of the Configuration Store API.

Note: The configuration store supports only Boolean, Integer, Long, and String values.
Making Traffic Paths Manageable
Writing test cases is a key part of developing new traffic paths. The WebLogic Network Gatekeeper Extension Toolkit provides substantial support for setting up test clients and simple test cases for checking both application and network initiated traffic and EDR/CDR generation.

Overview

Figure 10-1 provides an overview of the testing tools.
Testing Your Traffic Path

Figure 10-1 The Extension Toolkit Testing Tools

The Network Gatekeeper Extension Toolkit Testing Tools provides the following:

- Build support for automatically generating web service client stubs and callback web services (1)
- Abstract test cases for application and network initiated tests and duration tests (2)
- Component libraries for client-side session management (3), WS-Policy support, SOAP handlers, and general utilities (4)
- Support for initializing test cases through properties files (5)
- Outlines for Test Agents: gateway objects that the test client can query to retrieve “network-originated” traffic (6)
- Utilities for testing EDR/CDR generation
How It Works

There are two basic testing patterns supported by the test tools:

- Application-Initiated (MT)
- Network-Initiated (MO)

**Application-Initiated (MT)**

In this pattern, the Test Web Client:

1. Logs into Network Gatekeeper, using the Session Manager for 3.0 style traffic paths, or the Access Web Service for backwards compatible traffic paths.

2. Sends a Request to Network Gatekeeper, including all the necessary security and session headers

**Note:** The Network Simulator component is not supplied by the Testing Tools. It must be supplied by the implementer. The sample test case provided with the testing tools distribution uses an EJB (the Inbound EJB in the diagram below) as a sink for dumping the messages.
Network-Initiated (MO)

In this pattern, the Test Client:

1. Sends the “network-initiated” information to the Simulator (Outbound EJB)

   **Note:** The Simulator component is not supplied by the Testing Tools. It must be supplied by the implementer. The sample test case provided with the testing tools distribution uses an EJB (the Outbound EJB in the diagram below) to accept calls from the testing client and forward them to the traffic path.

2. Checks the Listener Agent to see what value was delivered to the callback Web Service implementation and compares that value with the value it sent.
Setting up the Testing Tools

The core idea of the testing tools is the test case. For each application-facing interface your traffic path supports and you wish to test, you must extend the abstract class com.bea.wlp.wlng.clients.tests.WlngTestCase.

This class controls the actual running of the test, including loading property files and establishing sessions. It can also check EDR/CDR generation and, in the case of Network-Initiated traffic, compare and match input and output.

By default, the testcases directory should have the following files and sub-directories:

- build.xml: The Ant build file. The sample test case includes a quite detailed example, which you can use as the basis of your own file if you wish.
Testing Your Traffic Path

- **ClientHandlerConfig.xml**: The client side Web Services handler configuration file. These handlers manage configuring SOAP headers with session IDs, etc.

- **Directory properties**: Holds the properties file, which the test case uses to initialize the test

- **Directory resource**: Holds the configuration files for the callback web service ear file

- **Directory src**: Holds the test case implementation source code

- **Directory wsdls**: Holds the WSDL documents that serve as the input for generating Web Services client stubs, as well as callback web services as necessary

The **testingtools** subdirectory, found in `<Extension_Toolkit>\testingtools\` in the default installation, contains three sample test cases, all of which can be built and run with the sample traffic path. Full examples of all of the following files and samples can be found in this directory.

By default, the **testcases** directory has the following directory structure:

**Listing 10-1  The Testcases Directory Structure**

testcases
  |--properties
    | |--common.properties
    | |--example
    |     |--tests.properties
  |--resource
    | |--ear
    |     |--META-INF
    |         |--application.xml
    |         |--weblogic-application.xml
  |--src
  |--wsdls
    | |--callback
There are seven basic tasks in getting ready to run a test case:

- Prepare to generate the Web Services client stub
- Prepare to generate the callback Web Service, if necessary
- Implement the TestAgentResult object
- Deploy the callback Web Services and the Listener Test Agent to a separate ear file
- Create the test case
- Implement the notification matching functionality in the test case
- Implement the EDR matching functionality in the test case
- Edit the properties files, as necessary

**Prepare to generate the Web Services client stub**

The `generate-client` target that is part of the `build.xml` file automatically generates the Web Services Client stub from the WSDL files that define the interfaces. The only thing that you need to do is to locate the WSDLs somewhere the build can access them - in the default directory structure (see Listing 10-1), they are stored in the `wsdls/service` directory. You must then edit the `build.xml` file to indicate where the WSDLs are by adding properties. For example:

```xml
<property name="example_data_send_wsdl" value="${wsdl.service}/example_data_send_service.wsdl"/>
```

You then must add the interface name to the `client_list` property to match the entry:

```xml
<property name="client_list" value="example_data_send_wsdl"/>
```

In addition, you will probably want to use one of the provided client-side handlers, both of which automatically manage inserting the session ID into the client SOAP header, and one of which, `ClientAttachmentHandler`, also manages SOAP attachments. You do this by editing the `ClientHandlerConfig.xml` file to indicate the handler you wish to use. For example:
Testing Your Traffic Path

Listing 10-2  Sample ClientHandlerConfig.xml

```
<weblogic-wsee-clientHandlerChain
xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:j2ee="http://java.sun.com/xml/ns/j2ee">
  <handler>
    <j2ee:handler-name>sessionidhandler</j2ee:handler-name>
    <j2ee:handler-class>com.bea.wlcp.wlng.client.handlers.ClientAttachmentHandler</j2ee:handler-class>
  </handler>
</weblogic-wsee-clientHandlerChain>
```

You need to do this for every interface you wish to test.

Prepare to generate the callback Web Service, if necessary

If your interface supports Network-Initiated traffic, you must also edit the `build.xml` file so that Ant will generate the appropriate Web Services to receive your callbacks, using the `generate-service` target. As in the case of the Web Services client, first you must locate the WSDLs where Ant can find them, and create a `wsdl` property in the `build.xml` file to tell it where to look. For example:

```
<property name="example_notif_wsdl" value="${wsdl.callback}/example_notification_service.wsdl"/>
```

You must also edit the `callback_list` property just as you did the `client_list` property:

```
<property name="callback_list" value="example_notif_wsdl"/>
```

You need to do this for every type of network-initiated traffic you wish to test.
Each interface produces a JWS interface file and an implementation file, which must be transferred to the `src` directory.

