Oracle® Communications IP Service Activator

API Developer's Guide

Release 7.4

E88213-01

December 2017

ORACLE®

Oracle Communications IP Service Activator API Developer's Guide, Release 7.4

E88213-01

Contents

Preface

This guide provides information about developing application programming interfaces (APIs) to Oracle Communications IP Service Activator.

This guide provides information about the following APIs:

- OSS Java Development Library (OJDL) API
- REST API

Audience

This guide is intended for systems integrators and developers who use a supported API to develop interfaces to IP Service Activator. For example, you can use OJDL to develop customized Web-based applications for Customer Network Management.

Readers should have knowledge of:

- The core IP Service Activator features
- The Oracle Solaris operating system and its commands

Accessing Oracle Communications Documentation

IP Service Activator for Oracle Communications documentation, and additional Oracle documentation, is available from Oracle Help Center:

http://docs.oracle.com

Documentation Accessibility

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

Access to Oracle Support

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

1

Working with the OJDL API

This chapter outlines OSS Java Development Library (OJDL) for Oracle Communications IP Service Activator, including the Java classes provided for developers.

The OJDL provides a Java-based Application Programming Interface (API) to IP Service Activator. It includes a set of Java classes with some code samples, and an example web interface.

This chapter assumes you have the following:

- Knowledge of the OSS Integration Manager (OIM), including the External Object Model (EOM), the OIM command language, and the ability to write scripts. See *IP Service Activator OSS Integration Manager Guide* for more information.
- Experience using the Java programming language and Java technologies.

About the OJDL API

The OJDL API is a generic Java API for IP Service Activator, which allows Java developers to develop or customize interfaces, including web-based or intranet-based user interfaces.

The OJDL package includes:

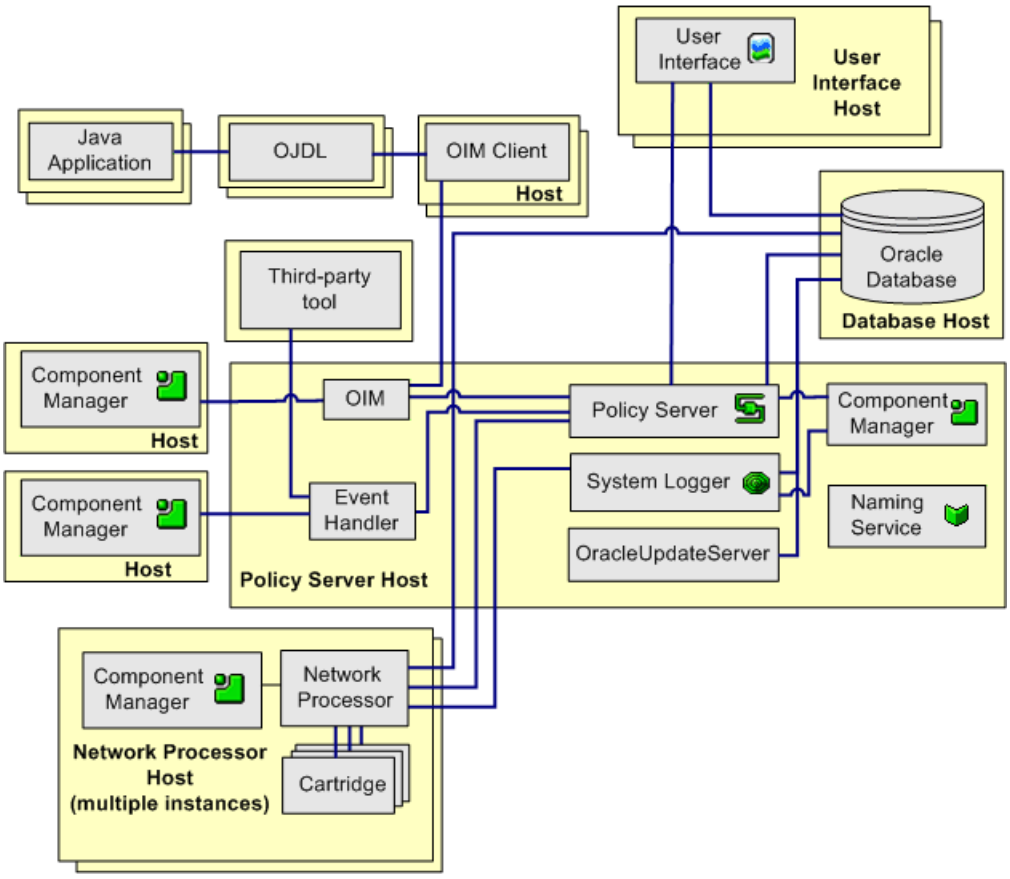
- Java classes
- Java code samples

You can use the OJDL to develop Java-based interfaces that are used to integrate IP Service Activator with components of your environment. These could include, for example, your internal Operational Support System (OSS) environment or an external Customer Network Management solution.

System Architecture

Figure 1–1 shows the relationship between the OJDL and the rest of the IP Service Activator system:

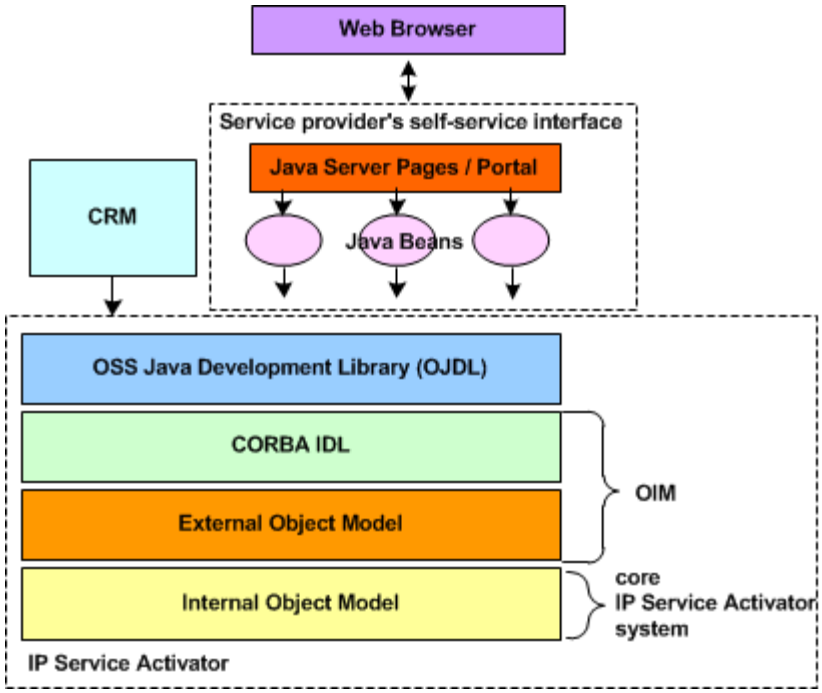
Figure 1–1 OJDL in the IP Service Activator System



The OJDL uses the OSS Integration Manager (OIM) interface and provides access to the External Object Model (EOM), a simplified version of IP Service Activator's internal object model used by the OIM API. The OJDL is OSS compliant.

In effect, the OJDL provides additional layers that are built on top of OIM, which in turn sits on top of the core IP Service Activator system, as shown in Figure 1–2.

Figure 1–2 OJDL in the IP Service Activator Architecture Layers



The EOM is a subset of IP Service Activator's internal object model. It defines all the objects that can be accessed or updated by external applications, including their attributes and the relationships between them. The EOM allows you to create and access data objects without requiring knowledge of the underlying complexity of the entire object model.

The OJDL Java classes provide access to the objects in the EOM. The OJDL provides the same functionality as the OIM CLI, allowing you to create objects, get and set attributes, search for objects, manage transactions, and report errors.

Prerequisites for Installing OJDL

There are no restrictions on where to install the OJDL directory on the host system.

The prerequisites for using the OJDL are:

- Your IP Service Activator installation must include an instance of the OIM. For more information, see *IP Service Activator Installation Guide*.
- If you are developing Java code or running web-based applications, a suitable Java development environment must be installed, such as the Java Platform, Standard Edition (Java SE). For more information, see "Java Development Environment".
- You must have IP Service Activator configured to allow users to log in concurrently, so that you can log in to IP Service Activator using OJDL. By default, the user that is created during IP Service Activator installation does not have this option enabled. For more information about enabling this option, see the section about changing the default user in *IP Service Activator System Administrator's Guide*.

Installing OJDL

The OJDL package is not installed as part of the IP Service Activator standard installation. It is a separate package that you can download from the Oracle software delivery website.

To install the OJDL:

1. Log in to the Oracle software delivery Web site and select **Product Downloads**. Select **Oracle Communications IP Service Activator**, and then select the Components folder for the release that you want.
2. Download the **ojdlpackage-***versionNum***.zip** file available at the following path on the Oracle software delivery website:

   Oracle Communications IP Service Activator Media Pack -> Oracle Communications IP Service Activator Software for Solaris

   where *versionNum* is the version of IP Service Activator.
3. Move the file to the desired directory on the host where you are installing OJDL. There are no restrictions on that directory path that you choose.
4. Unzip the file to create the OJDL directory. The name of the OJDL directory is in the following format: **ojdlpackage-***versionNum*.