**Implement the TestAgentResult object**

In order to make the results that reach the callback web service available to the test case, they must be added to the Listener Test Agent, which is a listener is registered in the JNDI tree that is stored in the callback ear. You must deliver this information using a result class that extends `com.bea.wlcp.wlng.callback.agent.TestAgentResult`. For example:

```
public class NotificationResult implements TestAgentResult {

    static final long serialVersionUID = 25436448921L;

    private String correlator;
    private String address;
    private String data;

    # constructor
    ...

    # setters and getters
    ...
}
```

You also need to add a bit of code to the Web Service implementation to add the result object to the TestAgent. For example:
Listing 10-4  Modify callback web service implementation to add data to Test Agent

```java
public void notifyDataReception(java.lang.String correlator,
                                  java.net.URI originatingAddress, java.lang.String data)
{
    String str = "notifyDataReception] correlator: " + correlator + "\n"
    + "originatingAddress: " + originatingAddress + "\n" + "data: " + data
    + "\n";
    System.out.println(str);

    TestAgentApp.getAgent()
        .addResult(
            new NotificationResult(correlator, originatingAddress.toString(),
                                      data));
    return;
}
```

**Deploy the callback Web Services and the Listener Test Agent to a separate ear file**

Wrap your callback Web Services and the Test Agent into an ear file. You will need to create the appropriate `application.xml` and `weblogic-application.xml` configuration files and store them in `resource/ear/META-INF/`, as shown in Listing 10-1 above. Brief examples follow:

Listing 10-5  Sample application.xml

```xml
<?xml version='1.0' encoding='UTF-8'?>
<application xmlns="http://java.sun.com/xml/ns/j2ee"
Once these files have been created, deploy your callback service ear using the deploy-service target in build.xml. You will need to modify it for your purposes.
Create the test case

Your test case must extend the abstract test case `com.bea.wlcp.wlng.clients.tests.WlngTestCase`. The most important method you need to implement is the `setUp` method, which you must override. This is the method that instantiates the service and the port.

**Listing 10-7  Test case snippet: setUp**

```java
protected void setUp() throws Exception {
    super.setUp();

    // Instantiate the service and port
    SendDataService service = new SendDataService_Impl(getProperty("url") + "/SendData?WSDL");
    port = service.getSendData();

    // This step is important: the underlying security will be prepared afterwards.
    setStub((Stub)port);
}
```

Notice that URL to be used for reaching the Web Service is input as a property:

`SendDataService_Impl(getProperty("url"))`

For more information on using properties files to hold changing configuration data that can be loaded at runtime, see Edit the properties files, as necessary below.

Also notice the `setStub` method. This automatically sets up the mechanism for inserting the appropriate WS-Policy information into the client stub request SOAP headers based, again, on configuration information in the properties files.
Implement the notification matching functionality in the test case

If your test case is testing network-initiated traffic, you must add code to your test case to fetch the result parameters that were delivered to the callback Web Service and then forwarded to the Test Agent (see Implement the TestAgentResult object). The abstract test case contains helper objects to help you find the Test Agent object. You can also add code to check whether the results are what you expected. See Listing 10-8 below:

Listing 10-8   Fetching data from TestAgent

```java
TestAgent agent = lookupAgent();
agent.clear();
List<TestAgentResult> results = agent.getResults();
assertEquals(false, results.isEmpty());
NotificationResult result = (NotificationResult) results.get(0);
assertEquals("address", result.getAddress(), getProperty("mo.originatingAddress"));
assertEquals("data", result.getData(), getProperty("mo.data"));
```

Implement the EDR matching functionality in the test case

Just as it is possible to use the test case to automatically fetch and compare result data from network-initiated traffic, it is also possible to have the test case to automatically fetch and compare EDR data.

**Note:** General events, charging events, and alarm events in Network Gatekeeper are all initially generated as EDRs, and are only separated by filtering mechanisms. So this EDR matching functionality can be used to test all three sorts of events. For more information on EDRs, see the “Introducing Traffic Paths” chapter in the Architectural Overview, a document in the Network Gatekeeper documentation set.

The abstract `WlngTestCase` contains methods both for fetching and for comparing selected EDR data (it includes its own External EDR listener) You simply need to add code to set up the
expected EDR values to which the received EDRs should be matched. You need to specify only the EDR values in which you are interested - other EDRs that are generated are simply ignored by the matching functionality.

**Listing 10-9  EDR matching snippet**

```java
List<EdrData> actualEdrs = getEdrs(Integer.parseInt(getProperty("sleeptime")));

assertTrue("Edrs received", actualEdrs.size() > 0);

EdrData expectedEdr = new EdrData();
expectedEdr.setValue(EdrConstants.FIELD_METHOD_NAME,
                      "SendDataResponse sendData(SendData)"可谓);
expectedEdr.setValue(EdrConstants.FIELD_CLASS_NAME,
                      "com.acompany.plugin.example.netex.send_data.north.SendDataPluginImpl");
expectedEdr.setValue(EdrConstants.FIELD_INTERFACE,
                      EdrConstants.VALUE_INTERFACE_NORTH);
expectedEdr.setValue(EdrConstants.FIELD_DIRECTION,
                      EdrConstants.VALUE_DIRECTION_SOUTH);
expectedEdr.setValue(EdrConstants.FIELD_POSITION,
                      EdrConstants.VALUE_POSITION_BEFORE);

assertTrue("EDRs matching", matchEdr(expectedEdr, actualEdrs));
```

In this snippet, the EDRs that are being selected for comparison are those that have the following structure:
These particular constants represent information that is collected for almost all EDRs. In creating your own traffic path, you also have the opportunity to create your own additional information types, and these can also be used to filter the EDRs that you wish to match.

**Edit the properties files, as necessary**

In order to provide simple and flexible runtime configuration of the test cases, many values that need to change from one run to the next are stored in properties files. There are two such properties files, `common.properties` and `tests.properties`. (See Listing 10-1 for where these files live.) The first file, `common.properties`, typically contains basic information which is not likely to change much during the run of your tests. The properties in Listing 10-2 should always be included. Others can be added as you like.

### Table 10-1  EDR to be matched

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD_METHOD_NAME</td>
<td>&quot;SendDataResponse sendData(SendData)&quot;</td>
</tr>
<tr>
<td>FIELD_CLASS_NAME</td>
<td>&quot;com.acompany.plugin.example.netex.send_data.north.SendDataPluginImpl&quot;</td>
</tr>
<tr>
<td>FIELD_INTERFACE</td>
<td>VALUE_INTERFACE_NORTH</td>
</tr>
<tr>
<td>FIELD_DIRECTION</td>
<td>VALUE_DIRECTION_SOUTH</td>
</tr>
<tr>
<td>FIELD_POSITION</td>
<td>VALUE_POSITION_BEFORE</td>
</tr>
</tbody>
</table>

These particular constants represent information that is collected for almost all EDRs. In creating your own traffic path, you also have the opportunity to create your own additional information types, and these can also be used to filter the EDRs that you wish to match.

### Table 10-2  common.properties contents

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The basic URL for accessing the Web Service you are testing</td>
</tr>
<tr>
<td>wspolicy</td>
<td>WS Policy mechanism: UsernameToken is currently supported</td>
</tr>
<tr>
<td>sessionman_url</td>
<td>The URL of the session manager Web Service</td>
</tr>
</tbody>
</table>
The file itself takes the form of a simple name=value property list. So the basic file would look something like this:

**Listing 10-10  common.properties from the sample test case**

```plaintext
# Basic URL for reaching Web services
url=http://hostname:8001/example

# The URL for getting sessionId
sessionman_url=http://hostname:8001/parlayx2/session_manager/SessionManager

# Web services security mechanism
wspolicy=UsernameToken

# Username, password
username=SP1APP1APPInstGroup1
password=SP1APP1APPInstGroup1
```

This file is loaded automatically by the test case on startup. Other values, usually ones that are specific to a particular run of a particular test, are stored in tests.properties. You must add

---

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>The login username assigned during provisioning of the test client in Network Gatekeeper</td>
</tr>
<tr>
<td>password</td>
<td>The login password assigned during provisioning of the test client in Network Gatekeeper</td>
</tr>
</tbody>
</table>
a snippet to your test case to load these LOCAL_PROPERTIES. See Figure 10-11 for a sample of how this can be done.

### Listing 10-11 Snippet from sample MO test case loading tests.properties

```java
public class TestMO extends WlngTestCase {

    private static final String LOCAL_PROPERTIES = 
        "properties/example/tests.properties";

    private NotificationManager port = null;

    public TestMO(String name) throws Exception {
        super(name);
        loadProperties(LOCAL_PROPERTIES);
    }
}
```

Again, the file should be made up of a list of `name=value` properties, as needed, based on your test case. See Listing 10-12, which shows the `tests.properties` from the sample test case.

### Listing 10-12 test.properties from the sample test case

```
# Local configuration for example TP testcase

mt.address=tel:1234
mt.data=MT data for testing example MT
mo.data=MO data for testing example MO
mo.correlator=cor888
```
Testing Your Traffic Path

```
mo.endpoint=http://localhost:8001/NotificationService/services/Notification
mo.originatingAddress=originatingAddress
mo.nwTransactionId=92938
mo.destinationAddress=tel:1234

edr.url=t3://localhost:8001/
edr.verbose=false
service_url=t3://localhost:8001
jndiname=org.netex.sim.Outbound
sleeptime=15000
```

Typically, these values are set in the test case using code like this:

**Listing 10-13  Snippet setting parameters in a test case from tests.properties**