   The OJDL directory consists of the following subdirectories:

   - **doc**: Contains the Java documentation (JavaDocs)
   - **lib**: Contains the OJDL jar file that contains the Java classes
   - **samples**: Contains code samples for testing purposes

Configuring SSL for OJDL

To configure SSL for OJDL:

1. Enable SSL for CORBA. See the section on "Enabling SSL for CORBA" in the chapter, "Using the Configuration GUI" in *IP Service Activator System Administrator's Guide* for more information.
2. Ensure that the application classpath includes the following JAR files:

   - lib/jacorb-3.8.jar
   - lib/jacorb-omgapi-3.8.jar
   - lib/slf4j-api-1.7.7.jar

   The JAR files in the preceding list are located in the following IP Service Activator installation directory:

   /opt/OracleCommunications/ServiceActivator/lib/java-lib

   If your development environment is on a separate machine, you must copy these JAR files from an IP Service Activator machine.
3. Depending on your OJDL configuration for SSL, set the system properties custom.props to one of the following:

   - /opt/OracleCommunications/ServiceActivator/Config/jsse_props.tcp
   - /opt/OracleCommunications/ServiceActivator/Config/jsse_props.ssl

     For example:

```
-Dcustom.props=/opt/OracleCommunications736/ServiceActivator/Config/jsse_
props.ssl
```

Using the OJDL API

This section outlines the OJDL, including the Java classes provided for developers.

Java Development Environment

In order to develop Java code you need a suitable development environment, such as the Java Platform, Standard Edition (Java SE), which includes the Java Development Kit (JDK). You can download Java SE from the Oracle Technology Network Web site:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

For information about the recommended JDK version for use with the OJDL, see *IP Service Activator Installation Guide*.

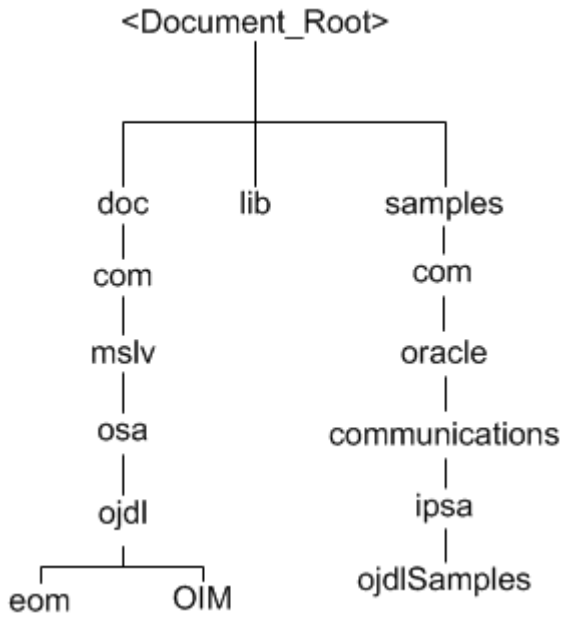
Java SE/JDK can be downloaded free of charge, and can be used for commercial or non-commercial purposes. However, you must retain the copyright notices.

This guide assumes the use of JDK, but other suitable Java tools can be used if required. You need to ensure they are configured correctly.

OJDL Directory and File Structure

When using the OJDL for developing Java code, you need the directories shown in Figure 1–3.

Figure 1–3 OJDL Directories



The doc Directory

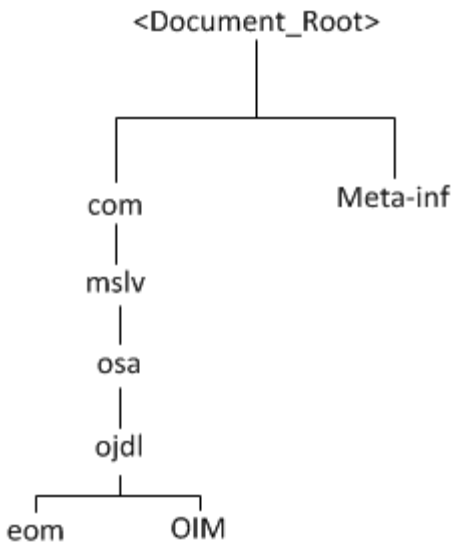
The doc directory includes HTML files that contain class and package information. The **index.html** file lists all the classes and packages, and contains links to access the HTML files that provide the relevant information. The doc directory contains the following two subdirectories:

- **doc\com\mslv\osa\ojdl\eom** contains HTML files describing the EOM Java API classes.
- **doc\com\mslv\osa\ojdl\Oim** contains HTML files describing the OIM Java API subclasses (for the OIM IDL).

The lib Directory

The lib directory includes the **ojdl.jar** file, which contains a compressed version of all the OJDL Java classes. Figure 1–4 shows the internal directory structure of **ojdl.jar**.

Figure 1–4 Internal Directory Structure of ojdl.jar



Note: You must put the **ojdl.jar** file in the class path.

The Samples Directory

The samples directory contains Java code samples, which provide an illustration of the use of the OJDL classes. The Java code samples are available in the following directory:

samples\com\oracle\communications\ipsa\ojdlSamples

For a brief explanation of the code samples, see the **README.txt** file in the ojdlSamples folder.

JavaDocs

The JavaDocs are stored in the doc directory. See "The doc Directory" for more information.

Java Classes

The OJDL Java classes provide access to the EOM objects. The classes can also be used to create Java beans which can then be used to create reusable user interface components for particular tasks. These may be written as Java applications, applets, or as scripts within a Java Server Page.

The OJDL Java classes are stored in the lib directory. See "The lib Directory" for more information.

The Java classes are documented as follows:

- Information about the classes is accessed through the **doc\index.htm** file.
- Details of methods and variables used are contained in the **doc\index-all.htm** file.

The main classes are summarized in Table 1–1. Refer to the JavaDocs for details.

Table 1–1 Main Java Classes

Class	Description
EomAttribute	A base class for representing an attribute within the EOM.
EomAttributesSe	Holds a set of EomAttribute objects.
EomConnectionManager	Defines a connection manager interface for connecting to the OIM.
EomDebug	Provides a way to enable traces.
EomDefaultConnection	The default implementation of EomConnectManager using the JDK ORB.
EomDifferenceResolver	Finds the logical difference of EomObjects contained in two iterators.
EomDiscovery	Enables the discovery of devices.
EomException	The base class for any exception thrown by the OJDL. May be thrown by methods interacting with the OIM.
EomExtendedSearchIterator	Extends the EomSearchIterator by searching on an iterator of EomObjects.
EomIntersectionResolver	Finds the logical intersection of EomObjects contained in two iterators.
EomIterator	An extension of the **java.util.Iterator** interface. Provides a wrapper around the search functionality of the OIM **find** command.
EomIteratorParameters	Provides a way to pass parameters to an EomSearchIterator to refine the search.
EomNonIntegerException	Thrown when a non-integer value is being assigned to an integer variable.
EomObject	A base class for representing objects within the EOM. Each object has an Id, name, and set of attributes.
EomObjectException	Thrown during an EomObject creation.
EomObjectFactory	A factory to build EomObjects.
EomOimException	Thrown when a command cannot be executed by OIM.
EomResolver	A base class for combining the results of two iterator sets.
EomSearchIterator	Looks for all objects of a specific type with a given attribute.
EomSession	Represents a connection to the OIM.
EomSessionException	Thrown by a method in EomSession when connected to the OIM.
EomTransaction	Models the general IP Service Activator transaction concept.
EomTransactionStateChange	A base class that allows IP Service Activator to synchronously return configuration success or failure messages through the OJDL for transactions which perform adds, modifies, or deletes.

Best Practices for Minimizing Commits

It is good to minimize the number of IP Service Activator transaction commits to complete an operation, because each commit introduces a delay when the object model is updated to reflect the new changes.

The following example shows how commits may be minimized when an application generates a large number of devices for testing. These devices are all of the same type and use the device capabilities of an existing device.

To link the desired capabilities, you must first unlink the default capabilities that are linked when the device is created. In the least efficient case, the client code would take three commits; device create, commit, setpath to device and unlink existing caps, commit, link new caps, commit.

In the more efficient form, the client code could accomplish this through one commit by constructing a reference to the default capabilities of the device by appending the following to the path of the device object:

```
/DeviceCapabilities:"DeviceCapabilities"
```

The following excerpt shows how to construct an EomIdentifier that references the capabilities linked to the device when the transaction is committed:

```
EomIdentifier idForNotYetLinkedDefaultCaps = new
EomIdentifier("DeviceCapabilities", "DeviceCapabilities", newDevice.getPath() +
"/" + "DeviceCapabilities" + ":\"DeviceCapabilities\"");
```

Then the default caps are unlinked and the appropriate device caps are linked using the following excerpt:

```
itsTransaction.unlinkObjects(newDevice,
itsSession.createEomObject(idForNotYetLinkedDefaultCaps));
itsTransaction.linkObjects(newDevice, deviceCaps);
tr = eomSession.openTransaction();
tr.commit();
```

Very large transactions can take more time to process after you commit them. This must be balanced against the overall number of commits you issue.