```
parameters.setAddress(new java.net.URI(getProperty("mo.destinationAddress")));
parameters.setCorrelator(getProperty("mo.correlator"));
parameters.setEndPoint(getProperty("mo.endpoint"));
```

Overview of Testing Tools Component Packages

The following packages are available as part of the testing tools files:

- **com.bea.wlcp.wlng.callback.agent**
  
  This package contains the interfaces and classes for implementing the Test Agent functionality

- **com.bea.wlcp.wlng.client.access** and **com.bea.wlcp.wlng.client.access.impl**
Overview of Testing Tools Component Packages

These packages contain the interfaces and the implementations for establishing sessions and session management. Although the packages contain mechanisms to support both 2.2 and 3.0 style logins, the assumption is that all new extension traffic paths will be developed using the 3.0 style exclusively.

- **com.bea.wlcp.wlng.client.access.wspolicy**
  This package contains the interface and one implementation for the mechanisms to support WS Policy. The single implementation supports UsernameToken but classes that support, for example, X.509 could also be developed.

- **com.bea.wlcp.wlng.client.edr**
  This package contains classes that support the EDR fetching and matching functionality.

- **com.bea.wlcp.wlng.client.handlers**
  This package contains two handler classes for setting up request SOAP headers.

- **com.bea.wlcp.wlng.client.tests**
  This package contains the abstract test cases that your test cases will extend.

- **com.bea.wlcp.wlng.client.util**
  This package contains a variety of helper classes.

For more detailed information on these packages, please see the Javadoc, in the `<bea_home><et_wlng_home>\testingtools\doc\javadoc` directory.
Adapting 2.2 Style Traffic Paths

If you created a Network Gatekeeper extension based on the 2.2 architecture, you will need to make some adjustments to your code so that it will continue to run in the new 3.0 environment. The following chapter provides an overview of the steps you should take. There are essentially three scenarios:

- **Adapting a 2.2 Style Plug-in for a 3.0 Backwards Compatible Traffic Path**
- **Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface**
- **Adapting a 2.2 Style Traffic Path Based on a non-3.0 Supported, Web Services-based, Application Facing Interface**

**Note:** The sending of requests to multiple plug-ins, as, for example, in the case of sendlists, is not supported in backwards compatible traffic paths. You must implement a 3.0 style traffic path to support this feature.

### Adapting a 2.2 Style Plug-in for a 3.0 Backwards Compatible Traffic Path

Some of the Network Gatekeeper 3.0 traffic paths are, in essence, adapted 2.2 style traffic paths. These include Third Party Call, a version of Short Messaging, Multimedia Messaging, Terminal Location, and Payment, all of whose application facing interfaces are based on the Parlay X 2.1 standard, and WAP Push, based on Network Gatekeeper’s own Extended Web Services interface. If your 2.2 based extension supports one of these interfaces, the only thing you need to do is to include your 2.2 style plugin jar file into the `wlng_nt.ear`. 
1. Unzip the `wlng_nt.ear` into a temporary directory

2. Copy your plugin jar file into the `wlng_nt` root directory

3. If the plug-in is for Short Messaging: Copy the file $DOMAIN_HOME/bc/y_sms_retained.jar to the `wlng_nt` root directory.

4. In the `wlng_nt` directory, modify the `META-INF/weblogic-extension.xml` file. Add the following stanza:

   **Listing 11-1 Add to weblogic-extension.xml**

   ```xml
   <custom-module>
   <uri> JAR-NAME </uri>
   <provider-name>SleeModule</provider-name>
   </custom-module>
   ```

5. If the plug-in is for Short Messaging:
   a. In the `wlng_nt` directory, modify the `META-INF/weblogic-extension.xml` file. Add the following before the plug-in definition is added:

   **Listing 11-2 Add to weblogic-extension.xml if adding an SMS plug-in**

   ```xml
   <custom-module>
   <uri>y_sms_retained.jar</uri>
   <provider-name>SleeModule</provider-name>
   </custom-module>
   ```

   b. Remove the SMPP plug-in for Short Messaging, since the two plug-ins cannot be installed in parallel. In `weblogic-extension.xml`, find the following entry and remove it:
Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface

Listing 11-3  remove from weblogic-extension.xml if adding an SMS plug-in

<custom-module>
  <uri>j_sms_smpp.jar</uri>
  <provider-name>SleeModule</provider-name>
</custom-module>

6. Zip the temporary directory back up into wlng_nt.ear.
7. Copy the ear file back into your wlng domain
8. Redeploy.

Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface

If your 2.2 style extension traffic path is based on an application-facing interface that is supported by Network Gatekeeper 3.0, but only in an Enhanced traffic path version, you will need to use the Eclipse plug-in supplied with the Extension Toolkit to generate a 3.0 style plug-in skeleton. For information on using the Eclipse plug-in, see Creating a plug-in project for an existing Network Gatekeeper application-facing interface in Chapter 3, “Using the Eclipse Plug-in for Extension Toolkit.” All of your 2.2 style code, including your 2.2 style Web Services layer, will execute within the context of that plug-in, in the Network Tier. The entirety of the Access Tier application-facing Parlay X 2.1 Web Services implementations that you will use are provided by the existing Enhanced traffic path to interact with service provider applications. For an overview of the code migration, see Figure 11-1 below.
Merge the 2.2 Web Services implementation

The first thing you must do is transfer your old Web Services implementation code into corresponding new skeletons that the Eclipse plugin generates for you. There will be one empty plug-in impl class for each interface from the Parlay X 2.1 interface set, placed in the appropriate package. So, for example, if you are transferring 2.2 style Terminal Status code, Eclipse generates two skeleton files: `TerminalStatusPluginImpl` and `TerminalStatusNotificationManagerPluginImpl`, into which you should copy the corresponding code from your 2.2 Web Services implementation. See the SMS-based example below, in Figure 11-2:
Because the generated code is created by the WLS web service implementation and not Axis, as was the case in 2.2, you will probably need to make some adjustments, particularly in argument and exception types. If you are implementing a Listener interface, you no longer need to spawn a new thread for each incoming request, which was considered a Best Practice in the 2.2 architecture.

**Implement the Managed Plugin class**

Once you have created these new implementation files, you must complete the implementation of the skeleton Managed Plugin class that the Eclipse plug-in generates for you. This file makes your plug-in code deployable as a SLEE service, with all the appropriate lifecycle methods. It also makes it possible for the entire "outer" plug-in to be registered with the Plug-in Manager. (The actual registration must be setup at runtime using OAM procedures.) There is a complete example version of the Managed Plugin class in Chapter 4, “Description of a Generated Project.”

**Note:** There are, in fact, two calls to Plug-in Manager in this model. The first is the one facilitated by the Managed Plugin class, which uses a Java call and the 3.0 style mechanisms for selecting appropriate 3.0 style plug-ins. The second is the call to the Plug-in Manager from the legacy plug-in module from your 2.2 implementation. This call is a CORBA call, and behaves just as it did in the 2.2 architecture.
Adapting 2.2 Style Traffic Paths

Any helper/handler, etc. code you have created should also be transferred here and adjusted to the new WLS based framework. In addition to making any adjustments in type, etc., you will also need to change how you get the Common Loader so that you can get a handle to your SESPA modules. Listing 11-4 below provides an example based on a converted SMS traffic path to accomplish this.

**Listing 11-4  Sample new getRefToSESPA method**

```java
public static Messaging_2 getRefToSespa() throws org.csapi.schema.parlayx.common.v2_1.ServiceException {
    SleeCommonLoader scl = SleeCommonLoader.getInstance();
    Messaging_2 sespaMess = null;
    try {
        sespaMess = (Messaging_2) scl.getObject("SESPA_messaging.Messaging_2");
    } catch (SleeCommonLoaderException commonLoaderEx) {
        throw new String[] { commonLoaderEx.getMessage() });
    }
    return sespaMess;
}
```

Login ticket handling has changed slightly from version 2.2. In version 3.0, login tickets (or Session IDs, as they are called in enhanced traffic paths) are removed from the application-initiated SOAP header in the Access Tier using `com.bea.wlcp.wlng.soap.SessionIdHandler`. This class stores the login ticket in a WorkContextMap, from which it can be fetched in the Network Tier. The ManagedPlugin class inherits a method, `getLoginTicket`, from `AbstractManagedPlugin` (see Figure 11-2 above) that can retrieve the login ticket from the Work Context.
In addition, if your traffic path implements listeners tied to its lifecycle, you may want to used the ManagedPlugin class to load your listeners into the SLEE Common Loader, as in the MMS-based example shown in Listing 11-5 below.

**Listing 11-5  Sample add listeners**

```java
/* (non-Javadoc)
 * @see com.bea.wlcp.wlmg.api.plugin.common.AbstractManagedPlugin#doStarted()
 */
@Override
public void doStarted() throws ServiceDeploymentException {
    SleeCommonLoader loader = SleeCommonLoader.getInstance();
    try {
        loader.addObject(OBJ_MMS_STATUS_NOTIFICATION_LISTENER,
                        new MmsStatusNotificationReqListener_impl());
    } catch (SleeCommonLoaderException e) {
        if(logger.isEnabledFor(Level.ERROR)){
            logger.error("SleeCommonLoaderException", e);
        }
    }

    try {
        loader.addObject(OBJ_MMS_NOTIFICATION_LISTENER,
                        new MmsNotification_impl());
    } catch (SleeCommonLoaderException e) {
        if(logger.isEnabledFor(Level.ERROR)){
            logger.error("SleeCommonLoaderException", e);
        }
    }
}
```
The rest of your code can be copied as is into the plug-in. OAM interfaces remain exactly as they were in 2.2.

**Adapting a 2.2 Style Traffic Path Based on a non-3.0 Supported, Web Services-based, Application Facing Interface**

Finally, if your 2.2 style extension traffic path is *not* based on an application-facing interface that is supported by Network Gatekeeper 3.0 at all, but *is* Web Services based, you can use the Network Gatekeeper Extension Toolkit to generate an entire traffic path project. See Chapter 3, “Using the Eclipse Plug-in for Extension Toolkit” for more information on this process. Also see Chapter 5, “The Traffic Path Example.”

**Note:** Only Web Services based traffic paths are supported by the Extension Toolkit.

The Toolkit generates the entire Access Tier based on the WSDL that you supply. Then you import your code into the Network Tier, as detailed in Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface with one additional step: you must implement one Request Factory class for each of your `PluginImpl` classes. Skeletons of these classes are generated by the Toolkit, and are in the Service Capability SLSB with filenames equal to `<InterfaceName>PluginFactory`. So in the SMS based example shown in Adapting a 2.2 Style Traffic Path Based on a 3.0 Supported Application Facing Interface above, the Request Factory classes would be `MessageNotificationManagerPluginFactory`, `ReceiveMessagePluginFactory`, and `SendMessagePluginFactory`.

These classes extend `RequestFactory`, and provide two key pieces of functionality. First, they define a method `createRequestInfo`, which extracts routing information from the incoming request and makes it available to the `PluginManager`.

See Listing 11-6 below for an example `createRequestInfo` snippet.
public RequestInfo createRequestInfo(Class<? extends Plugin> type,
    Method method, Object... args) {

    String methodName = method.getName();

    if (methodName.equals(START_EVENT_NOTIFICATION_STRING)) {

        Object arg1 = getArgument(0, args);

        if (arg1 instanceof StartEventNotification) {

            StartEventNotification startEventNotification =
                (StartEventNotification) arg1;

            URI destinationAddress = startEventNotification.getAddress();

            return new AddressRequestInfo(type, method, destinationAddress,
                args);

        }

    } else if (methodName.equals(STOP_EVENT_NOTIFICATION_STRING)) {

        Object arg1 = getArgument(0, args);

        if (arg1 instanceof StopEventNotification) {

        }
Adapting 2.2 Style Traffic Paths

StopEventNotification stopEventNotification =
        (StopEventNotification) arg1;

String correlator = stopEventNotification.getCorrelator();

return new CorrelatorRequestInfo(type, method, correlator, args);
}
}

throw new AssertionError("Failed to process request");
}

Second, these classes convert plug-in exceptions to types that can be returned to the application: See Listing 11-7 below.

Listing 11-7  Convert exception types snippet

@Override

public Throwable convertEx(Method method, Throwable e) {
    if (logger.isInfoEnabled()) {
        logger.info("Internal exception", e);
    }

    if (e instanceof ServiceException) {
        return e;
    }

    if (e instanceof PluginException) {
private Throwable convertEx(PluginException e) {

    if (e instanceof DenyPluginException) {
        DenyPluginException dpe = (DenyPluginException) e;
        DenyException de = (DenyException) dpe.getCause();
        if (de != null) {
            return ExceptionType.POLICY_ERROR.createEx(
                    de.denyCode + e.getMessage());
        }
        return com.acompany.example.plugin.ExceptionType.POLICY_ERROR.createEx(
                e.getMessage());
    } else if (e instanceof NoAvailablePluginException) {
        return ExceptionType.SERVICE_ERROR.createEx(e.getMessage());
    }

    return ExceptionType.SERVICE_ERROR.createEx(e.getMessage());
}
Adapting 2.2 Style Traffic Paths

**Note:** The sample version of this functionality available in the version of the Sample Traffic Path implements the Request Factory interface in multiple files rather than in a single class file. The Extension Toolkit generated file assumes that both types of functionality will exist in the same file.
SLA Parameters and Policy enforcement for new plug-ins

If creating an enhanced plug-in for a traffic path that is delivered as an backwards compatible traffic path with the Network Gatekeeper, namely:

- MultiMedia Messaging
- Payment
- Terminal Location

another set, compared with the backwards compatible traffic paths, of SLA tags and SLA parameters are applicable. Also the policy rules can use another set of parameters. This section describes the these settings.

In enhanced (3.0 type) traffic paths, the policy rules operate on methods identified by the Java representation of the operation on the application-facing interface. 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:

\texttt{arg<n>.<parameter name>}

where `<n>` in \texttt{arg<n>} depends on the WSDL that defines the application-facing interface, normally this is \texttt{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.

Parlay X 2.1 Part 5: MultiMedia Messaging

Interface: SendMessage

Value in <scs> is com.bea.wlcp.wlng.px21.plugin.SendMessagePlugin.

sendMessage

Value in <methodName> and <methodNameGuarantee> is sendMessage.

Table 12-1 Interface: SendMessage, operation sendMessage