Managing Configuration Policies Using the OJDL API

This section outlines how to use the OJDL API to manage Configuration Policies in Oracle Communications IP Service Activator.

Initial Setup

The following JAR files are required in the application classpath in order to create configuration policies using the OJDL API. They are installed with the Network Processor.

- **servicemodelextensions.jar** contains XML Bean Classes for the Configuration Policy Service Model Extensions.
- **xbean.jar** contains Apache XML Beans API.
- **jsr173_api.jar** contains streaming API for XML, provided as part of the Apache XML Beans API.
- **ojdl.jar** contains IP Service Activator OSS Java Development Library (OJDL) API.

These files are located in the following IP Service Activator installation directory:

- Solaris: **/opt/OracleCommunications/ServiceActivator/lib/java-lib**

If your development environment is on a separate machine you will have to copy the JAR files from an IP Service Activator machine.

Creating a Configuration Policy

Configuration policies are optional XML extensions to the IP Service Activator object model that are supported by the Network Processor cartridges.

A configuration policy can be created using the OJDL API by creating a RuleGeneric object. A RuleGeneric object must have two parent objects: a Policy Type object, and the object to which you want it to apply. The latter object can be an interface, or other applicable objects. For details, refer to the discussion of the RuleGeneric object and the external object model in *IP Service Activator OSS Integration Manager Guide*.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in **samples\com\oracle\communications\ipsa\ojdlSamples**. The Configuration Policy XML definition is set in the RuleGeneric ContentValue attribute.

The structure of the XML for each configuration policy is defined by an XML Schema specification in **servicemodelextension-api-***versionNum.buildNum***.zip**, which is installed with the IP Service Activator client in the *Service_Activator_Home***SamplePolicy** folder. An API is provided to programmatically construct the configuration policy XML data structures using Java XML Beans, using the Apache XML Beans technology available at the Apache web site:

http://xmlbeans.apache.org/

Creating the Configuration Policy Data Type

Each configuration policy top level XML element is represented by an XML Beans Document class. For example, the StaticNats configuration policy is created as a StaticNatsDocument object. Refer to "Configuration Policy Classes" for the complete configuration policy class mapping.

The content of the StaticNats object is set using the XML Beans API.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

Creating the RuleGeneric Object to Contain the Configuration Policy

Configuration Policy objects are represented in the IP Service Activator External Object Model (EOM) as RuleGeneric objects. The following two attributes must be set:

- ContentType: the configuration policy type
- ContentValue: the configuration policy xml string

The ContentValue configuration policy XML is generated by invoking the **toString()** function.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

When passing XML strings into the EOM object attributes, some special characters need to be escaped by pre-pending an additional \ character. For example, \" and \'

must be fully escaped to \\\" and \\\' respectively. This conversion is performed by the **escapeForOIM()** function provided in the example.

Assigning the Configuration Policy to the Required Device and Interface Roles

The RuleGeneric object can be created as a child of many objects in the object hierarchy (as documented in *IP Service Activator OSS Integration Manager Guide*). However, the policy object **Concrete** is applied on any of the Interface objects in the inheritance hierarchy that match the RuleGeneric Roles. The RuleGeneric device and interface roles must match the device and interface roles on the interface where the configuration policy is applied.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

Modifying a Configuration Policy

Modification of a configuration policy involves querying the object model for the current configuration policy definition, modifying the configuration policy, and updating the whole definition back into the object model.

Querying the EOM for the Configuration Policy

The configuration policy XML can be obtained from the RuleGeneric ContentValue parameter. The XML is parsed back into the XML Beans object definition of the service model extension.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

As with creating the configuration policy, the XML content of RuleGeneric is updated to handle the extra escape characters around the \" and \' characters. This conversion is performed by the **unescapeFromOIM()** function.

Modifying the Policy Definition

The configuration policy definition is modified using the XML Bean API for the service model extension documents.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

Registering an Interface Policy

Creating a new interface in IP Service Activator through the OIM and OJDL APIs involves a specialized use of configuration policies with the interface configuration management framework. As with interface management through IP Service Activator, there are three types of interface management interactions:

- Main interface creation
- Subinterface creation
- Interface decoration

Each possible interaction must be registered as an Interface Policy Registration. The Interface Policy Registration objects can either be pre-configured in IP Service Activator, or created using the IP Service Activator APIs.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the nterfaceManagementPolicyExample file in the samples directory.

Once an Interface Policy Registration is used to create or decorate an interface it cannot be modified or deleted until all dependent parent interfaces have been deleted or unlinked from the policy registration.

Creating a Subinterface

This section describes how to create a new interface in IP Service Activator so that the new interface configuration will also be correctly provisioned on the device.

As a prerequisite the appropriate subinterface creation Interface Policy Registration must be created.

Creating the Subinterface Object

Create a new subinterface object under the target interface. For consistency, it is recommended that you create the child subinterface with the correct ifType, although IP Service Activator will update this value on the next device discovery.

Create a new subinterface object under the target interface. For consistency, it is recommended that you create the child subinterface with the correct ifType, although IP Service Activator will update this value on the next device discovery.

The following example shows the creation of a new subinterface:

```
// Create the new subinterface interface object
String subinterfaceName = "Serial1/3.100";
attributes = new EomAttributesSet();
attributes.setAttribute("Type", "32");
EomObject subinterface = tr.createObject(parentInterface, "Subinterface",
  subinterfaceName, attributes);
```

Linking the New Subinterface Object to the Interface Policy Registration

The created subinterface object is linked to the previously defined Interface Policy Registration. The act of linking the policy registration automatically creates a new RuleGeneric configuration policy object with the correct data type settings based on the Interface Policy Registration definition.

The following example shows the linking of the subinterface object with the interface policy registration:

```
// Link the new subinterface object to the interface policy registration
tr.linkObjects(subinterface, registrationPolicy);
tr.commit();
```

The new RuleGeneric object name consists of the interface names with **-Data** appended to it.

Modifying the Interface Configuration Policy Data

The interface management configuration policy does not contain any default settings. These must be manually created using the appropriate XML data structure for the configuration policy data type defined in the interface registration policy. The XML content can be created manually or using the XML Beans API provided.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **InterfaceManagementPolicyExample** file in the samples directory.

Linking the New Subinterface to an Interface Role

Up to this point the new subinterface has only been created in the IP Service Activator object model. Before committing the subinterface creation to the device, the new subinterface object must be linked to an appropriate interface role.

There are code examples available in the additional documentation included with the OJDL libraries. For more information, see the **InterfaceManagementPolicyExample** file in the samples directory.

(Optional) Discovering the Device

Optionally, the device can be re-discovered to align any interface changes (such as to the ifType or VC objects) with the object model. For interface types that have child VC object created by the configuration (such as the framerelay DLCI) device re-discovery is recommended.

The following example shows the optional device discovery:

```
// Optionally rediscover the device the get any VC level objects (in this example
// the DLCI)
eomSession.sendCommandtoOIM("discover " + parentDeviceId);
```

Creating a Main Interface

The steps for main interface creation are largely the same as for subinterface creation. For a main interface, the new interface is created as a child of the device and is linked to an appropriate Interface type Interface Policy Registration object.

When creating a main interface, the Interface Policy Registration must define the default capabilities that the interface (and its sub-interfaces and VCs) will be assigned. If the default settings are used, the created interface will not have any capabilities assigned and a capabilities reset and re-discovery must be performed instead.

Decorating an Interface

For interface decoration, follow the same steps as with subinterface creation, with the exception that the interface does not need to be created first. For interface decoration, the existing interface must be linked to a **Decorate** type Interface Policy Registration object.

Comparing Created and Discovered Interfaces

It is possible to determine if an interface was created using the IP Service Activator interface configuration management framework or was initially discovered from the device by inspecting the IsConfigurable parameter on the Interface or SubInterface object.

If IsConfigurable is set to **True** then the interface was created within IP Service Activator. If it is set to **False** then the interface was added through discovery.

Configuration Policy Classes

Table 1–2 lists the configuration policy classes.