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>addresses</td>
<td>arg0.addresses</td>
<td>java.net.URI[]</td>
</tr>
<tr>
<td>senderAddress</td>
<td>arg0.senderAddress</td>
<td>String</td>
</tr>
<tr>
<td>subject</td>
<td>arg0.subject</td>
<td>String</td>
</tr>
<tr>
<td>priority</td>
<td>arg0.priority</td>
<td>String enumeration</td>
</tr>
<tr>
<td>charging</td>
<td>arg0.charging.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charging.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charging.description</td>
<td>String</td>
</tr>
<tr>
<td>receiptRequest</td>
<td>receiptRequest.correlator</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>receiptRequest.endpoint</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>receiptRequest.interfaceName</td>
<td>String</td>
</tr>
</tbody>
</table>

getMessageDeliveryStatus

Value in <methodName> and <methodNameGuarantee> is getMessageDeliveryStatus.
Interface: ReceiveMessage


**getMessage**

Value in <methodName> and <methodNameGuarantee> is getMessage.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestIdentifier</td>
<td>arg0.requestIdentifier</td>
<td>String</td>
</tr>
</tbody>
</table>

Table 12-3 Interface: ReceiveMessage, operation getReceivedMessages

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageRefIdentifier</td>
<td>arg0.messageRefIdentifier</td>
<td>String</td>
</tr>
</tbody>
</table>

**getReceivedMessages**

Value in <methodName> and <methodNameGuarantee> is getReceivedMessages.
SLA Parameters and Policy enforcement for new plug-ins

getMessageURIs
Value in <methodName> and <methodNameGuarantee> is getMessageURIs.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>registrationIdentifier</td>
<td>arg0.registrationIdentifier</td>
<td>String</td>
</tr>
<tr>
<td>priority</td>
<td>arg0.priority</td>
<td>String enumeration</td>
</tr>
</tbody>
</table>

getMessgeURIs
Value in <methodName> and <methodNameGuarantee> is getMessageURIs.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageRefIdentifier</td>
<td>arg0.messageRefIdentifier</td>
<td>String</td>
</tr>
</tbody>
</table>

Interface: MessageNotificationManager
Value in <scs> is com.bea.wlcp.wlng.px21.plugin.MessageNotificationManagerPlugin.

startMessageNotification
Value in <methodName> and <methodNameGuarantee> is startMessageNotification.
stopMessageNotification
Value in <methodName> and <methodNameGuarantee> is stopMessageNotification.

Table 12-6  Interface: MessageNotificationManager, operation startMessageNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageServiceActivationNumber</td>
<td>arg0.messageServiceActivationNumber</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>reference</td>
<td>arg0.reference.correlator</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.endpoint</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.interfaceName</td>
<td>String</td>
</tr>
<tr>
<td>criteria</td>
<td>arg0.criteria</td>
<td>String</td>
</tr>
</tbody>
</table>

Table 12-7  Interface: MessageNotificationManager, operation stopMessageNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg0.correlator</td>
<td>String</td>
</tr>
</tbody>
</table>

Interface: MessageNotification
SLA Parameters and Policy enforcement for new plug-ins

### notifyMessageReception

Table 12-8  Interface: MessageNotification, operation notifyMessageReception

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg1.correlator</td>
<td>String</td>
</tr>
<tr>
<td>message</td>
<td>arg1.message.messageIdentifier</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.message.messageServiceActivationNumber</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.message.senderAddress</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>arg1.message.subject</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.message.priority</td>
<td>String enumeration</td>
</tr>
<tr>
<td></td>
<td>arg1.message.message</td>
<td>String</td>
</tr>
</tbody>
</table>

In addition, arg0 (String) contains the URL to the application endpoint.

### notifyMessageDeliveryReceipt

Table 12-9  Interface: MessageNotification, operation notifyMessageDeliveryReceipt

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg1.correlator</td>
<td>String</td>
</tr>
<tr>
<td>deliveryStatus</td>
<td>arg1.deliveryStatus.address</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>arg1.deliveryStatus.deliveryStatus</td>
<td>String enumeration</td>
</tr>
</tbody>
</table>

### Parlay X 2.1 Part 6: Payment

**Interface: AmountCharging**

Value in <scs> is com.bea.wlcp.wlng.px21.plugin.AmountChargingPlugin.
chargeAmount
Value in `<methodName>` and `<methodNameGuarantee>` is chargeAmount.

Table 12-10  Interface: AmountCharging, operation chargeAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>charge</td>
<td>arg0.charge.description</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.code</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
</tbody>
</table>

refundAmount
Value in `<methodName>` and `<methodNameGuarantee>` is refundAmount.

Table 12-11  Interface: AmountCharging, operation refundAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
</tbody>
</table>
Table 12-11 Interface: AmountCharging, operation refundAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>charge</td>
<td>arg0.charge.description</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.code</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
</tbody>
</table>

**Interface: VolumeCharging**

Value in `<scs>` is `com.bea.wlep.wlng.px21.plugin.VolumeChargingPlugin`.

**chargeVolume**

Value in `<methodName>` and `<methodNameGuarantee>` is `chargeVolume`.

Table 12-12 Interface: VolumeCharging, operation chargeVolume

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>billingText</td>
<td>arg0.billingText</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
<tr>
<td>property[]</td>
<td>arg0.property[].name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.property[].value</td>
<td>String</td>
</tr>
</tbody>
</table>
**getAmount**

Value in `<methodName>` and `<methodNameGuarantee>` is `getAmount`.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>property[]</td>
<td>arg0.property[].name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.property[].value</td>
<td>String</td>
</tr>
</tbody>
</table>

**refundVolume**

Value in `<methodName>` and `<methodNameGuarantee>` is `refundVolume`.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>billingText</td>