Table 1–2 Configuration Policy Classes

Extension	Configuration Policy	Java XMLBeans Class
AtmPvcVcClassModule	atmPvcVcClass	com.metasolv.serviceactivator.atmpvcvcclass.AtmPvcVcClassDocument
CatOSPolicingRuleModule	catOSPolicingRule	com.metasolv.serviceactivator.catospolicingrule.CatOSPolicingRuleDocument
CiscoEthernetPortCharacteristicsModule	ciscoEthernetPortCharacteristics	com.metasolv.serviceactivator.ciscoEthernetPortCharacteristics.CiscoEthernetPortCharacteristicsDocument
CiscoQosPfcTxPortQueuesModule	ciscoQosPfcTxPortQueues	com.metasolv.serviceactivator.ciscoqospfctxportqueues.CiscoQosPfcTxPortQueuesDocument
DlswModule	dlswDevice	com.metasolv.serviceactivator.dlsw.DlswDeviceDocument
DlswModule	dlswEthernetInterface	com.metasolv.serviceactivator.dlsw.DlswEthernetInterfaceDocument
DlswModule	dlswTokenRingInterface	com.metasolv.serviceactivator.dlsw.DlswTokenRingInterfaceDocument
InterfaceConfigMgmtModule	atmSubInterfaceData	com.metasolv.serviceactivator.subinterface.AtmSubInterfaceDataDocument
InterfaceConfigMgmtModule	backUpInterfacePolicy	com.metasolv.serviceactivator.subinterface.BackUpInterfacePolicyDocument
InterfaceConfigMgmtModule	basicRateInterfaceData	com.metasolv.serviceactivator.subinterface.BasicRateInterfaceDataDocument
InterfaceConfigMgmtModule	ciscoUniversalInterface	com.metasolv.serviceactivator.subinterface.CiscoUniversalInterfaceDocument
InterfaceConfigMgmtModule	dialerInterface	com.metasolv.serviceactivator.subinterface.DialerInterfaceDocument
InterfaceConfigMgmtModule	e1ChannelizedSerialInterface	com.metasolv.serviceactivator.subinterface.E1ChannelizedSerialInterfaceDocument
InterfaceConfigMgmtModule	e1Controller	com.metasolv.serviceactivator.controller.E1ControllerDocument
InterfaceConfigMgmtModule	e3ChannelizedSerialInterface	com.metasolv.serviceactivator.subinterface.E3ChannelizedSerialInterfaceDocument
InterfaceConfigMgmtModule	e3Controller	com.metasolv.serviceactivator.controller.E3ControllerDocument
InterfaceConfigMgmtModule	frSubInterfaceData	com.metasolv.serviceactivator.subinterface.FrSubInterfaceDataDocument
InterfaceConfigMgmtModule	hsrp	com.metasolv.serviceactivator.hsrp.HsrpDocument
InterfaceConfigMgmtModule	loopbackInterfaceData	com.metasolv.serviceactivator.subinterface.LoopbackInterfaceDataDocument
InterfaceConfigMgmtModule	multilinkInterface	com.metasolv.serviceactivator.subinterface.MultilinkInterfaceDocument
InterfaceConfigMgmtModule	plPosInterfaceData	com.metasolv.serviceactivator.subinterface.PlPosInterfaceDataDocument
InterfaceConfigMgmtModule	pppMultilink	com.metasolv.serviceactivator.subinterface.PppMultilinkDocument

Table 1–2 (Cont.) Configuration Policy Classes

Extension	Configuration Policy	Java XMLBeans Class
InterfaceConfigMgmtModule	stm1ChannelizedSerialInterface	com.metasolv.serviceactivator.subinterface.Stm1ChannelizedSerialInterfaceDocument
InterfaceConfigMgmtModule	stm1Controller	com.metasolv.serviceactivator.controller.Stm1ControllerDocument
InterfaceConfigMgmtModule	t1ChannelizedSerialInterface	com.metasolv.serviceactivator.subinterface.T1ChannelizedSerialInterfaceDocument
InterfaceConfigMgmtModule	t1Controller	com.metasolv.serviceactivator.controller.T1ControllerDocument
InterfaceConfigMgmtModule	t3ChannelizedSerialInterface	com.metasolv.serviceactivator.subinterface.T3ChannelizedSerialInterfaceDocument
InterfaceConfigMgmtModule	t3Controller	com.metasolv.serviceactivator.controller.T3ControllerDocument
InterfaceConfigMgmtModule	virtualTemplateInterface	com.metasolv.serviceactivator.subinterface.VirtualTemplateInterfaceDocument
InterfaceConfigMgmtModule	vlanSubInterface	com.metasolv.serviceactivator.subinterface.VlanSubInterfaceDataDocument
InterfaceConfigMgmtModule	vrfExportRouteFilter	com.metasolv.serviceactivator.vrfexportroutefilter.VrfExportRouteFilterDocument
IpsecModule	IPsecModule	com.metasolv.serviceactivator.ipsecmodule.IpsecmoduleDocument
LspModule	lspTunnel	com.metasolv.serviceactivator.lsp.LspTunnelDocument
L2QosModule	rateLimit	com.metasolv.serviceactivator.l2Qos.RateLimitDocument
JuniperQosCosAttachmentModule	juniperQosCosAttachment	com.metasolv.serviceactivator.juniperqoscosattachment.JuniperQosCosAttachmentDocument
MiscPluginsModule	atmVcClass	com.metasolv.serviceactivator.vcclass.AtmVcClassDocument
MiscPluginsModule	banners	com.metasolv.serviceactivator.banner.BannersDocument
MiscPluginsModule	bgpCE	com.metasolv.serviceactivator.bgpce.BgpCEDocument
MiscPluginsModule	dailerList	com.metasolv.serviceactivator.dialerList.DialerListDocument
MiscPluginsModule	dslInterfaceData	com.metasolv.serviceactivator.subinterface.DslInterfaceDataDocument
MiscPluginsModule	extendedAcl	com.metasolv.serviceactivator.extendedAcl.ExtendedAclDocument
MiscPluginsModule	ipPools	com.metasolv.serviceactivator.ippool.IpPoolsDocument
MiscPluginsModule	keyChains	com.metasolv.serviceactivator.keyChain.KeyChainsDocument
MiscPluginsModule	saveConfig	com.metasolv.serviceactivator.saveConfig.SaveConfigDocument
MiscPluginsModule	staticNats	com.metasolv.serviceactivator.staticnat.StaticNatsDocument

Table 1–2 (Cont.) Configuration Policy Classes

Extension	Configuration Policy	Java XMLBeans Class
MiscPluginsModule	staticRoutes	com.metasolv.serviceactivator.staticroute.StaticRoutesDocument
MiscPluginsModule	userAuth	com.metasolv.serviceactivator.userAuth.UserAuthDocument
MiscPluginsModule	userData	com.metasolv.serviceactivator.userData.UserDataDocument
MulticastModule	multicastAutoRp	com.metasolv.serviceactivator.multicast.MulticastAutoRpDocument
MulticastModule	multicastBootstrapRouter	com.metasolv.serviceactivator.multicast.MulticastBootstrapRouterDocument
MulticastModule	multicastDevice	com.metasolv.serviceactivator.multicast.MulticastDeviceDocument
MulticastModule	multicastInterface	com.metasolv.serviceactivator.multicast.MulticastInterfaceDocument
MulticastModule	multicastVrf	com.metasolv.serviceactivator.multicast.MulticastVrfDocument
PrefixListModule	prefixListEntries	com.metasolv.serviceactivator.prefixlist.PrefixListEntriesDocument
QosCosAttachmentModule	qosCosAttachment	com.metasolv.serviceactivator.qoscosattachment.QosCosAttachmentDocument
RoutePolicyModule	bgpRoutePolicy	com.metasolv.serviceactivator.routePolicy.BgpRoutePolicyDocument
RoutePolicyModule	vrfRoutePolicy	com.metasolv.serviceactivator.routePolicy.VrfRoutePolicyDocument
SubInterfaceModule	plSerialInterfaceData	com.metasolv.serviceactivator.subinterface.PlSerialInterfaceDataDocument
ServiceAssuranceModule	collectorParameters	com.metasolv.serviceactivator.collectorParameters.CollectorParametersDocument
ServiceAssuranceModule	netflowParameters	com.metasolv.serviceactivator.netflowParameters.NetflowParametersDocument
ServiceAssuranceModule	rtrResponder	com.metasolv.serviceactivator.rtrr.RtrResponderDocument
SgbpModule	sgbp	com.metasolv.serviceactivator.sgbp.SgbpDocument
SnmpModule	snmpCommunities	com.metasolv.serviceactivator.snmp.SnmpCommunitiesDocument
SnmpModule	snmpHosts	com.metasolv.serviceactivator.snmp.SnmpHostsDocument
VlanModule	vlanDefinitions	com.metasolv.serviceactivator.vlanModule.VlanDefinitionsDocument
VlanInterfaceModule	mgmtVlanInterface	com.metasolv.serviceactivator.vlanInterface.MgmtVlanInterfaceDocument
VlanInterfaceModule	vlanInterface	com.metasolv.serviceactivator.vlanInterface.VlanInterfaceDocument

Table 1–2 (Cont.) Configuration Policy Classes

Extension	Configuration Policy	Java XMLBeans Class
VrfCustomNamingModule	vrfCustomNaming	com.metasolv.serviceactivator.vrfCustomNaming.VrfCustomNamingDocument
VrfIPsecModule	customerIPsec	com.metasolv.serviceactivator.vrfipsec.CustomerIPsecDocument
VrfIPsecModule	publicIPsec	com.metasolv.serviceactivator.vrfipsec.PublicIPsecDocument

Example Source Code

Code examples are available in the additional documentation included with the OJDL libraries.

For configuration policy example source code, see the **StaticNATsConfigurationPolicyExample** file in the samples directory.

For interface management example source code, see the **InterfaceManagementPolicyExample** file in the samples directory.

2

Installing and Configuring the REST API Web Service

This chapter describes how to install and configure the Oracle Communications IP Service Activator REST API web service.

Installing the REST API

To install the IP Service Activation REST API Web service:

1. Install the Oracle JDK and the Java environment.

   When installing Oracle JDK and the Java environment, set the JAVA_HOME path and add JAVA_HOME**/bin** to PATH.

   See the *IP Service Activator Installation Guide* for more information about installing Java for IP Service Activator.

2. Install and configuring Oracle WebLogic Server.

   See "Installing and Configuring Oracle WebLogic Server" for more information.

3. Create a WebLogic Server domain.

   See Oracle WebLogic Server product documentation for more information about creating a WebLogic Server domain.

4. Create a JMS queue in Weblogic Server.

   This JMS queue is the IPSA Web Service action queue.

5. Install IP Service Activator.

   To use the REST API with IP Service Activator, you select the Web Service optional integration component when you run the Oracle Universal Installer. Alternatively, if you are running a silent installation, you use the Web Service response file to install IP Service Activator with REST API capability.

   If the Web Service component was installed on a previous version of IP Service Activator, upgrade to the latest version and then redeploy the REST API web service.

   For more information about installing IP Service Activator, see *IP Service Activator Installation Guide*.

6. Configure IP Service Activator REST API.

   You configure the REST API in IP Service Activator's Configuration GUI. See "Configuring the REST API Web Service" for more information.

7. Deploy the REST API Web service to an Oracle WebLogic Server application server.

   You deploy the REST API from the IP Service Activator's Configuration GUI. See "Deploying and Undeploying Web Services" for more information.

Installing and Configuring Oracle WebLogic Server

You can install Web services on the same server with other IP Service Activator components, or you can install Web services as a standalone component.

Installing Oracle WebLogic Server

When installing Oracle WebLogic Server for the REST API, consider the following:

- Install Oracle WebLogic Server (12.2.1.2 or later). Use the **fmw_12.2.x.y.z_ infrastructure.jar** as the installer.
- Create database schemas using the Repository Creation Utility (RCU), which is included in the WebLogic Server installation. The schemas are required for creating the WebLogic Server domain. Each schema can be used by only one domain. If you create a new domain, you must also create a new schema. Use unique schema names. For complete information about using RCU, see *Oracle Fusion Middleware Creating Schemas with the Repository Creation Utility* at the Oracle Help Center:

  https://docs.oracle.com/middleware/12212/lcm/RCUUG/toc.htm
- When creating the domain, select JRF.

For more information about installing WebLogic Server, see *Fusion Middleware Installing and Configuring Oracle WebLogic Server and Coherence* at the Oracle Help Center:

http://docs.oracle.com/middleware/12212/lcm/WLSIG/toc.htm

Setting Up WebLogic Server Security

To use a REST API Web service, you must set up Secure Sockets Layer (SSL) in WebLogic Server to ensure that all connections are secure and encrypted.

Configuring Identity and Trust Keystores in WebLogic Server

To configure the WebLogic Server to use SSL, you must have an SSL certificate for the server that is running WebLogic Server. Production servers should have a trusted certificate. Lab and testing servers can use self-signed certificates. Use Java to generate custom keystore files and then configure WebLogic Server to use those files.

To configure the WebLogic Server to use custom identity and trust keystore files:

1. Generate custom SSL identity and trust files by entering the following in the Java utility:

```
keytool -genkey -alias mykey -keyalg RSA -keysize 1024
        -sigalg SHA256withRSA -validity 128 -keypass 123456 -keystore
identity.jks -storepass 123456
keytool -export -alias mykey -file root.cer
        -keystore identity.jks -storepass 123456
keytool -import -alias mykey -file root.cer -keystore trust.jks
        -storepass 123456
```

where *mykey* is the key name, and *123456* is the key password.

The system generates the following two files:

- **identity.jks**
- **trust.jks**

2. Copy the **.jks** files to the security directory of the WebLogic Server, for example, **OracleHome/user_projects/domains**/*domain_name*/**security**.
3. Log in to the WebLogic Server.

   The WebLogic Administration Console is displayed.
4. In the left pane, expand **Environment**, and select **Servers**.
5. Select the server where you want to configure the identity and trust keystores.
6. Select **Configuration**, and then select **Keystores**.
7. Select the **Custom Identity and Custom Trust** option, and specify the **identity.jks** and **trust.jks** files that you generated in step 1.

   Note: The key name must match the key password that you entered when you generated the files.

8. Click the **SSL** tab to define the SSL configuration options for the private key alias and password.

For more information about setting up WebLogic Server security, see *Fusion Middleware Administering Security for Oracle WebLogic Server* at the Oracle Help Center:

http://docs.oracle.com/middleware/12212/wls/SECMG/toc.htm

Testing the SSL Configuration

You can test whether SSL is set up correctly in the WebLogic Server.

To test the SSL configuration:

1. In a browser, go to the WebLogic Administration Console, for example, enter:

   http://*hostname*:7001/**console**

   If SSL is configured correctly, the browser connects to the WebLogic Administration Console using a secure connection and the URL changes from http:// to https://, for example:

   https://*hostname*:7002/**console**

   The default port number for SSL is 7002 and if your browser connects, SSL is configured correctly.

   Note: If you are using a self-signed certificate for authentication, the browser might need to import the certificate before you make the connection.

Security and Authentication

The REST API Web service supports only secure connections with authentication.

When you deploy the REST web service, the system creates and configures a WebLogic group called **IpsaDomainController**. The REST web service user, which is called **ipsa_ws_user** by default, is configured in the **Web Service/Common** section of the IP Service Activator Configuration graphical user interface (GUI). This web service user is automatically added to the IpsaDomainController group. See "Deploying and Undeploying Web Services" for information about deploying the Web service for use with the REST API.

The REST Web service is configured to accept calls only from a user that belongs to the IpsaDomainController group, as authenticated by WebLogic Server, or is a specific user with the name IpsaDomainController.

You can configure additional users and add them to this group using the WebLogic Server Administration Console.

For more information about setting up WebLogic Server security, see *Fusion Middleware Administering Security for Oracle WebLogic Server* at the Oracle Help Center:

`http://docs.oracle.com/middleware/12212/wls/SECMG/toc.htm`

Configuring the REST API Web Service

You use the IP Service Activator Configuration GUI to configure the REST API Web service and deployment parameters.

> **Note:** The database and CORBA components must also be configured for the Web service to function correctly. See *IP Service Activator System Administrator's Guide* for information about configuring other components using the Configuration GUI.

To configure the REST API web service:

1. Open the IP Service Activator Configuration GUI.

   If you installed the Web Services component during IP Service Activator installation, the IP Service Activator Configuration GUI displays the **Web Service** folder in the tree view.

   See *IP Service Activator System Administrator's Guide* for more information.

2. In the Configuration GUI tree view, double-click the **Web Service** folder.
3. Click **Common**.
4. Enter the configuration parameters.

   For information about Web service parameters, see *IP Service Activator Installation Guide*, Post-Installation Tasks.

> **Note:** If you change IP Service Activator Web service parameters, re-deploy the Web service to ensure that the changes take effect. See "Deploying and Undeploying Web Services" for more information.

Table 2–1 describes the Web service configuration parameters.

Table 2–1 Web Service Configuration Parameters

Parameter	Description
IPSA ORB Initial Host	The host machine for IPSA CORBA. Default is **127.0.0.1**.
IPSA ORB Initial Port	The host port for IPSA CORBA. Default is **2809**.
Database Server IP Address	Database server IP address.
Database Server Port	The database server port. Default is **1521**.
Database Service Name	The database service name. Default is **IPSA.WORLD**.
Database User Id	The database user ID. Default is **admin**.
Database User Password	The database user password.
Confirm Database User Password	Re-enter the database user password.
IPSA User Name	The IP Service Activator user name. Default is **admin**.
IPSA User password	The IP Service Activator web service user password.
Confirm IPSA User password	Re-enter the IP Service Activator web service user password.
IPSA Web Service User Name	The IP Service Activator web service user name. Default is **ipsa_ws_user**.
IPSA Web Service User password	The IP Service Activator web service user password.
IPSA Web Service JMS Queue JNDI	The JMS queue JNDI for the REST API action queue. The default value is: **jms/IPSAJMSQueue**
IPSA Web Service JMS connection factory JNDI	The JMS Connection factory JNDI for the REST API action queue. The default value is: **jms/IPSAConnectionFactory**
Maximum Query Load	The maximum query load in bytes. Default is **1024000**.
EOM Debug Level	Select an option to define the IP Service Activator EOM Debug level. **OFF**: logging is disabled **ERROR**: unexpected exceptions are logged at this level (default) **TRACE**: all logging is enabled. OIM commands and responses are logged at this level. **DEBUG**: lower logging level than Trace **INFO**: informational logging. Lower logging level than Debug.
Maximum Retry on Connection Failure	The maximum number of retries on recoverable conditions, for example, database/OIM failures. Default is **3**.
OIM Session Timeout	OSS Integration Manager session timeout in seconds. Default is **1200**.
OJDL Transaction Short Watch Interval	The OJDL transaction short watch interval in seconds. Default is **5**. Also, you can use this value to configure transaction and discovery monitoring.