<td>arg0.billingText</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
<tr>
<td>parameters[]</td>
<td>arg0.parameters[].name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.parameters[].value</td>
<td>String</td>
</tr>
</tbody>
</table>
Interface: ReserveAmountCharging

Value in `<scs>` is com.bea.wlcp.wlng.px21.plugin.ReserveAmountChargingPlugin.

chargeVolume

Value in `<methodName>` and `<methodNameGuarantee>` is chargeReservation.

Table 12-15  Interface: ReserveAmountCharging, operation chargeReservation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservation Identifier</td>
<td>arg0.reservationIdentifier</td>
<td>String</td>
</tr>
<tr>
<td>charge</td>
<td>arg0.charge.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.code</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.description</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
</tbody>
</table>

releaseReservation

Value in `<methodName>` and `<methodNameGuarantee>` is releaseReservation.

Table 12-16  Interface: ReserveAmountCharging, operation releaseReservation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservation Identifier</td>
<td>arg0.reservationIdentifier</td>
<td>String</td>
</tr>
</tbody>
</table>
reserveAmount
Value in `<methodName>` and `<methodNameGuarantee>` is reserveAmount.

Table 12-17 Interface: ReserveAmountCharging, operation reserveAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentity</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>charge</td>
<td>arg0.charge.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.code</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.description</td>
<td>String</td>
</tr>
</tbody>
</table>

reserveAdditionalAmount
Value in `<methodName>` and `<methodNameGuarantee>` is reserveAdditionalAmount.

Table 12-18 Interface: ReserveAmountCharging, operation reserveAdditionalAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservationIdentifier</td>
<td>arg0.reservationIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>charge</td>
<td>arg0.charge.amount</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.code</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.currency</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.charge.description</td>
<td>String</td>
</tr>
</tbody>
</table>

Interface: ReserveVolumeCharging
Value in `<scs>` is com.bea.wlcp.wlng.px21.plugin.ReserveVolumeChargingPlugin.
chargeReservation
Value in `<methodName>` and `<methodNameGuarantee>` is chargeReservation.

Table 12-19  Interface: ReserveVolumeCharging, operation chargeReservation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservation Identifier</td>
<td>arg0.reservationIdentifier</td>
<td>String</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>billingText</td>
<td>arg0.billingText</td>
<td>String</td>
</tr>
<tr>
<td>referenceCode</td>
<td>arg0.referenceCode</td>
<td>String</td>
</tr>
</tbody>
</table>

getAmount
Value in `<methodName>` and `<methodNameGuarantee>` is getAmount.

Table 12-20  Interface: ReserveVolumeCharging, operation getAmount

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>endUserIdentifier</td>
<td>arg0.endUserIdentifier</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>parameters</td>
<td>arg0.parameters[].name</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.parameters[].value</td>
<td>String</td>
</tr>
</tbody>
</table>

releaseReservation
Value in `<methodName>` and `<methodNameGuarantee>` is releaseReservation.
reserveAdditionalVolume
Value in <methodName> and <methodNameGuarantee> is reserveAdditionalVolume.

Table 12-21  Interface: ReserveVolumeCharging, operation releaseReservation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservation Identifier</td>
<td>arg0.reservationIdentifier</td>
<td>String</td>
</tr>
</tbody>
</table>

Table 12-22  Interface: ReserveVolumeCharging, operation reserveAdditionalVolume

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reservation Identifier</td>
<td>arg0.reservationIdentifier</td>
<td>String</td>
</tr>
<tr>
<td>volume</td>
<td>arg0.volume</td>
<td>long</td>
</tr>
<tr>
<td>billingText</td>
<td>arg0.billingText</td>
<td>String</td>
</tr>
</tbody>
</table>

reserveVolume
Value in <methodName> and <methodNameGuarantee> is reserveVolume.
Parlay X 2.1 Part 9: Terminal Location

Interface: TerminalLocation

Value in <scs> is com.bea.wlcp.wlng.px21.plugin.TerminalLocationPlugin.

getLocation

Value in <methodName> and <methodNameGuarantee> is getLocation.

Table 12-24  Interface: TerminalLocation, operation getLocation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>arg0.address</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>requested Accuracy</td>
<td>arg0.requestedAccuracy</td>
<td>int</td>
</tr>
<tr>
<td>acceptable Accuracy</td>
<td>arg0.acceptableAccuracy</td>
<td>int</td>
</tr>
</tbody>
</table>
Table 12-24  Interface: TerminalLocation, operation getLocation

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum Age</td>
<td>arg0.maximumAge</td>
<td>TimeMetric._metric enumeration</td>
</tr>
<tr>
<td>responseTime</td>
<td>arg0.responseTime</td>
<td>TimeMetric._units int</td>
</tr>
<tr>
<td>tolerance</td>
<td>arg0.tolerance</td>
<td>String enumeration</td>
</tr>
</tbody>
</table>

getLocationForGroup

Value in <methodName> and <methodNameGuarantee> is getLocationForGroup.

Table 12-25  Interface: TerminalLocation, operation getLocationForGroup

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>addresses</td>
<td>arg0.addresses</td>
<td>java.net.URI[]</td>
</tr>
<tr>
<td>acceptable Accuracy</td>
<td>arg0.acceptableAccuracy</td>
<td>int</td>
</tr>
<tr>
<td>requested Accuracy</td>
<td>arg0.requestedAccuracy</td>
<td>int</td>
</tr>
<tr>
<td>maximum Age</td>
<td>arg0.maximumAge</td>
<td>TimeMetric._metric enumeration</td>
</tr>
<tr>
<td>responseTime</td>
<td>arg0.responseTime</td>
<td>TimeMetric._units int</td>
</tr>
<tr>
<td>tolerance</td>
<td>arg0.tolerance</td>
<td>String enumeration</td>
</tr>
</tbody>
</table>

getTerminalDistance

Value in <methodName> and <methodNameGuarantee> is getTerminalDistance.
Interface: TerminalLocationNotificationManager

Value in `<ses>` is com.bea.wlcp.wlng.px21.plugin.TerminalLocationNotificationManagerPlugin.

**endNotification**

Value in `<methodName>` and `<methodNameGuarantee>` is endNotification.