Table 2–1 (Cont.) Web Service Configuration Parameters

Parameter	Description
OJDL Transaction Short Watch Period	The OJDL transaction short watch period in seconds. Default is **300**. Also, you can use this value to configure transaction and discovery monitoring.
OJDL Transaction Long Watch Interval	The OJDL transaction long watch interval in seconds. Default is **60**. Also, you can use this value to configure transaction and discovery monitoring.
OJDL Transaction Long Watch Period	The OJDL transaction long watch period in seconds. Default is **3600**. Also, you can use this value to configure transaction and discovery monitoring.
OJDL Transaction Commit Period	The OJDL transaction commit period in seconds. Default is **60**. Also, you can use this value to configure transaction and discovery monitoring. This value is used as the time out value for a transaction commit or for device discovery completion. The REST API posts **Timedout** to the JMS action queue if the time exceeds this value.

Configuring OSS Integration Manager

If you have an OSS Integration Manager (OIM), or multiple OIMs on multiple servers, that you previously installed and configured in IP Service Activator, you can configure the parameters to enable the Web service to interact with those OIMs.

Note: IP Service Activator does not support multiple OIMs on a single server.

Using the **OIM Configuration** component in IP Service Activator Configuration GUI, you can add, delete, and modify the OIM configurations that are used for Web services.

For more information about installing and configuring OIMs in IP Service Activator, see *IP Service Activator OSS Integration Manager Guide*.

To configure OIM for Web services:

1. In the Configuration GUI tree view, double-click the **Web Service** folder.
2. Click **OIM Configuration**.
3. Enter the configuration parameters for the OIM that you want to configure.

   For information about OIM configuration parameters, see Table 2–2.

Table 2–2 OIM Configuration Parameters

Parameter	Description
Name	The CORBA name of the integration manager.
Maximum Sessions	The maximum number of OIM sessions. Default is **10**.
Minimum Idle Sessions	The minimum number of idle sessions. Default is **5**.

Table 2–2 (Cont.) OIM Configuration Parameters

Parameter	Description
Read Only	Select this option if you want to use the integration manager for read only. Deselect this option if you want to use it for both reading and writing.

Deploying and Undeploying Web Services

You use the IP Service Activator Configuration GUI to deploy the REST API Web service and deployment parameters. Deploy the REST API Web service after you configure all parameters, including the deployment parameters, in the IP Service Activator Configuration GUI. For information about Web service parameters, see "Configuring the REST API Web Service". For information about OIM configuration parameters, see "Configuring OSS Integration Manager".

You can also undeploy the Web service.

> **Note:** To configure the Web service deployment, you require information about the WebLogic Server on which the Order and Service Management (OSM) server is deployed. WebLogic Server parameters are required to connect to a Oracle WebLogic Server.

See "Installing and Configuring Oracle WebLogic Server" for information about WebLogic Server. See *Order and Service Management Concepts* and *Order and Service Management Installation Guide* for information about OSM.

To deploy the Web service:

1. In the Configuration GUI tree view, double-click the **Web Service** folder.
2. Click **Deployment**.
3. Enter the configuration parameters for the Web service deployment.
4. Click **Deploy**.

   The configuration tool does the following:

   - Updates the **IpsaWebService.ear** file with the parameter values that you entered in the web service node.
   - Creates a JMS Server, a JMS Module, and JMS queues in WebLogic, if they are not already created.
   - Creates a web service security user group and a user in WebLogic, if they are not already created.
   - Deploys the **IpsaWebService.ear** file to WebLogic.

   For information about web service deployment parameters, see Table 2–3.

Table 2–3 Web Service Deployment Parameters

Parameter	Description
Weblogic Host	The WebLogic host. Default is **127.0.0.1**.
Weblogic Port	The port number for the WebLogic server. Default is **7001**.
Weblogic Admin User Name	The WebLogic administrator user name. Default is **weblogic**.

Table 2–3 (Cont.) Web Service Deployment Parameters

Parameter	Description
Weblogic Admin User Password	The WebLogic administrator user password.
Confirm Weblogic Admin User Password	Re-enter the WebLogic administrator user password.
Weblogic Secure Connection	Select this option if you want to use a secure connection to the WebLogic server. Check box is selected by default.
Weblogic Target Server	The WebLogic target server where you want to deploy the IP Service Activator web service.
Weblogic Home	The directory where WebLogic is installed on the server.

To undeploy the web service:

1. In the Configuration GUI tree view, double-click the **Web Service** folder.
2. Click **Deployment**.
3. Click **Undeploy**.

About Web Service Security

IP Service Activator access control security for Web services determines the functionality to which each user has access. To set up access control security, create a security role. Give this role the privilege to start IP Service Activator Web services. When the Web service client accesses the Web service, the client needs to authenticate itself to the Oracle WebLogic Server hosting IP Service Activator Web service. See *Oracle WebLogic Administration Guide* for information about setting up access security.

Note: Oracle WebLogic access control security protects only WebLogic Server resources and does not cover secure communication with IP Service Activator Web services. As a result, SOAP messages transmitted between the Web service and its clients are in plain text.

The REST API Web service allows only access-level security. Clients must use a user ID that is a member of the IPSA_WS_USERS_GROUP group to communicate with IP Service Activator Web services. The **web.xml** file defines the security role IPSA_WS_USERS and the **weblogic.xml** file defines the security principal name as IPSA_WS_USERS_GROUP.

Running the installer creates a default user. For information about the default user names and passwords used with Web services, see *IP Service Activator Installation Guide*. This user is a member of the IPSA_WS_USERS_GROUP group. Due to limitations of the WebLogic Server console, information created by the command line tools, such as the role name, might not be available on the console.

3

Working with the Programmatic Intent-Based Network REST API

This chapter describes the Oracle Communications IP Service Activator programmatic intent-based network Representational State Transfer (REST) API (application programming interface). You can use the REST API to provision customer-defined services. Also, the IP Service Activator REST API combines a REST architecture with JMS notifications to integrate IP Service Activator with Oracle Communications Order and Service Management (OSM) (or any third-party software solution) and to enable OSM to manage service activation transactions.

About the IP Service Activator REST API

HTTP requests to IP Service Activator execute actions (such as Post, Get, Delete, or Put) against objects defined in custom Groovy scripts. The custom scripts also include information to map the actions and objects to IP Service Activator operations.

If the HTTP request executes an action that initiates a device discovery or a transaction, the IP Service Activator REST API enables you to monitor the result with the JMS action queue.

REST API

You use REST API constraints to create a software architecture style that is based on resources. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it. It is similar to an object in an object-oriented programming language; however, only a few standard methods are defined for a resource, while an object typically has many methods.

In the REST-based architecture, you access resources by using a common interface that is based on HTTP standard. A REST server manages and provides access to the resources, and a REST client accesses and modifies the resources through the common API. The common API is called the REST API and the services that support the API are called the REST web service.

The REST API includes an API Software Development Kit (SDK) that enables you to define API calls with a high level of granularity, which simplifies the logic that is required to provision complex services.

REST API Methods

Every resource supports some or all of the HTTP common methods. A resource is identified by a global ID that is typically a URI. A resource can be in a variety of formats, such as XML, JavaScript Object Notation (JSON), plain text, HTML, and

user-defined data format. A REST client application can require a specific representation format by using the HTTP/HTTPS protocol content negotiation.

The common REST API methods are the following:

- **GET**: Retrieves one or more resources. You can use this method to check the state of a resource.
- **PUT**: Updates a resource. The PUT method updates the full definition of a resource, regardless of what has changed.
- **DELETE**: Removes one or more resources.
- **POST**: Creates a new resource.
- **PATCH**: Updates only the parts of a resource definition that have changed.

A common flow includes using a method to perform an action on a resource, and then using the GET method to check the state of that action. For example, you can create a resource using the POST method (which includes a URI that points to the new resource), and then, because it might take a long time for that resource to be applied to the network, use the GET method to periodically check the state of the new resource. See "About Polling Using the GET Method" for more information.

JMS Action Queue

When the IP Service Activator REST API receives a request that includes an **X-Request-ID** value in the HTTP request header, it saves the **X-Request-ID** value to a correlation ID. During a triggering event (when a transaction is created or when a device discovery is initiated), the REST API sends a JMS message to the JMS action queue with the correlation ID and with a type and a result. The type can hold the values **Transaction** or **Discovery**, and the result can hold the values **Succeeded**, **Failed**, **Invalid**, and **Timedout** (the REST API uses the same time out value for device discovery and for transaction monitoring).

Note: The REST API posts to a single JMS action queue. Multiple IP Service Activator clients can monitor the action queue if you use sufficiently descriptive correlation ID String values.

The REST API can complete a request immediately with an HTTP return code **200** (OK) or with an error code (for example, **400** bad request). The REST API posts nothing to the JMS action queue when a request completes with an error. If the return code is **202** (Accepted or Pending), the REST API posts the state change to the JMS action queue.

Note: The REST API monitoring processes persist information over restarts.

Transactions

If REST methods are intended to modify the system, the system creates transactions. REST methods such as POST, PUT, PATCH, and DELETE can modify the system. If there are no commands in the output map, the system does not need to be modified and no transaction is created. See *IP Service Activator Concepts* for information about transactions.

When commands are generated based on a REST API request and a transaction is generated in IP Service Activator, the REST API periodically polls the transaction status in the Integration Manager. When the transaction completes, the REST API posts the result to the JMS action queue. When Groovy script generation fails or when a command delivery to IP Service Activator fails, the REST API returns an error and no JMS message is sent.

Device Discovery

When an SNMP discovery operation completes, the REST API posts a message to the JMS action queue based on the Boolean value defined for a device object. The Boolean value defined for the device object is called **discovery** and is set to **true** while the SNMP discovery process is in progress. The value is set to **false** when the discovery operation completes.

The REST API returns **Failed** if an error is detected. Otherwise, the REST API returns **Succeeded**. If the device discovery does not complete before the time out expires, then the REST API returns **Timedout**.

The REST API monitors for the following errors:

- IDS_DISCOVERY_ERX_NOT_ALLOWED (1633: Core discovery is not allowed for Juniper E)
- IDS_DISCOVERY_DEVICE_FAIL (1634: Device discovery failed)
- IDS_DISCOVERY_DEVICE_TYPE_NOT_ALLOWED (1635: Core discovery is not allowed for this device type)

Working with the Groovy Scripting Language

Groovy script is a general-purpose scripting language that runs on the Java Virtual Machine (JVM). The syntax that is used for Groovy scripts is similar to the syntax for Java code. Most Java code is also valid Groovy script.

REST resources are mapped to Groovy scripts using a registry. Each REST call is done to a specific resource.

For example, a REST request to activate an Ethernet service could be done using a REST PUT method to a resource called **SCA_ETH_FDFr_EC**. In this example, the URI that is called using the REST service would be the following:

https://*hostname*:**7002/Oracle/CGBU/IPSA/DomainController/resources/data/SCA_ETH_FDFr_EC**.

The first part of the URI references the server with the web service, that is: **https://***hostname*:**7002**. The next part references the IP Service Activator REST API, that is: **Oracle/CGBU/IPSA/DomainController/resources/data**. The last part references the resource, and can also contain a hierarchy, for example, **Ethernet/SCA_ETH_FDFr_EC**. The corresponding Groovy registry entry is like the following example:

```
<groovyScript>
  <name>groovy/Post_SCA_ETH_FDFr_EC.groovy</name>
  <target>SCA_ETH_FDFr_EC</target>
  <operation>POST</operation>
</groovyScript>
```

The registry entry has the following components:

- **Name**: The name of the Groovy script that you want to run. In the example, the Groovy script is contained in a directory.
- **Operation**: The REST methods that are supported by the script. You can include multiple method entries. See "REST API Methods" for supported methods and their definitions.
- **Target**: The resource supported by the script. This can be a single resource (for example, EthernetConnection), or a hierarchy with a resource (for example, Services/Ethernet/EthernetConnection).

The registry is loaded from the following directory: *Service_Activator_home*/**DomainController/groovy.registry**

A sample Groovy registry and Groovy scripts are provided in the *Service_Activator_home*/**ServiceActivator/DomainController/sample** directory. You can copy the registry and scripts directly into the *Service_Activator_home*/**ServiceActivator/DomainController** directory for testing. Sample JSON input is also provided in corresponding .txt files.

Note: When using sample Groovy scripts, you must change the input to match the specific devices and interfaces that are configured in IP Service Activator.

Developing Custom Groovy Scripts

You run Groovy scripts to process REST requests. Groovy scripts interface with IP Service Activator in the Integration Manager by executing commands (find operations, for example) or by adding Integration Manager commands to an **output** variable to be executed in a trackable transaction (using the **X-Request-ID** value specified in the HTTP request header). See *Oracle Communications IP Service Activator OSS Integration Manager Guide* for more information about integration manager commands.

Table 3–1 lists and describes the variables that are available for creating custom Groovy scripts.

Table 3–1 Variables for Creating Custom Groovy Scripts

Variable	Description
json	The input JSON format payload, converted to a map representation. If no JSON payload is provided (for example, with a GET method), this variable is an empty map.
output	An ArrayList of strings. These are the OIM commands that the script will generate. They are processed as a single transaction after the Groovy script returns its results. For information about commands and their formats, see *IP Service Activator OSS Integration Manager Guide*.
returnedJson	A HashMap that is converted to the JSON String returned from the request. This is a component of the Response (javax.ws.rs.core.Response) and is returned with the return code.
uriArray	An array of the elements of the URI. This is useful when you are specifying a hierarchy in the registry with multiple resources mapping to the same script, and enables the script to see what resource and hierarchy it is called with.

Table 3–1 (Cont.) Variables for Creating Custom Groovy Scripts

Variable	Description
queryMap	Map of any query parameters passed on the request. For example, a GET request might specify: **https://.../Layer3Ethernet?Customer=MyCustomer** The parts after the '?' will be parsed into the queryMap.

transactionNameArray	An array of strings that are used when constructing the name of the transaction. This is optional in the Groovy script.
	The following code illustrates how transactions are named:
	`<operation><objectName>_<transNameArray0>_ <transNameArray1>_<timestamp>_<counter>`
	`<operation> : The operation that is performed (Post, Get, Delete).`
	`<objectName>: The object against which the action is performed.`
	`< transNameArrayX>: (Optional) If a string is added to transactionNameArray, each entry is added, followed by an underscore.`
	`<timestamp>: System date in the following format: yyyMMddHHmmssSS`
	`<counter>: A counter to ensure uniqueness of the transaction name. The value can contain up to 3 digits and resets to 0 after 999.`

domainControllerDir	The absolute path to the Service Activator Domain Controller directory. This value is useful when a script calls another script, or when a path name is required.
requestId	The value of **X-Request-ID** from the HTTP request header. This is used to track the request's result up to the JMS action queue, where the ID is used as the correlation ID. Also, this can be used to log messages for reporting.
helper	The API that is provided for assistance. This API has its own javadoc and is provided to facilitate IP Service Activator operations (for example, looking up resources or attributes) and constructing some OIM commands automatically without needing to explicitly create and add them to the output variable.
return	This is not a variable, but is the return code from the script. It is returned as the status of the REST call. If the status is not successfully, for example it is 400 or greater, any IP Service Activator operations are not performed. If the script returns successfully, the REST call might still receive an error if the methods are invalid or if IP Service Activator does not accept the transaction.

Groovy Script Examples

The examples in this section are intended to give further guidance about using the sample Groovy scripts that are provided with IP Service Activator.

Example: Generating CTM Commands

This example implements a REST-based mechanism for generating CTM commands. The sample Groovy script is available in the following location: *Service_Activator_home*/**DomainController/sample/groovy/Post_CTM.groovy**.

Note: Oracle recommends using the POST method to implement this script because generating these commands is similar to the commands for creating a resource, even though this example does not create resources.

Step 1: Configuring JSON

Using JSON format, you must first design the input/output of the service that you want to implement. The input must indicate the template that you want to use (name, versions, driver type, and so on), as well as a list of attributes (name/value pairs).

The input is the following:

```
{
  "templateName":"String",
  "deviceRoll":"String",
  "interfaceRoll":"String",
  "schemaRelease":"Number",
  "templateVersion":"Number",
  "driverType":"String",
  "objectType":"String",
  "interfaceType":"String",
  "templateVariables": {
      "Name1":"Value1",
      "Name2":"Value2",
      …
  }
}
```

In this example, the template name is mandatory and all other template information is optional. You can use wildcards in the CTM call for non-mandatory template information. The structure of the **templateVariables** attribute contains a list of the variables that are part of the template. The list of variables depends on the type of template that you are using. The Groovy script does not enforce the variables in the list; however, CTM generates an error if the variables are incorrect for the template.

The output is the following simple JSON with an array of strings that contain the generated commands:

```
{
    "Commands": [
        "StringCommand1",
        "StringCommand2",
        …
        "StringCommandN"
    ]
}
```

Step 2: Developing the Groovy Script

This section of the example shows the Groovy script that you can develop to accomplish the task of implementing a REST-based mechanism for generating CTM commands.

You can use the helper API to check for the mandatory parts of the incoming JSON code. Create a map that contains the parts of the JSON code that are mandatory, in this example that is the **templateName** and **templateVariables**. Note that this example does not check for specific variable names in the **templateVariables** because these can change based on the specific template.