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>arg0.address</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>latitude</td>
<td>arg0.latitude</td>
<td>float</td>
</tr>
<tr>
<td>longitude</td>
<td>arg0.longitude</td>
<td>float</td>
</tr>
</tbody>
</table>

Interface: TerminalLocationNotificationManager, operation endNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg0.correlator</td>
<td>String</td>
</tr>
</tbody>
</table>

startGeographicalNotification

Value in `<methodName>` and `<methodNameGuarantee>` is startGeographicalNotification.
Table 12-28  Interface: TerminalLocationNotificationManager, operation startGeographicalNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>arg0.reference.correlator</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.endpoint</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.interfaceName</td>
<td>String</td>
</tr>
<tr>
<td>addresses</td>
<td>arg0.addresses</td>
<td>java.net.URI[]</td>
</tr>
<tr>
<td>latitude</td>
<td>arg0.latitude</td>
<td>float</td>
</tr>
<tr>
<td>longitude</td>
<td>arg0.longitude</td>
<td>float</td>
</tr>
<tr>
<td>radius</td>
<td>arg0.radius</td>
<td>float</td>
</tr>
<tr>
<td>trackingAccuracy</td>
<td>arg0.trackingAccuracy</td>
<td>float</td>
</tr>
<tr>
<td>criteria</td>
<td>arg0.criteria</td>
<td>String enumeration</td>
</tr>
<tr>
<td>checkImmediate</td>
<td>arg0.checkImmediate</td>
<td>boolean</td>
</tr>
<tr>
<td>frequency</td>
<td>arg0.frequency</td>
<td>TimeMetric</td>
</tr>
<tr>
<td>duration</td>
<td>arg0.duration</td>
<td>TimeMetric</td>
</tr>
<tr>
<td>count</td>
<td>arg0.count</td>
<td>Integer</td>
</tr>
</tbody>
</table>

**startPeriodicNotification**

Value in `<methodName>` and `<methodNameGuarantee>` is startPeriodicNotification.
SLA Parameters and Policy enforcement for new plug-ins

Table 12-29 Interface: TerminalLocationNotificationManager, operation startPeriodicNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in SLAs and policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>addresses</td>
<td>arg0.addresses</td>
<td>java.net.URI[]</td>
</tr>
<tr>
<td>duration</td>
<td>arg0.duration</td>
<td>String enumeration</td>
</tr>
<tr>
<td>frequency</td>
<td>arg0.frequency</td>
<td>String enumeration</td>
</tr>
<tr>
<td>reference</td>
<td>arg0.reference.correlator</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.endpoint</td>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
<td>arg0.reference.interfaceName</td>
<td>String</td>
</tr>
<tr>
<td>requested Accuracy</td>
<td>arg0.requestedAccuracy</td>
<td>int</td>
</tr>
</tbody>
</table>

**Interface: TerminalLocationNotification**


**locationEnd**

Table 12-30 Interface: TerminalLocationNotification, operation locationEnd

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg1.correlator</td>
<td>String</td>
</tr>
</tbody>
</table>

In addition, arg0 (String) contains the URL to the application endpoint.
In addition, arg0 (String) contains the URL to the application endpoint.

**locationError**

Table 12-31  Interface: TerminalLocationNotification, operation locationError

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg1.correlator</td>
<td>String</td>
</tr>
<tr>
<td>address</td>
<td>arg1.address</td>
<td>java.net.URI</td>
</tr>
<tr>
<td>reason</td>
<td>arg1.reason.messageId</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.reason.text</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.reason.variables</td>
<td>String[]</td>
</tr>
</tbody>
</table>

**locationNotification**

Table 12-32  Interface: TerminalLocationNotification, operation locationNotification

<table>
<thead>
<tr>
<th>Name of parameter in request</th>
<th>Name in policy rules</th>
<th>In policy rules, the parameter is of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>correlator</td>
<td>arg1.correlator</td>
<td>String</td>
</tr>
<tr>
<td>data[]</td>
<td>arg1.data[].address</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].currentLocation.accuracy</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].currentLocation.altitude</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].currentLocation.latitude</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].currentLocation.longitude</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].currentLocation.timestamp</td>
<td>java.util.Calendar</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].reportStatus</td>
<td>RetrievalStatus enumeration</td>
</tr>
<tr>
<td></td>
<td>arg1.data[].errorInformation</td>
<td>ServiceError</td>
</tr>
<tr>
<td>criteria</td>
<td>arg1.criterea</td>
<td>String enumeration</td>
</tr>
</tbody>
</table>
SLA Parameters and Policy enforcement for new plug-ins

In addition, arg0 (String) contains the URL to the application endpoint.
Checklist

This section contains a short summary checklist to use when creating extensions to Network Gatekeeper:

- When creating the management interface, consider if the management operations and attributes should be cluster-wide or local.
- Make sure to follow the plug-in naming convention: Plugin_<web service interface part>_network_protocol_.
- Make sure to implement customMatchCriteria of the ManagedPlugin to be sure that requests end up in the correct plug-in. This is important to be able to run multiple plug-ins for the same traffic path.
- Create exception types that are very specific to various error scenarios. This will allow fine grain control of the alarms that are generated.
- Have a clean separation between the north and the south side of the plug-in.
- Make sure to register all PluginNorth and PluginSouth objects within the ManagedPlugin before registering the plug-in in the PluginManager.
- Make sure to implement the resolveAppInstanceGroupId() method for each PluginSouth instance (if applicable).
- Annotate each parameter in the south object methods that you need to have when aspect calls back the resolveAppInstanceGroupId() or the prepareRequestContext() methods.
Checklist

- Consider what additional EDR fields you need to add. Annotate all the methods you want to be woven using the @Edr annotation.

- Annotate the specific arguments you want to see in the EDR for each annotated methods. Use either @ContextKey or @ContextTranslate depending on the kind of argument.

- Add all the EDRs you are triggering to the edr.xml file, either manually or with the @Edr annotation.