The input JSON is located in the variable **json**, which is included in the call to **isJsonValid**, as in the following:

```
def expectedJson = [ templateName:"",
                     templateVariables:""
                   ]
if (!helper.isJsonValid(expectedJson, json)) {
    returnedJson.BadRequestErrorType = [ "title":"Exception",
                "detail":'Invalid ctm input json']
    logger.log(Level.SEVERE, "Input is missing required ctm fields")
    return 400;
}
```

The next part of this step is to put the variables into a hashtable so that they can be passed to CTM when generating the template. Groovy provides a way to iterate over the items in the JSON format map, and allows you to add each key/value pair to the new hashtable. Using validation, you can ensure that all the types are strings, in case an incorrect structure was accidentally passed into this part of the JSON. Even numeric fields are strings because JSON format does not differentiate numbers from strings. For example:

```
def Hashtable<String, String> fieldMap = new Hashtable<String, String>()
json.templateVariables.each { key, value ->
    if (value.getClass() == String) {
        fieldMap.put(key, value)
    }
}
```

You create Groovy variables that store the values that are needed to specify the template. In this way, you can also check when a value is not specified and then set it to null. Null is used by the CTM call to indicate that the value should not be used when searching for the template and acts as a wildcard. The template name is also stored in a variable for convenience. For example:

```
def templateName = json.templateName
def deviceRoll = null
if (json.containsKey("deviceRoll")) {
    deviceRoll = json.deviceRoll
}

def interfaceRoll = null
if (json.containsKey("interfaceRoll")) {
    interfaceRoll = json.interfaceRoll
}

def schemaRelease = null
if (json.containsKey("schemaRelease")) {
    schemaRelease = Integer.valueOf(json.schemaRelease)
}
templateVersion = null
if (json.containsKey("templateVersion")) {
    templateVersion = json.templateVersion
}

def driverType = null
if (json.containsKey("driverType")) {
    driverType = json.driverType
}

def objectType = null
```

```
if (json.containsKey("objectType")) {
    objectType = json.objectType
}

def interfaceType = null
if (json.containsKey("interfaceType")) {
    interfaceType = json.interfaceType
}
```

All the data that is required for generating the commands from the template is now prepared, and you can call the CTM method using the helper API. This returns a vector containing the string commands:

```
def Vector<String> cmds = helper.generateCtmCommands(templateName, deviceRoll,
interfaceRoll, schemaRelease, templateVersion, driverType, objectType,
interfaceType, fieldMap)
```

If there is an error and the commands could not be generated, the helper API returns **null**. If this occurs, you can construct a different JSON output that indicates the failure. Use the variable **returnedJson** to construct the JSON map that gets sent back. In this example, the JSON has two elements, a "title" and "detail." It also returns a status 400, which indicates a "Bad Request," because something was wrong with the data that prevented the commands from being generated. This return code is passed back to the calling system.

```
if (cmds == null)
{
    returnedJson.BadRequestErrorType = [ "title":"Exception",
             "detail":'Error generating template']
    return 400
}
```

If the command vector is successfully generated, you can put the vector into the **Commands** element of the returned JSON. The vector maps to an array in the JSON and then returns the status of 200 to indicate that it was successful.

```
returnedJson.Commands = cmds


return 200
```

The result is that the registry is edited with the following entry added for this service:

```
<groovyScript>
  <name>groovy/Post_CTM.groovy</name>
  <target>CTM</target>
  <operation>POST</operation>
</groovyScript>
```

Example: Deleting a Layer 2 Ethernet Service

This example is for deleting a layer 2 service by using the sample that is in the following location: *Service_Activator_home*/**DomainController/sample/groovy/ Post_SCA_ETH_FDFr_EC.groovy**. This example uses Groovy functions.

The first Groovy function is used to substitute all ':' characters in a string with '_' and return the result, as in the following:

```
def String sanitizeIdentifier(String source)
{
    if (source == null)
```

```
        return null
    return source.replaceAll(':', '_')
}
```

One of the results of creating this service is that subinterfaces that do not already exist might be created. This method searches for all the subinterfaces under the customer. Subinterfaces are IP Service Activator objects. For information about finding and retrieving IP Service Activator objects, see *IP Service Activator OSS Integration Manager Guide*.

The following Groovy script searches all the subinterfaces and uses the helper API to find a generic rule with a specific name that matches the one used in the creation of the subinterface. For information about the RuleGeneric object type, see *IP Service Activator OSS Integration Manager Guide*. If the script locates a subinterface, it adds the delete command with the object ID to the variable output. This output is what gets processed when the script returns and performs operations on the OIM.

```
def void deleteGeneratedInterfaces(customerPath)
{
    // First, find all the subs
    subs = helper.findObjects("Subinterface", customerPath, [:])
    for (Map sub : subs)
    {
        // Now look for the subinterface creation policy underneath each
        subifCreation = helper.findObjectPath("RuleGeneric",
                                              sub.id,
                                              ["name":sub.name + "-Data"])
        if (subifCreation != null && !subifCreation.isEmpty())
        {
            output.add("delete [" + sub.id + "]")
        }
    }
}
```

This is a delete method, so there is no JSON output. Instead, the parameters that are used to specify the instance that you want to delete are provided as part of the URI. For example:

https://*hostname*:**7002/Oracle/CGBU/IPSA/DomainController/resources/data/SCA_ETH_FDFr_EC?evcCfgIdentifier=EVC_BOA_001_1_BOA_002**

A "?" separates the resource (SCA_ETH_FDFr_EC) from the parameters. In this example, you must specify a parameter called **evcCfgIdentifier**, which identifies the specific instance of the resource that is to be deleted. Not specifying this parameter results in the following error:

```
if (queryMap.evcCfgIdentifier == null)
{
    returnedJson.BadRequestErrorType = [ "title":"Exception",
                "detail":"No evcCfgIdentifier specified on delete operation" ]
    return 400
}
```

Begin deleting the service by using the **evcCfgIdentifier** parameter in the queryMap. This is a map that is provided with all the query parameters. You can do this using a loop that will support multiple **evcEvcIdentifier** parameters using a single URI.

Run the character conversion function on each identifier (which is also done on create), to get an ID without ':' characters. Then you can search for the customer that matches that ID. If there is no result, you can assume it is already removed and ignore the ID.

In this way, if there is a duplicate or resent request, the system does not generate an error (idempotent).

If you find the customer, you can add a delete method to the output to delete the customer. Doing this also deletes everything contained within that customer (for example, sites, VPNs, and so on). At this point, the customer has not yet been deleted, but the delete command has been added to the output array.

Now you can call the method to remove any generated subinterfaces.

> **Note:** The commands put in the output are buffered and not executed in real time. They are processed only when the script completes, which makes it safe for the method **deleteGeneratedInterfaces** to search on the customer, even though the previous line adds the command to delete the customer to the output. It would not be safe for this method to reference the customer object in anything it added to the output buffer.

```
{
    def String deleteId=sanitizeIdentifier(id)
    String customerPath = helper.findObjectPath("Customer", "/",
                                                ["name":"CE_" + deleteId])
    if (customerPath != null) {
        // Start by removing any created interfaces
        output.add("delete " + customerPath)
        deleteGeneratedInterfaces(customerPath)
    }
}
```

Next, you return a success code 202, accepted for processing. The processing of the output happens in the background after this returns.

```
return 202 // Accepted for processing, not completed
```

Finally, you must add an entry to the Groovy registry for this script, for example:

```
<groovyScript>
  <name>groovy/Delete_SCA_ETH_FDFr_EC.groovy</name>
  <target>SCA_ETH_FDFr_EC</target>
  <operation>DELETE</operation>
</groovyScript>
```

About Polling Using the GET Method

If you're not using the JMS action queue for monitoring, you can poll manually to determine when the state of a resource has changed by using the GET method to retrieve the current state of the resource.

Plan your strategy for using the GET method to poll a resource for a state change (for example, when creating a resource). Running the GET method too frequently can negatively affect system performance, whereas running the GET method too infrequently means the system might not be responsive enough.

Determine the polling strategy by considering how long it takes for the service to typically be applied to the network and routers. For example, if the sample Ethernet service takes a minimum of 20 minutes to apply to the network (with slow routers and low bandwidth), it would not be useful to poll for the status every 20 seconds. In this

case, polling in 5-minute intervals is more acceptable, although the polling interval also depends on the calling system's latency requirements.

About Logging

You can configure logging using the WebLogic Administration Console and the IP Service Activator Configuration GUI. You can configure logging for the REST web service by using Groovy scripts and the Java logging utilities.

Logging Using WebLogic Server Configuration

You can configure and manage logging by using WebLogic Server. You set logging levels in WebLogic for the server that is running the REST web service.

By default, the system logs errors at the ERROR level. When you are troubleshooting REST web service errors, you can change the logging to report DEBUG logs. If you change the logging option, you might have to restart the WebLogic server for changes to take effect.

For more information about setting up logging in WebLogic Server, see WebLogic Server documentation.

Configuring EOM Logging Using the IP Service Activator Configuration GUI

Configuring EOM logging by using the IP Service Activator Configuration GUI provides logging information only about the connection between the REST web service and IP Service Activator.

If you change this configuration, you must redeploy the REST web service for the change to take effect.

See *IP Service Activator System Administrator's Guide* for information about using configuration GUI log files.

Configuring Additional Logging Using Groovy Scripts

You can use Groovy scripts to configure additional logging on the REST web service by using the Java logging framework. This log output is included in the set of logs that is managed by WebLogic Server. See "Working with the Groovy Scripting Language" for more information.

Add logging to a Groovy script by importing the Java logging utilities at the beginning of the script, as in the following:

```
import java.util.logging.Level
import java.util.logging.Logger

def Logger logger = Logger.getLogger("MyClassOrFileName")
```

You can then use the logger within the Groovy script, for example:

```
logger.log(Level.SEVERE, "Error msg")
```